ISEP / LEI
ESINF
2021/2022
G76
Autores – Jorge Ferreira (1201564) Rafael Leite (1201566) Rui Pina (1201568)

RELATÓRIO ESINF

 # Abstract #

Uma empresa de logística de transporte requer um Sistema de software capaz de gerir as suas logísticas. Esta empresa opera tanto no mar, como em terra, por vários continentes e possui diversos armazéns espalhados pelo globo.

 # Introdução #

A finalidade deste relatório é de documentar o desenvolvimento das User Stories, no âmbito de ESINF. Todos os passos percorridos para a elaboração de uma solução para os problemas apresentados, assim como a complexidade temporal de cada algoritmo adotado.

De acordo com as boas práticas aprendidas em ESOFT, no semestre passado foi adotado uma metodologia ágil de Scrum e as práticas de Programação Orientada a Objetos.

Com isto as UserStories foram divididas de maneira justa por todos os membros da equipa.

 # Responsáveis das User Stories #

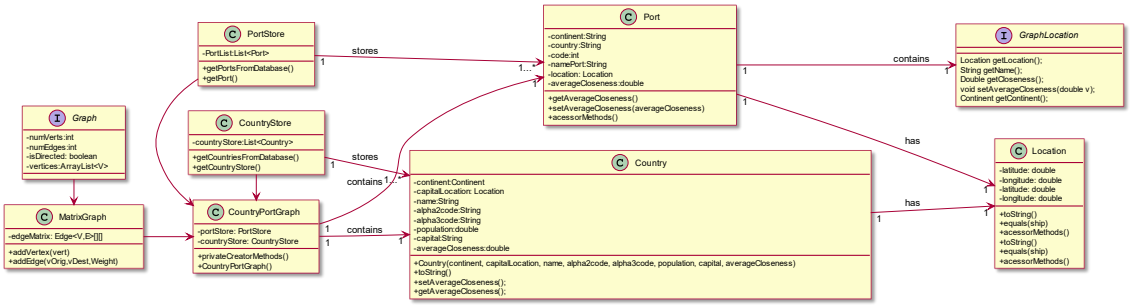
US 301 – Rafael Leite 1201566

US 302 – Jorge Ferreira 1201564

US 303 – Rui Pina 1201568

- **US 301**

Class Diagram



A classe CountryPortGraph vai instanciar um grafo representado através de uma matriz de adjacência com conexões entre as capitais do países que fazem fronteira, os seus portos e as capitais , os portos dentro de os países e finalmente os n portos mais perto uns dos outros no(n inserido pelo utilizador).

A interface GraphLocation é o tipo de dado utilizado no grafo enquanto o peso das arestas são doubles que utilizam a localização que está no método getLocation da interface GraphLocation.

Adição dos países e dos portos

```
public MatrixGraph<GraphLocation,Double> createGraphWithPortsAndCountries(int n){
    MatrixGraph<GraphLocation,Double> graph= new MatrixGraph<>( directed: false);
    countryStore.getCountriesFromDatabase();
    for (Country country: countryStore.getCountryStore()){
        graph.addVertex(country);
    }
    portStore.getPortsFromDatabase();
    for (Port port: portStore.getPortList()){
        graph.addVertex(port);
    }
```

A complexidade temporal é simples O(ncountries+ nports). Este método adiciona os países e os portos ao grafo como vértices.

