SPRINT 3

RELATORY ESINF

1201518 | ANA ALBERGARIA
1201284 | DIOGO VIOLANTE
1200049 | JOÃO WOLFF
1201592 | MARTA RIBEIRO

G54

CLASS 2DF

PROFESSOR: FÁTIMA RODRIGUES

JANUARY 2022

Índice

I. US301 – import data from countries, ports, borders and seadists files from the database to build a freight network.

2.1 Problem and justification

US301 – import data from countries, ports, borders and seadists files from the database to build a freight network.

- **Contributes:**
  - **Code:**
    - Ana Albergaria, 1201518
    - João Wolff, 1200049
  - **tests and complexity analysis:**
    - João Wolff, 1200049

In order to import the data from countries, ports, borders and seadists from the database to build a freight network, following the given acceptance criterias:

- The capital of a country has a direct connection with the capitals of the countries with which it borders. The ports of a country, besides connecting with all the ports of the same country, the port closest to the capital of the country connects with it; and finally, each port of a country connects with the n closest ports of any other country.

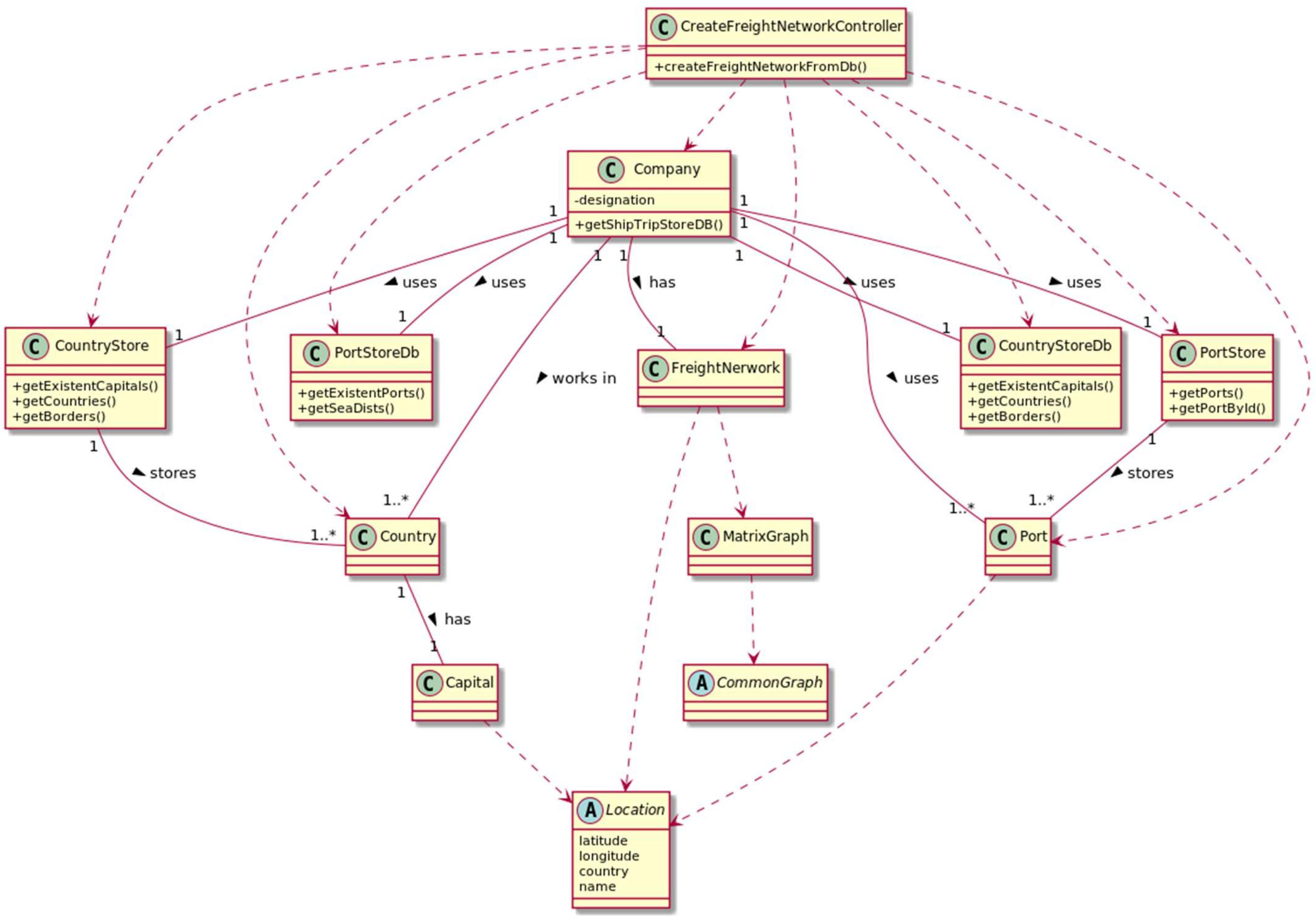
- The calculation of distances in Kms between capitals, and ports and capitals must be done using the GPS coordinates.

- The graph must be implemented using the adjacency matrix representation and ensuring the indistinct manipulation of capitals and seaports.

The generic classes which were developed in ESINF classes Graph, MatrixGraph, Edge and the algorithms were the base to the freight network. to ensure indistinct manupulation of capitals and seaports a superclass Location has been done to be the base to both Capital and Port classes. Next the class freight network was developed holding a MatrixGraph<Location, Double> and all the needed methods for the implementations of the us's.

