Instituto Superior de Engenharia do Porto

Projeto Integrador
Sprint 3 - ESINF

03/01

1150609 – Tiago Silva
1220716 - João Botelho
1220962 – Alfredo Ferreira
1220976 - Ricardo Dias

Índice:

Introdução -

Este projeto foi feito perante o enunciado fornecido na unidade curricular de Estruturas de Informação (ESINF), com o objetivo de incentivar os alunos a continuar a desenvolver uma biblioteca de classes, e os respetivos testes, que permitam gerir informação fornecida. Seguindo o exemplo passado, o requisitado era igualmente a navegação nos dados relativos à rede de distribuição de cabazes de produtos agrícolas entre localidades registadas no sistema que podem conter, ou não, "hubs" de distribuição de mercadoria. Nesta parte do projeto, o grupo continuou a desenvolver classes e algoritmos que permitiam uma análise mais detalhada das estruturas de informação denominadas de "grafos", habilitando funcionalidades outrora impossíveis.

Este relatório encontra-se dividido por cada exercício, no qual são apresentados os algoritmos de todas as funcionalidades implementadas e a sua respetiva análise de complexidade. O relatório é também iniciado com um diagrama de classes, no qual são demonstradas as interações entre as funcionalidades incluídas no programa.

Este trabalho foi realizado pelo grupo de trabalho com os elementos: Alfredo Ferreira (1220962), João Botelho (1220716), Ricardo Dias (1220976) e Tiago Silva (1150609); lecionados pelas professoras: Isabel Sampaio (PL), Ana Maria Madureira (TP) e Fátima Rodrigues (T).

USEI06 -

```
public Map<LinkedList<Localidade>, double[]> getAllAutonomousPaths(MapGraph<Localidade, Double> graph, Localidade origin,
                                                                   Localidade destination, double autonomy, double averageSpeed){
    List<LinkedList<Localidade>> paths = allPaths(graph, origin, destination);
    List<Edge<Localidade, Double>> edges = graph.edges().stream().toList();

    Map<LinkedList<Localidade>, double[]> pathInfo = new HashMap<>();
    for(LinkedList<Localidade> a : paths) {
        double total = 0;
        double timeTaken = 0;
        Localidade previousLocation;
        for (int i = 1; i < a.size(); i++) {
            previousLocation = a.get(i-1);
            for(int j = 0; j < edges.size(); j++){
                if(edges.get(j).getVOrig().equals(previousLocation) && edges.get(j).getVDest().equals(a.get(i))){
                    timeTaken += edges.get(j).getWeight()/1000/averageSpeed;
                    total += edges.get(j).getWeight()/1000;
                }
            }
            if(a.get(i).getIsHub() && !a.get(i).getIdLocalidade().startsWith("[")) a.get(i).setIdLocalidade("[HUB] " + a.get(i).getIdLocalidade());
        }
        if(total < autonomy){
            double[] b = {total, timeTaken};
            pathInfo.put(a, b);
        }
        total = 0;
        timeTaken = 0;
    }

    return pathInfo;
}
```

Nesta US é pedido que sejam encontrados, todos os percursos possíveis entre um local de origem e um hub. Estes percursos têm de ser possíveis só com a autonomia dos veículos elétricos, sem carregamentos durante o percurso.

No algoritmo, é primeiro criado uma lista de todas os caminhos entre a origem e o destino, retornadas pelo algoritmo allPaths estudado. Cada caminho será um LinkedList com todas as localidades que foram atravessadas. De seguida, guarda-se todos os edges do grafo para uma lista e é criado um HashMap, do tipo LinkedList<Localidade> e Double[], para poder guardar os caminhos possíveis só com autonomia, na LinkedList, e o tempo que demorou bem como a distância percorrida, no array de Double.