Criação das fronteiras

```
private void makeBorderEdges(MatrixGraph<GraphLocation,Double> graph){
    DatabaseConnection databaseConnection= new DatabaseConnection( url: "jdbc:oracle:thin:@vsgate-s1.dei.isep.ipp.pt:10713/xepdb1?oracle.net.disableOob=t
    ResultSet rSet;
    try(CallableStatement callStmtAux = databaseConnection.getConnection().prepareCall( sql: "{ ? = call fnc_getAllBorders()}")){
        callStmtAux.registerOutParameter( parameterIndex: 1, OracleTypes.CURSOR);
        callStmtAux.execute();
        rSet = (ResultSet) callStmtAux.getObject( parameterIndex: 1);
        while(rSet.next()){
            Country country1= countryStore.getCountryByName(rSet.getString( columnIndex: 1));
            Country country2= countryStore.getCountryByName(rSet.getString( columnIndex: 2));
            graph.addEdge(country1,country2, Calculator.calculateLocationDifference(country1.getLocation(),country2.getLocation()));
        }
    }catch(SQLException ignored) {
        ignored.printStackTrace();
    }
```

Este método tem complexidade temporal de o(nFronteiras). Este método adiciona as fronteiras como arestas utilizando a distancia entre as capitais como peso.

Ligação entre os portos de cada país e a sua capital

```java
for (Country country:countryStore.getCountryStore()){
    double temp =Double.MAX_VALUE;
    Port portfacade= new Port( continent: "ContinentFacade", country: "CountryFacade", code: 99999, namePort: "PortFacade",new Location( latitude: "-86.6222", longitude: "-128.4
    Location capitalLocation = country.getLocation();
    for (Port port:portStore.getPortList()){
        if (port.getCountry().equals(country.getName())){
            if (Calculator.calculateLocationDifference(capitalLocation,port.getLocation())<temp){
                temp=Calculator.calculateLocationDifference(capitalLocation,port.getLocation());
                portfacade=port;
            }
        }
    }
    if (portfacade.getCode()!=99999){
        graph.addEdge(country,portfacade,temp);
    }
```

Este método tem complexidade temporal de O(nCountries)* O(nPortos). Este método cria a ligação entre os portos de cada país e a sua capital utilizando a diferença entre as locations do método getLocation da interface

Ligação entre os portos do mesmo país

```java
private void makeSameCountryPortDistance(MatrixGraph<GraphLocation,Double> graph){
    DatabaseConnection databaseConnection= new DatabaseConnection( url: "jdbc:oracle:thin:@vsgate-s1.dei.isep.ipp.pt:10713/xepdb1?oracle.net.disableOob=true",
    ResultSet rSet;
    try(CallableStatement callStmtAux = databaseConnection.getConnection().prepareCall( sql: "{ ? = call fnc_getAllPortDistance()}")){
        callStmtAux.registerOutParameter( parameterIndex: 1, OracleTypes.CURSOR);
        callStmtAux.execute();
        rSet = (ResultSet) callStmtAux.getObject( parameterIndex: 1);
        while(rSet.next()){
            Port port1= portStore.getPortByCode(Integer.parseInt(rSet.getString( columnIndex: 1)));
            Port port2= portStore.getPortByCode(Integer.parseInt(rSet.getString( columnIndex: 2)));
            if (port1.getCountry().equals(port2.getCountry())) {
                graph.addEdge(port1, port2, Double.parseDouble(rSet.getString( columnIndex: 3)));
            }
            }

    }catch(SQLException ignored) {
        ignored.printStackTrace();
```

Este método tem complexidade temporal de O(nPortDistance) do ficheiro seadist.csv. Este método cria a ligação entre os portos do mesmo país utilizando como peso a distancia no ficheiro seadist.csv.