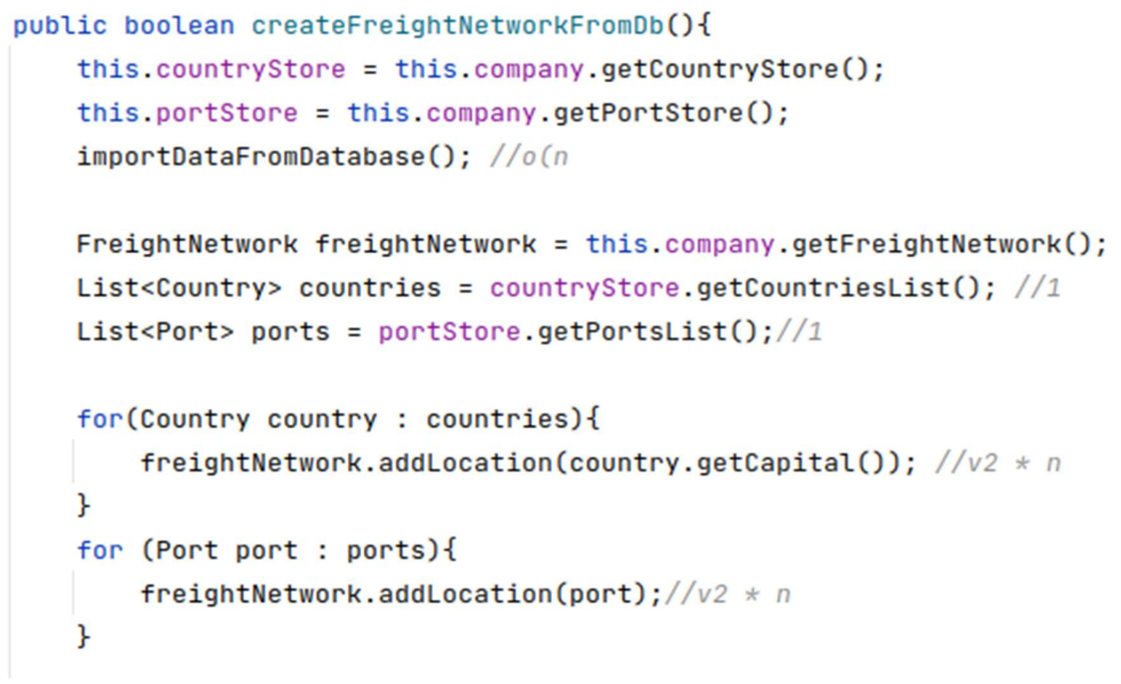
2.2 Class Diagram Analysis

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



2.3 Complexity Analysis

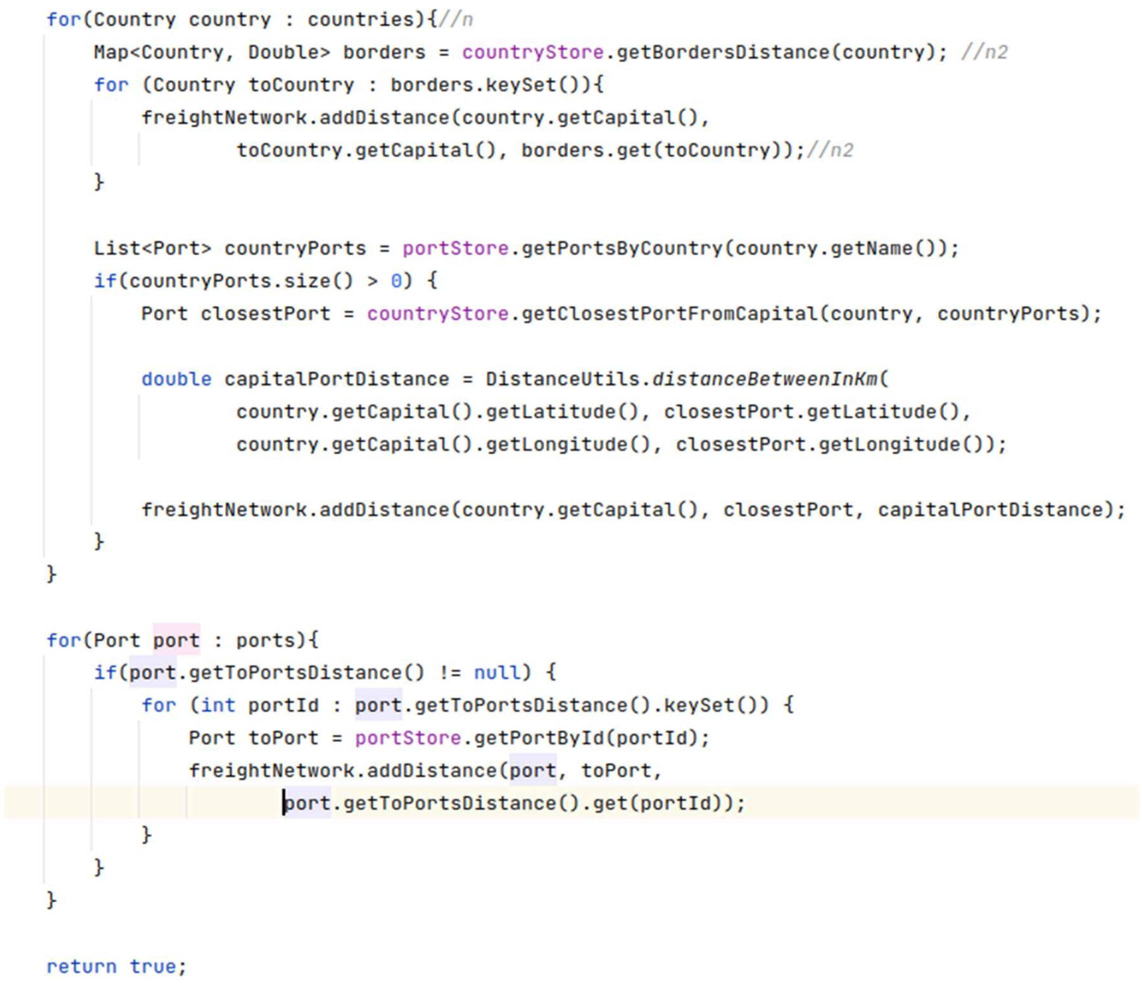
The creation of the freight network is done by one principal method, divided in two parts:



```
public boolean createFreightNetworkFromDb(){
    this.countryStore = this.company.getCountryStore();
    this.portStore = this.company.getPortStore();
    importDataFromDatabase(); //o(n

    FreightNetwork freightNetwork = this.company.getFreightNetwork();
    List<Country> countries = countryStore.getCountriesList(); //1
    List<Port> ports = portStore.getPortsList();//1

    for(Country country : countries){
        freightNetwork.addLocation(country.getCapital()); //v2 * n
    }
    for (Port port : ports){
        freightNetwork.addLocation(port);//v2 * n
    }
```