É percorrido cada um dos caminhos e dentro de cada localidade é adicionado ao total o tempo que demorou a chegar a esta e a distância total. Finalmente, se esta localidade for um hub, este irá ser corretamente identificado. No final de cada caminho é verificado se a distância entre todas as localidades, é menor que a autonomia e, se for, é adicionado ao HashMap criado para ser retornado.

Após obter todas os caminhos possíveis, irá retornar o HashMap, que irá ser percorrido e mostrado o caminho obtido, bem como a distância total e o tempo que demorou a fazer esse percurso.

Este algoritmo tem uma complexidade de O(n*m*o), onde n é a quantidade de caminhos, m é a quantidade de localidades no caminho e o é a quantidade de edges no grafo.

USEI07-

```java
public HubsPath maximumHubsPath(double autonomia, Localidade pontoDePartida, double tempoDescarga, double tempoCarga, double velocidadeMedia, String horaInicial) {
    Path optimalHubPath; // Hub possivel de alcançar com autonomia, com maior numero de remainingHubs aberto e que fecha primeiro
    Path absolutePath = new Path(/*Caminho Total, Distância total do percurso, Número de carregamentos, Tempo total */);
    ArrayList<String> arrivalTimes = new ArrayList<>(); //Hora de chegada a todos os locais e hora de partida nos hubs

    Map<Localidade, PathsToHubs> hubsList = createOptimalHubPaths(autonomia, velocidadeMedia, tempoCarga, pontoDePartida);

    do {
        optimalHubPath = getOptimalHubPath(pontoDePartida, hubsList, horaInicial, tempoDescarga, tempoCarga); //Vai buscar os hubs que estão abertos e é possivel lá chegar

        if (optimalHubPath.getDuration()!=0) {
            //Adicionar o caminho ao caminho total
            absolutePath.addPathStops(optimalHubPath.getPathStops());
            absolutePath.addDistance(optimalHubPath.getPathDistance());
            absolutePath.addRechargeStops(optimalHubPath.getRechargeStops());
            absolutePath.addDuration(optimalHubPath.getDuration());

            //Adicionar as horas de chegada a cada local
            addArrivalTimes(arrivalTimes, optimalHubPath, horaInicial, autonomia, velocidadeMedia, tempoCarga, tempoDescarga);

            //Remover o hub visitado
            hubsList.remove(pontoDePartida);

            //Seguir o caminho definido
            pontoDePartida = optimalHubPath.getPathStops().getLast();
            horaInicial = Utils.convertMinutesToTime(caculatesNewStartHour(horaInicial, tempoDescarga, optimalHubPath.getDuration(), tempoCarga));
            absolutePath.addRechargeStop(pontoDePartida);   //Assumo que carrega sempre que chega a um hub.
        }
    } while (hubsList.size() != 1 && optimalHubPath.getDuration()!=0);

    if (hubsList.size() !=1){
        StringBuilder nonVisitedHubs = new StringBuilder();
        nonVisitedHubs.append("Hubs que não foram visitados: ");
        hubsList.remove(pontoDePartida);
        for (Localidade hub : hubsList.keySet()){
            nonVisitedHubs.append(hub.getIdLocalidade()).append(", ");
        }
        nonVisitedHubs.delete(nonVisitedHubs.length()-2,nonVisitedHubs.length());
        nonVisitedHubs.append(".");
        arrivalTimes.add(nonVisitedHubs.toString());
    }
    return new HubsPath(absolutePath, arrivalTimes);
}
```

```java
private Map<Localidade, PathsToHubs> createOptimalHubPaths(double autonomia, double velocidadeMedia, double tempoCarga, Localidade pontoDePartida) {
    Map<Localidade, PathsToHubs> hubsList = new HashMap<>();

    for (Localidade primaryHub : hubs.keySet()) {
        PathsToHubs newHub = new PathsToHubs();
        Set<Path> pathsToOtherHubs = new HashSet<>();
        for (Localidade destinationHub : hubs.keySet()) {