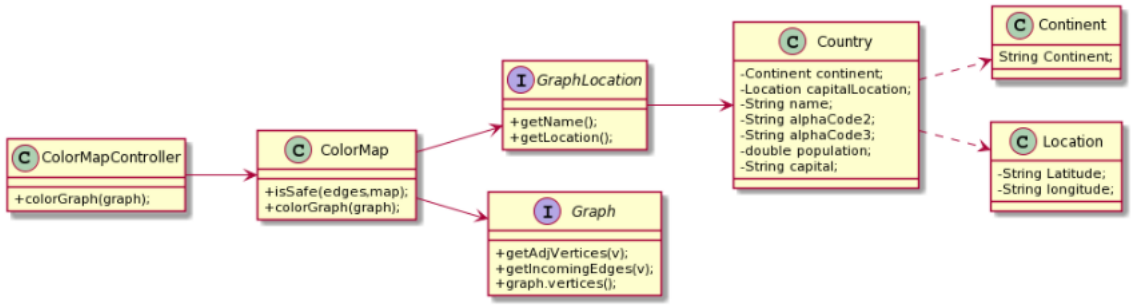
Ligação entre os portos e os n portos mais perto de outros países

```java
private void makeClosestPortOutsideOfCountryDistance(MatrixGraph<GraphLocation,Double> graph,int n){
    for (Port port: portStore.getPortList()){
        List< Pair<Port,Double>> pairList=new ArrayList<>();
        int cont =0;
        DatabaseConnection databaseConnection = new DatabaseConnection( url: "jdbc:oracle:thin:@vsgate-s1.dei.isep.ipp.pt:10713/xepdb1?oracle.net.disableOob=
        ResultSet rSet;
        try (CallableStatement callStmtAux = databaseConnection.getConnection().prepareCall( sql: "{ ? = call fnc_getAllPortDistanceSorted()}")) {
            callStmtAux.registerOutParameter( parameterIndex: 1, OracleTypes.CURSOR);
            callStmtAux.execute();
            rSet = (ResultSet) callStmtAux.getObject( parameterIndex: 1);
            while (rSet.next()) {
            if (rSet.getInt( columnIndex: 1)==port.getCode()&&cont <n){
                Pair<Port,Double> pair= new Pair<>(portStore.getPortByCode(rSet.getInt( columnIndex: 2)),rSet.getDouble( columnIndex: 3) );
                pairList.add(pair);
                cont++;
            }

            }
        for (Pair<Port,Double> pair:pairList){
            graph.addEdge(port,pair.getFirst(), pair.getSecond());
        }
        } catch (SQLException ignored) {
            ignored.printStackTrace();
        }
    }
```

Este método tem complexidade temporal de O(nPorts) *O(nPortDistance).Este método percorro para cada porto uma lista ordenada pelas distancias dos seadists e coloca as primeiras n seadists para esse porto .

- **US 302**

Class Diagram



Na classe ColorMapController, é possível instanciar o método colorGraph, que tem como paramêtro um grafo, neste caso, o grafo criado na US301. Este método retorna um Map, em que a chave é o País, e o valor é a cor do mesmo, representada por um número.

O método colorMap da classe ColorMap, é onde está o código central que permite resolver a esta user storie. Tem também o método isSafe, que é um método auxiliar. O colorMap usa também métodos da interface Graph, trabalhada nas aulas de ESINF.

É possível verificar o funcionamento da US nos testes do ColorMapController.

- **colorGraph**

```
public static Map<GraphLocation, Integer> colorMap(Graph<GraphLocation, Double> G) {

    GraphLocation v = G.vertices().get(0);
    for (GraphLocation vertex : G.vertices()) {
        Collection<GraphLocation> adjV = G.adjVertices(vertex);
        adjV.removeIf(vert -> !vert.getClass().equals(Country.class));
        Collection<GraphLocation> vSize = G.adjVertices(v);
        vSize.removeIf(vert -> !vert.getClass().equals(Country.class));
        if (adjV.size() > vSize.size()) {
            v = vertex;
        } }

    Map<GraphLocation, Integer> coloredMap = new HashMap<>();
    Queue<GraphLocation> queue = new LinkedList<>();
    queue.add(v);

    for (GraphLocation country : G.vertices()) {    // Atribuição de -1 como cor a todos os países (default)
        coloredMap.put(country, -1); }

    int n = 100;                                    // Número máximo de cores
    int colorsCardinality = 0;
```