First of the data has to be imported from the database and saved into local stores as well as added as vertex in the graph of the freight network. just like we can see in the image above.

After that, with the vertex already inserted, the edges are placed in the matrix, respecting the rules given the acceptance criteria. Code which is represented by the block in the image below:



```java
for(Country country : countries){//n
    Map<Country, Double> borders = countryStore.getBordersDistance(country); //n2
    for (Country toCountry : borders.keySet()){
        freightNetwork.addDistance(country.getCapital(),
                toCountry.getCapital(), borders.get(toCountry));//n2
    }

    List<Port> countryPorts = portStore.getPortsByCountry(country.getName());
    if(countryPorts.size() > 0) {
        Port closestPort = countryStore.getClosestPortFromCapital(country, countryPorts);

        double capitalPortDistance = DistanceUtils.distanceBetweenInKm(
                country.getCapital().getLatitude(), closestPort.getLatitude(),
                country.getCapital().getLongitude(), closestPort.getLongitude());

        freightNetwork.addDistance(country.getCapital(), closestPort, capitalPortDistance);
    }
}

for(Port port : ports){
    if(port.getToPortsDistance() != null) {
        for (int portId : port.getToPortsDistance().keySet()) {
            Port toPort = portStore.getPortById(portId);
            freightNetwork.addDistance(port, toPort,
                    port.getToPortsDistance().get(portId));
        }
    }
}

return true;
```

The resultant complexity of this method would be given $O(NV^2)$ given the insertions performed N times, insertion which has a complexity of V^2 in the adjacency matrix.

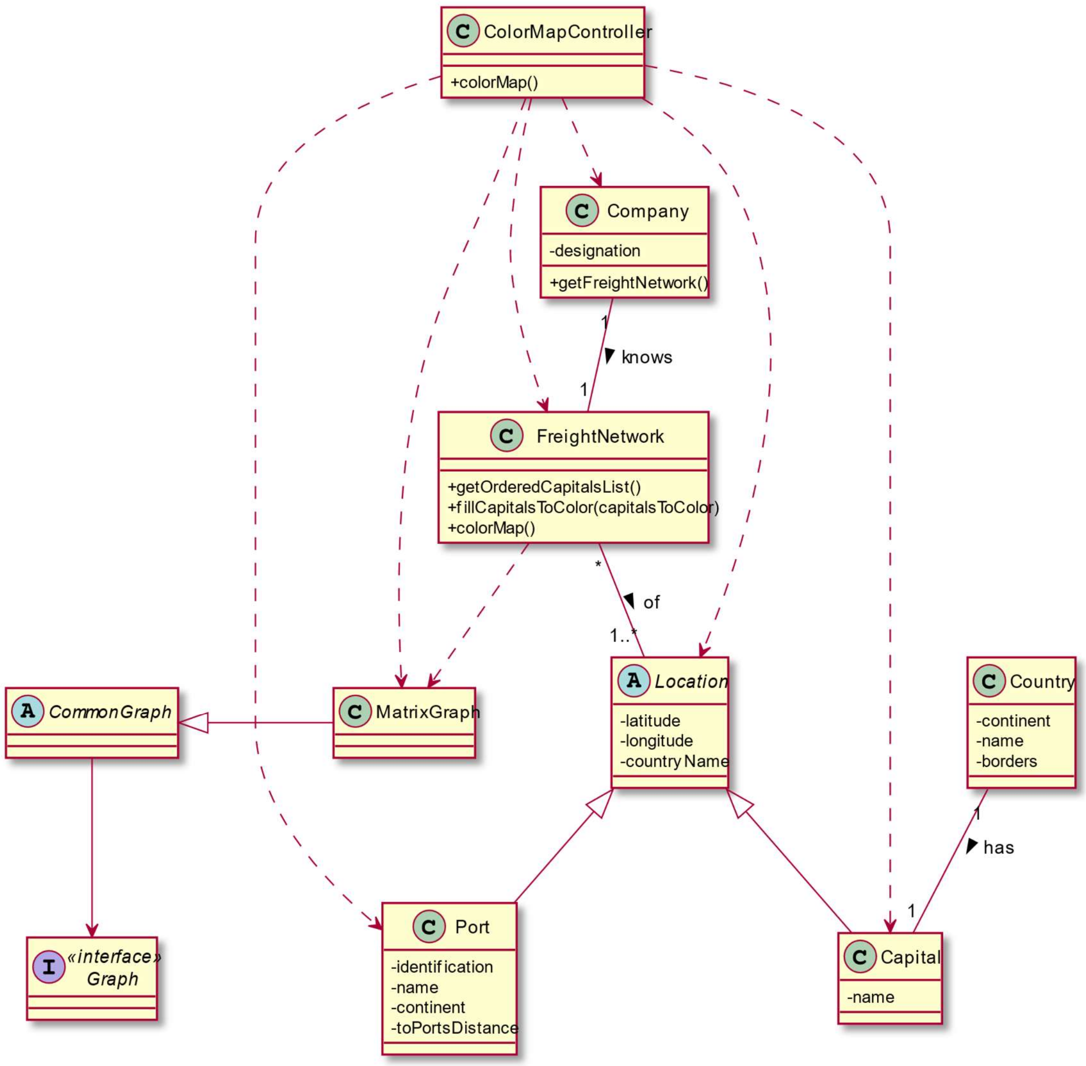
II. US302 – Colour the map using as few colours as possible

2.1 Problem and justification

As a Traffic manager I wish to colour the map using as few colours as possible.

- **Acceptance Criteria:**
  - Neighbouring countries must not share the same colour
- **Contributes:**
  - **Code, tests and complexity analysis:**
    - Ana Albergaria, 1201518

2.2 Class Diagram Analysis



The Class Diagram contains the relevant concepts from the Domain Model for the US302: Company, FreightNetwork, Location, Capital, Port and Country.

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

In order to color the map, the Freight Network must have been built according to the accepting criteria of US301 previously explained. Hence why the classes MatrixGraph, CommonGraph and the interface Graph are presented in the Class Diagram with its respective dependencies with each other. The same applies to the classes Port, Capital and their super class Location.

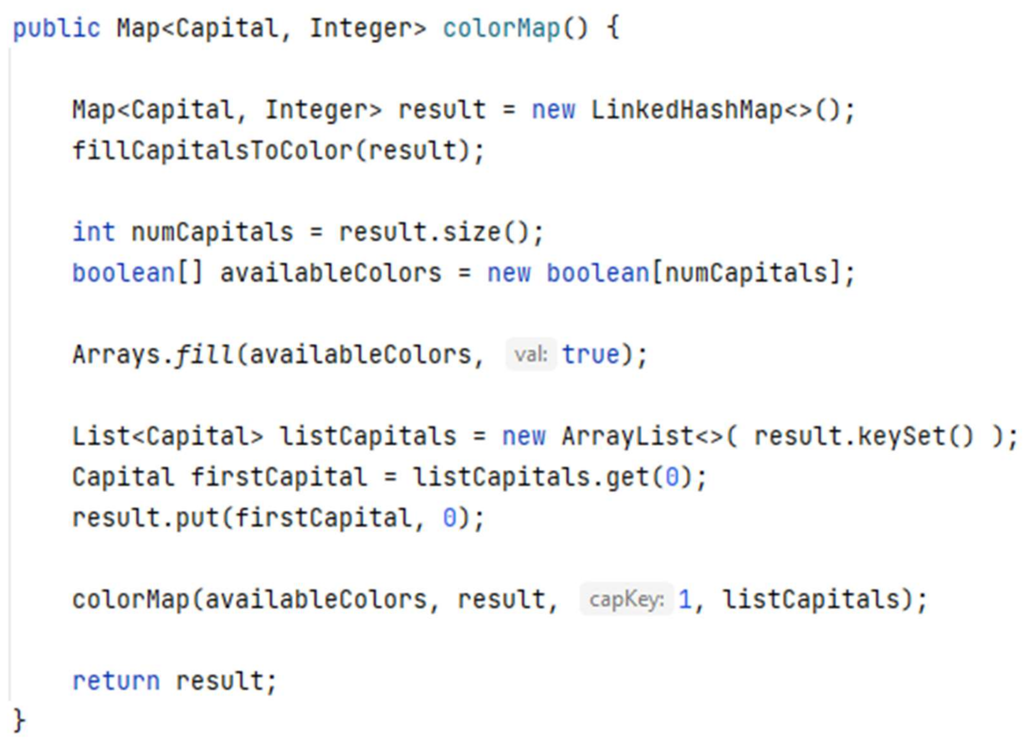
Finally, the class Company must be present as well as this is the class responsible for knowing the Freight Network.

And it will be the class Freight Network the responsible for coloring the Map and all the required methods, once it’s the class who knows all the capitals and connections.