Inicialmente, o vértice inicial é definido como o país com mais vértices adjacentes, de forma a começar pelo ponto “mais critico”. Esse vértice é adicionado a uma queue. O objetivo é pintar todos os países, de forma que países vizinhos não tenham a mesma cor. É criado o Map, em que inserimos todos os países e definimos a cor de todos como -1 (default).

```
while (!queue.isEmpty()) {
    GraphLocation i = queue.poll();
    Collection<Edge<GraphLocation, Double>> edges = G.incomingEdges(i);
    Collection<GraphLocation> adjV = G.adjVertices(i);

    for (GraphLocation e : adjV) {
        int colorEnd = coloredMap.get(e);
        if (colorEnd == -1) {
            queue.add(e);
        } }

    for (int j = 0; j < n; j++) {
        coloredMap.put(i, j);
        if (colorsCardinality < j) {
            colorsCardinality = j;
        }
        if (isSafe(edges, coloredMap)) {
            break;
        } } }
```

Obtemos as ligações e os vértices adjacentes do primeiro vértice da queue, e se os vértices adjacentes tiverem cor -1, são adicionados à queue para serem os próximos a serem pintados. É atribuído uma cor para cada país, e vai sendo atualizada a cardinalidade da cor. Por fim é instanciado o método auxiliar isSafe. Estas operações são repetidas até não existirem vértices adjacentes pintados, ou seja, até a queue ser vazia.

O método isSafe retorna true se todas as ligações do vértice têm o vértice de origem e

```
public static boolean isSafe(Collection<Edge<GraphLocation, Double>> edges,
                             Map<GraphLocation, Integer> coloredMap) {
    for (Edge<GraphLocation, Double> e : edges) {
        GraphLocation startVertex = e.getVOrig();
        GraphLocation endVertex = e.getVDest();
        int colorStart = coloredMap.get(startVertex);
        int colorEnd = coloredMap.get(endVertex);
        if (colorStart == colorEnd)
            return false;
    }
    return true;
}
```

o vértice de destino com cor diferente. Caso contrário retornará false.

Finalmente, no método ColorMap, passamos para um map todos os vértices que são países e definimos os países cuja cor ainda é (-1), ou seja, países sem vizinhos (p.e. ilhas), como

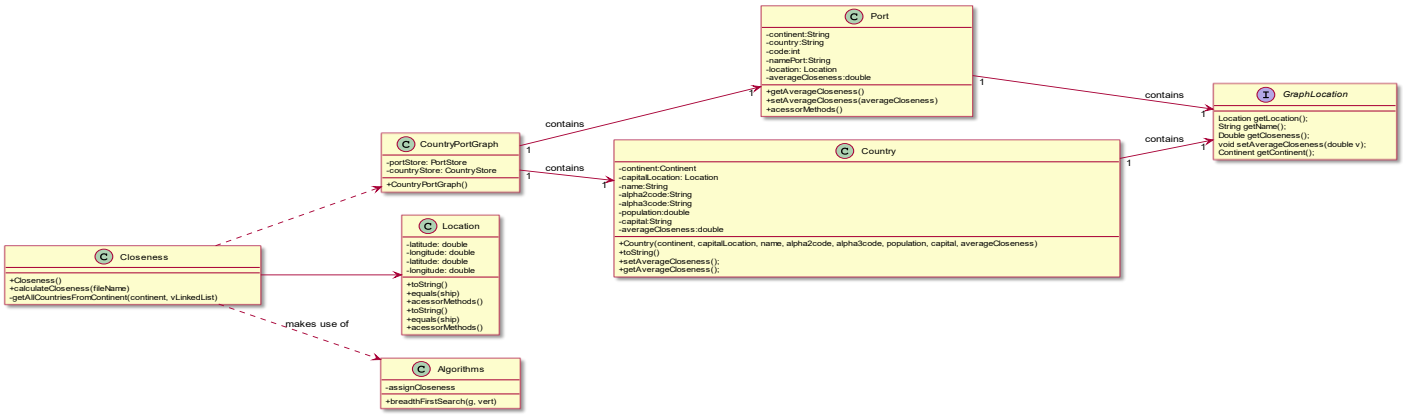
```
    Map<GraphLocation, Integer> countriesColored = new HashMap<>();
    for (GraphLocation g : coloredMap.keySet()) {
        if (g.getClass().equals(Country.class) && coloredMap.get(g) != -1) {
            countriesColored.put(g, coloredMap.get(g));
        } else if (g.getClass().equals(Country.class)) {
            countriesColored.put(g, 0);
        } }

    return countriesColored;
}
```