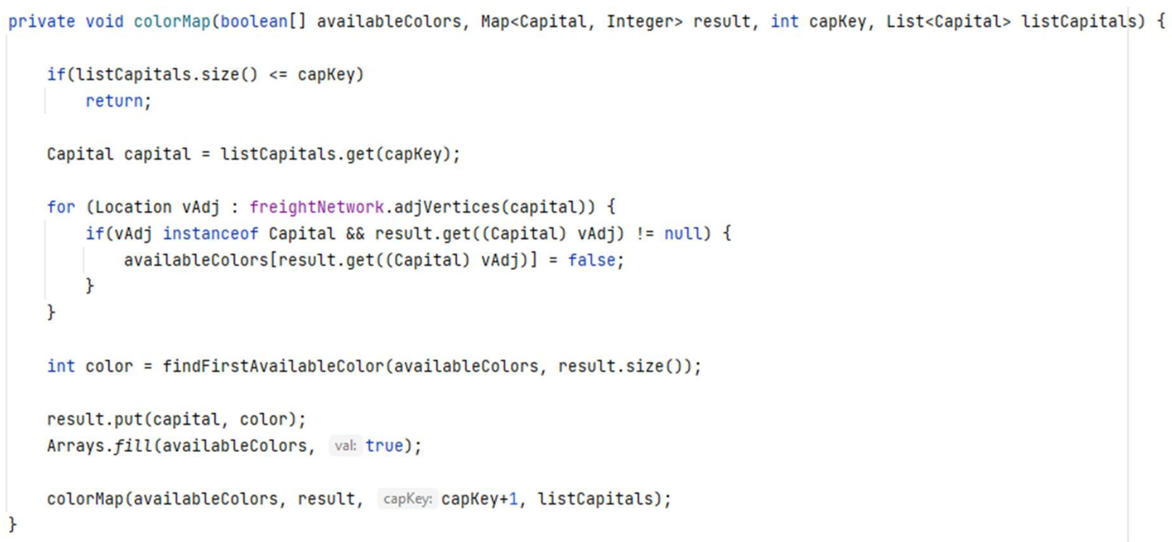
In short, the Class Diagram was designed to ensure the solicited requisites.

2.3 Complexity Analysis

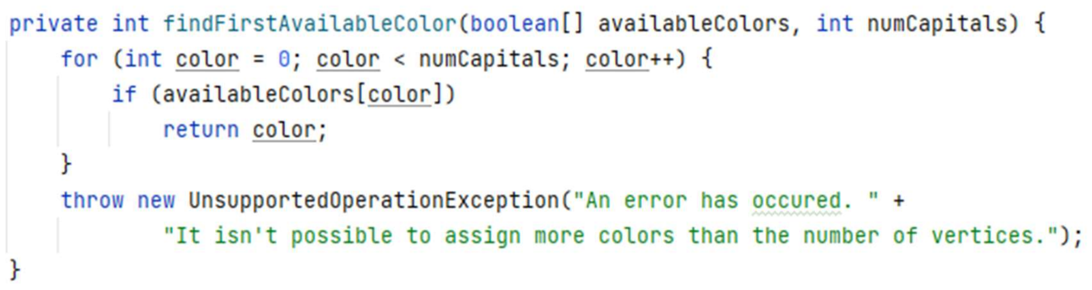
The main methods to color the map are the following:



```java
public Map<Capital, Integer> colorMap() {

    Map<Capital, Integer> result = new LinkedHashMap<>();
    fillCapitalsToColor(result);

    int numCapitals = result.size();
    boolean[] availableColors = new boolean[numCapitals];

    Arrays.fill(availableColors,  val: true);

    List<Capital> listCapitals = new ArrayList<>( result.keySet() );
    Capital firstCapital = listCapitals.get(0);
    result.put(firstCapital, 0);

    colorMap(availableColors, result,  capKey: 1, listCapitals);

    return result;
}
```



```java
private void colorMap(boolean[] availableColors, Map<Capital, Integer> result, int capKey, List<Capital> listCapitals) {

    if(listCapitals.size() <= capKey)
        return;

    Capital capital = listCapitals.get(capKey);

    for (Location vAdj : freightNetwork.adjVertices(capital)) {
        if(vAdj instanceof Capital && result.get((Capital) vAdj) != null) {
            availableColors[result.get((Capital) vAdj)] = false;
        }
    }

    int color = findFirstAvailableColor(availableColors, result.size());

    result.put(capital, color);
    Arrays.fill(availableColors,  val: true);

    colorMap(availableColors, result,  capKey: capKey+1, listCapitals);
}
```



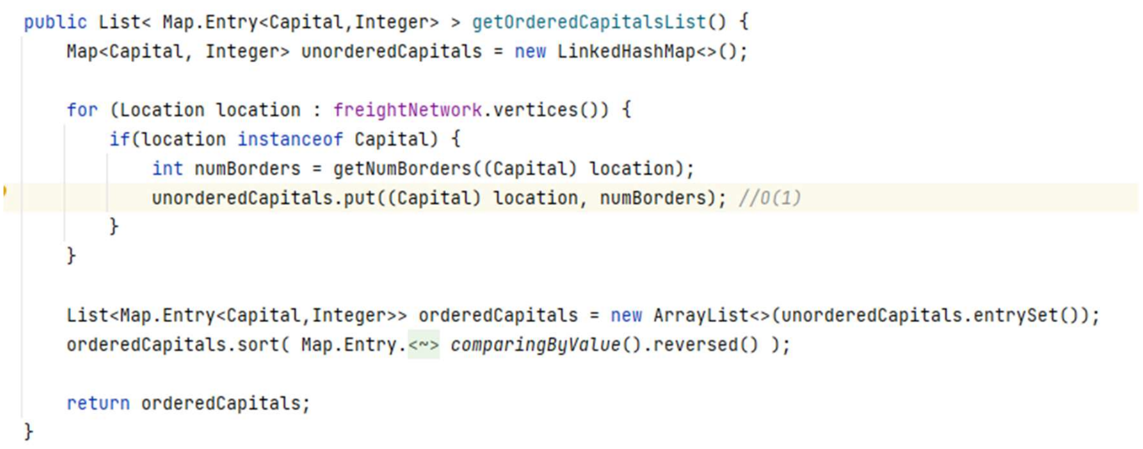
```java
private int findFirstAvailableColor(boolean[] availableColors, int numCapitals) {
    for (int color = 0; color < numCapitals; color++) {
        if (availableColors[color])
            return color;
    }
    throw new UnsupportedOperationException("An error has occured. " +
            "It isn't possible to assign more colors than the number of vertices.");
}
```

This algorithm was based on the known Greedy Algorithm to color a Graph, which works the following way:

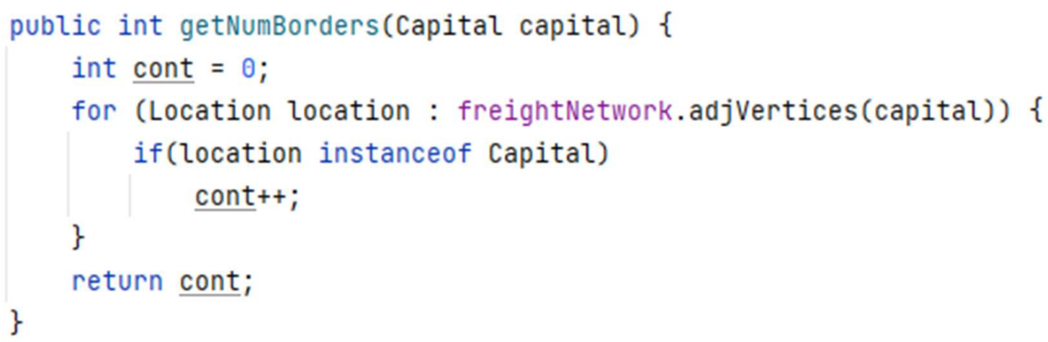
- Color first vertex with first color.
- Do following for remaining V-1 vertices.
    - Consider the currently picked vertex and color it with the lowest numbered color that has not been used on any previously colored vertices adjacent to it. If all previously used colors appear on vertices adjacent to v, assign a new color to it.

However, to make this algorithm more efficient, it is advisable to first order the Capitals by the number of the borders in its decreasing orders, to ensure the acceptance criteria of using a minimum of colors.

Therefore, these two methods were required to order the Capitals:

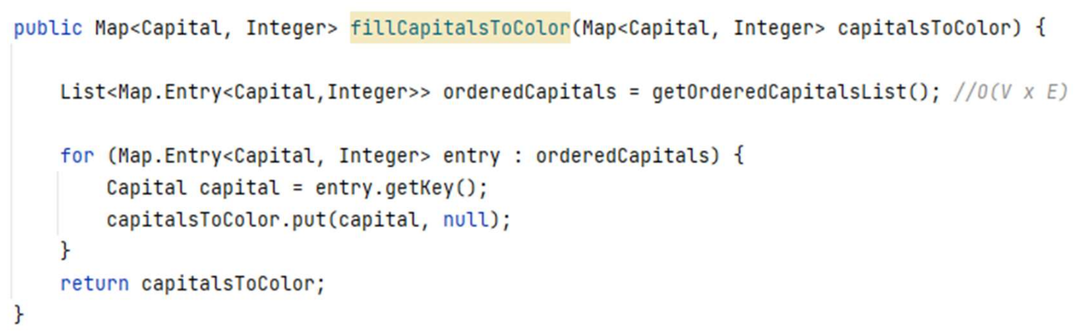


```java
public List< Map.Entry<Capital,Integer> > getOrderedCapitalsList() {
    Map<Capital, Integer> unorderedCapitals = new LinkedHashMap<>();

    for (Location location : freightNetwork.vertices()) {
        if(location instanceof Capital) {
            int numBorders = getNumBorders((Capital) location);
            unorderedCapitals.put((Capital) location, numBorders); //O(1)
        }
    }

    List<Map.Entry<Capital,Integer>> orderedCapitals = new ArrayList<>(unorderedCapitals.entrySet());
    orderedCapitals.sort( Map.Entry.<~> comparingByValue().reversed() );

    return orderedCapitals;
}
```



```java
public int getNumBorders(Capital capital) {
    int cont = 0;
    for (Location location : freightNetwork.adjVertices(capital)) {
        if(location instanceof Capital)
            cont++;
    }
    return cont;
}
```

Having all the Capitals ordered, it was also required another method to fill a map with them and all its colors set to null (which is Complexity O(V)):



```java
public Map<Capital, Integer> fillCapitalsToColor(Map<Capital, Integer> capitalsToColor) {

    List<Map.Entry<Capital,Integer>> orderedCapitals = getOrderedCapitalsList(); //O(V x E)

    for (Map.Entry<Capital, Integer> entry : orderedCapitals) {
        Capital capital = entry.getKey();
        capitalsToColor.put(capital, null);
    }
    return capitalsToColor;
}
```

As it calls the method `getOrderedCapitalsList()`, `fillCapitalsToColor()` method has a complexity of O(V x E).