0, pois é necessário usar o menor número de cores possível.

Complexidade Temporal: O(V + E) (*V -> vértices E -> ligações)*

- **US303**

Class Diagram



A US303 irá utilizar a class CountryPortGraph onde terá as instâncias de country e port que implementam a interface GraphLocation.

Com isto podemos usar os métodos que o CountryPortGraph usa, nomeadamente todos os métodos definidos no “projeto” dos grafos

Algoritmo 1 AssignCloseness

```
private static <V> void assignCloseness(LinkedList<V> vectors) {
    for (V country1 : vectors) {
        GraphLocation country1Casted;
        double cont = 0;
        double sumDistance = 0;

        if(country1 instanceof Country) {
            country1Casted = (Country) country1;
        }else{
            country1Casted = (Port) country1;
        }
        for (V country2 : vectors) {
            GraphLocation country2Casted;

            if(country2 instanceof Country) {
                country2Casted = (Country) country2;
            }else{
                country2Casted = (Port) country2;
            }
            if ((!(country1Casted).getName().equals((country2Casted).getName())) &&
                    (country1Casted).getContinent().getName().equals((country2Casted).getContinent().getName())) {
                sumDistance += Calculator.calculateLocationDifference((country1Casted).getLocation(), (country2Casted).getLocation());
                cont++;
            }

        }
        if (cont != 0) {
            country1Casted.setAverageCloseness(sumDistance / cont);
        }
    }
}
```

Consoante quer o objecto da interface for Country ou Port nós vamos os buscar a todos e calculamos a distância aos restantes que têm o mesmo continente e depois calculamos a média das distâncias do tal instance de GraphLocation aos restantes.

Complexidade temporal: O(V^2).

```
public List<GraphLocation> calculateCloseness(int numberOfGraphs, String continent) {
    vLinkedList = Algorithms.breadthFirstSearch(matrixGraph, matrixGraph.vertex( key: 3));
    assert vLinkedList != null;
    vLinkedList.sort(Comparator.comparingDouble(GraphLocation::getCloseness));
    graphLocations = getAllCountriesFromContinent(continent, vLinkedList);
    if (graphLocations.size() >= numberOfGraphs) {
        return getAllCountriesFromContinent(continent, vLinkedList).subList( i: 0, numberOfGraphs);
    } else {
        return Collections.emptyList();
    }
}
```

Aqui iremos pegar atravessar o grafo, dar set ao closeness a cada Country/Port e dar sort ao ao LinkedList pelo menor closeness. Isto é apenas para apresentar o resultado ao cliente, não é estritamente necessário para o funcionamento do programa. Com isto podemos ver que o breadthFirstSearch confere O(V + E), o sort O(nlogn) e a chamada do getAllCountriesFromContinent O(n)*.

Complexidade temporal: O(nlogn).

*

```
private static List<GraphLocation> getAllCountriesFromContinent(String continent, LinkedList<GraphLocation> vLinkedList)
    List<GraphLocation> countriesInContinent = new ArrayList<>();
    for (GraphLocation graph : vLinkedList) {
        if (graph.getContinent().getName().equals(continent)) {
            countriesInContinent.add(graph);
        }
    }
    return countriesInContinent;
}
```

Time complexity: O(V+E + nlogn + V^2) = 0(V^2).