So in conclusion, the main method:

- For each Capital, it goes through all its adjacent Capitals and for each adjacent Capital, it verifies if any color has been assigned and if so, then that color will be set to unavailable
    - **Time Complexity:** O(V x E).
- After that, it calls the `findFirstAvailableColor` method in order to find the minimum color available not assigned to its adjacent Capitals. As it calls this method for each Capital while it goes through the number of capitals at worst case scenario:
    - **Time Complexity:** O(V^2)
- **Therefore, the final complexity is O(V^2).**

III US303 – Know which places (cities or ports) are closest to all other places (closeness places).

3.1 Problem and justification

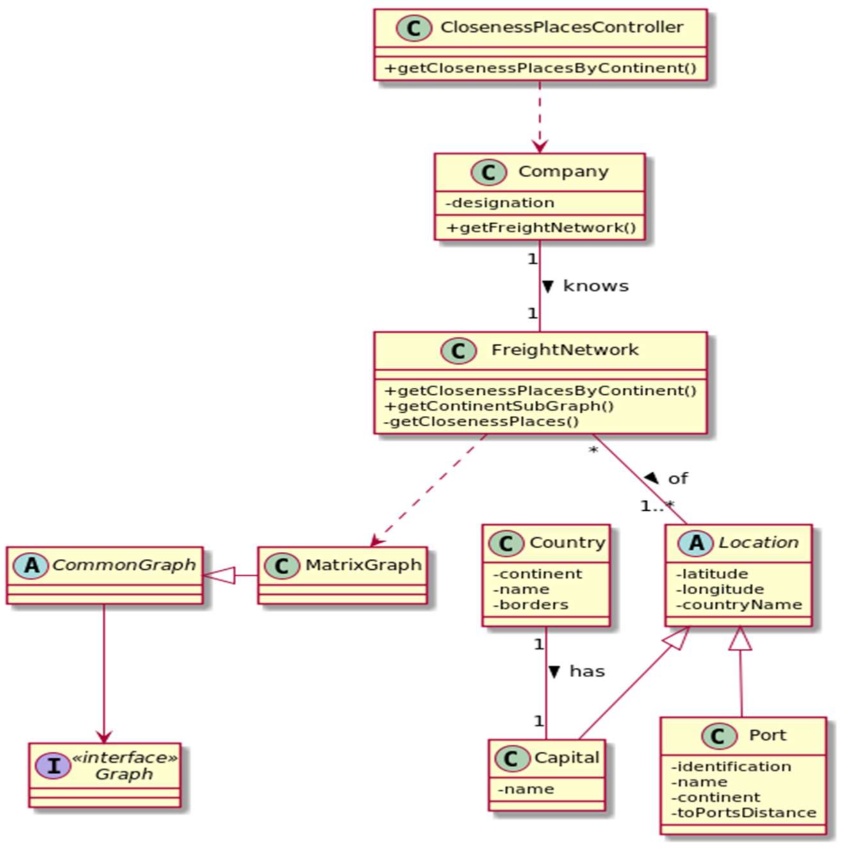
US303 - As a Traffic manager I wish to know which places (cities or ports) are closest to all other places (closeness places).

- **Acceptance criteria:**
  - Return the n closeness locals by continent.
  - The measure of proximity is calculated as the average of the shortest path length from the local to all other locals in the network.

- **Contributes:**
  - **Code, tests and complexity analysis:**
    - João Wolff, 1200049

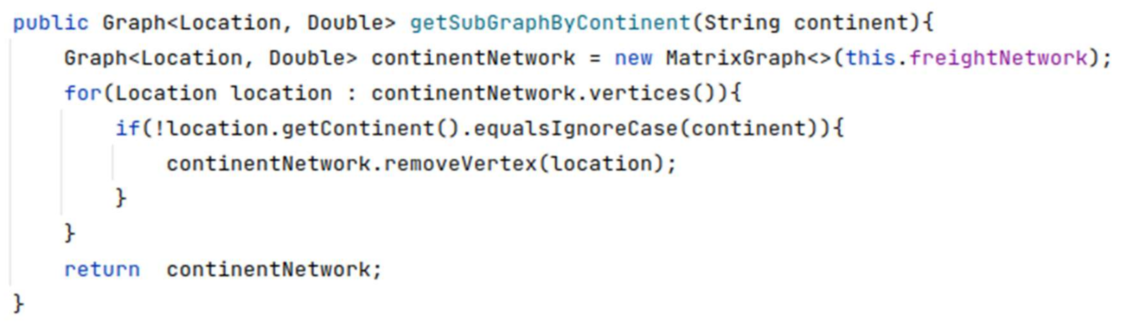
3.2 Class Diagram Analysis

For the us 303 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:



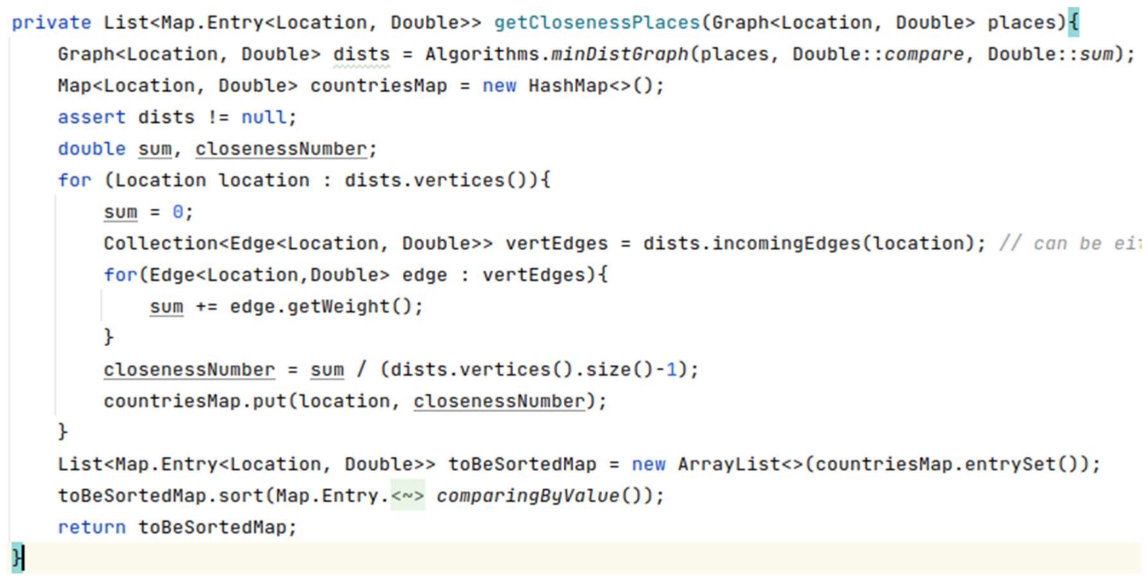
3.3 Complexity Analysis

For solving the problem of returning the n closeness locals by continent, the first step was separating the whole graph by sub-graphs divided by continent. Represented by the code below, with a complexity of O(V).



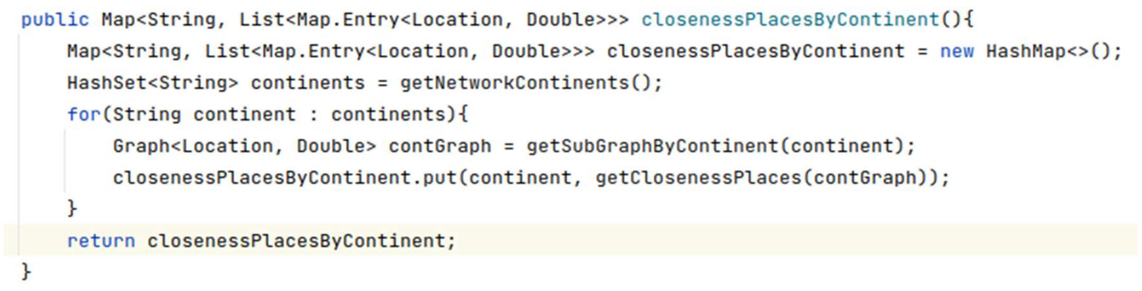
```java
public Graph<Location, Double> getSubGraphByContinent(String continent){
    Graph<Location, Double> continentNetwork = new MatrixGraph<>(this.freightNetwork);
    for(Location location : continentNetwork.vertices()){
        if(!location.getContinent().equalsIgnoreCase(continent)){
            continentNetwork.removeVertex(location);
        }
    }
    return  continentNetwork;
}
```

Then with each continent graph the Floyd Warshall algorithm was used to get the weights adjacency matrix of the continent graph connections. And with this weight graph the mean of each vertex weight is calculated. Just lIke the method below, with complexity O(V³) given the floyd warshall's algorithm call:



```java
private List<Map.Entry<Location, Double>> getClosenessPlaces(Graph<Location, Double> places){
    Graph<Location, Double> dists = Algorithms.minDistGraph(places, Double::compare, Double::sum);
    Map<Location, Double> countriesMap = new HashMap<>();
    assert dists != null;
    double sum, closenessNumber;
    for (Location location : dists.vertices()){
        sum = 0;
        Collection<Edge<Location, Double>> vertEdges = dists.incomingEdges(location); // can be ei
        for(Edge<Location,Double> edge : vertEdges){
            sum += edge.getWeight();
        }
        closenessNumber = sum / (dists.vertices().size()-1);
        countriesMap.put(location, closenessNumber);
    }
    List<Map.Entry<Location, Double>> toBeSortedMap = new ArrayList<>(countriesMap.entrySet());
    toBeSortedMap.sort(Map.Entry.<~> comparingByValue());
    return toBeSortedMap;
}
```

Both of these method are first called and controlled by a third an last method, which iterates through every continent that exists in the whole graph, divides it into subgraphs and appends the return of the closeness places of it into a map to be returned divided by the name of the continent. This can be seen in the following function:



```java
public Map<String, List<Map.Entry<Location, Double>>> closenessPlacesByContinent(){
    Map<String, List<Map.Entry<Location, Double>>> closenessPlacesByContinent = new HashMap<>();
    HashSet<String> continents = getNetworkContinents();
    for(String continent : continents){
        Graph<Location, Double> contGraph = getSubGraphByContinent(continent);
        closenessPlacesByContinent.put(continent, getClosenessPlaces(contGraph));
    }
    return closenessPlacesByContinent;
}
```

Now doing the complexity analysis by the max(O) of this method we can have that the final complexity would be O(NV^3) because of the method getClosenessPlaces inside a loop. Although counting with domain knowledge the N of continents would always be minor or equal to 7, leading to a smaller worst complexity than expected.