            if (!primaryHub.getIdLocalidade().equalsIgnoreCase(destinationHub.getIdLocalidade())) {
                Path hubPath = new Path();
                if (createOptimalPath(primaryHub, destinationHub, autonomia, velocidadeMedia, tempoCarga, hubPath)) {
                    pathsToOtherHubs.add(hubPath);
                }
            }
        }
        newHub.setPathsToOtherHubs(pathsToOtherHubs);
        hubsList.put(new Localidade(primaryHub), new PathsToHubs(newHub));
    }

    PathsToHubs newHub = new PathsToHubs();
    Set<Path> pathsToOtherHubs = new HashSet<>();
    for (Localidade destinationHub : hubs.keySet()) {
        Path hubPath = new Path();
        if (createOptimalPath(pontoDePartida, destinationHub, autonomia, velocidadeMedia, tempoCarga, hubPath)) {
            pathsToOtherHubs.add(hubPath);
        }
    }

    newHub.setPathsToOtherHubs(pathsToOtherHubs);
    hubsList.put(new Localidade(pontoDePartida), new PathsToHubs(newHub));

    return hubsList;
}
```

```java
private Path getOptimalHubPath(Localidade pontoPartida, Map<Localidade, PathsToHubs> hubsList, String horaInicial, double tempoDescarga, double tempoCarga) {
    Path optimalHubPath = new Path();
    int maxOpenHubs = 0;


    Set<Path> paths = hubsList.get(pontoPartida).getPathsToOtherHubs();

    for (Path path : paths) {
        if (hubIsOpen(path.getPathStops().getLast(), Utils.convertMinutesToTime(caculatesNewStartHour(horaInicial, tempoDescarga: 0.0, path.getDuration(), tempoCarga: 0.0)))) {
            if (hubsList.get(path.getPathStops().getLast())!=null) {
                updateRemainingOpenHubs(horaInicial, hubsList.get(path.getPathStops().getLast()), pontoPartida, tempoDescarga, tempoCarga, path.getDuration());

                if (optimalHubPath.getDuration() != 0) {   //duration==0 é o default
                    if (maxOpenHubs < hubsList.get(path.getPathStops().getLast()).getRemainingOpenHubs()) { //Se o número de remainingHubs anterior for menor, então guarda o novo Path.
                        maxOpenHubs = hubsList.get(path.getPathStops().getLast()).getRemainingOpenHubs();
                        optimalHubPath = new Path(path);
                    } else if (maxOpenHubs == hubsList.get(path.getPathStops().getLast()).getRemainingOpenHubs()) { //Se forem iguais, então compara as horas de fechar.
                        if (checkIfClosesFirst(optimalHubPath.getPathStops().getLast(), path.getPathStops().getLast())) { //Confirma se a hora de fechar do anterior é mais tarde do que o novo.
                            optimalHubPath = new Path(path);     //Se for, muda para o novo Path
                        }
                    }
                } else {            //Ou seja, ainda não tem nenhum Path.
                    maxOpenHubs = hubsList.get(path.getPathStops().getLast()).getRemainingOpenHubs();
                    optimalHubPath = new Path(path);
                }
            }
        }
    }

    return optimalHubPath;
}
```

Este algoritmo procura os melhores caminhos que um certo carro com a sua autonomia, tempo de carga, tempo de descarga e velocidade média tem de percorrer para alcançar o maior número de "hubs" possíveis, antes de estes fecharem.

Este resultado é alcançado através do cálculo inicial do menor caminho possível tendo em conta a autonomia do carro para todos os hubs e a procura de quais é que estão abertos após a viagem deste mesmo. Depois disso, calcula os menores caminhos entre hubs para poder analisar, para cada hub, quantos hubs é possível alcançar antes de estes fecharem. Depois de definir este número, o código procura qual destes tem o maior e caso exista mais do que um com o mesmo, este avalia qual deles fecha primeiro. Após a conclusão de qual é o melhor caminho para prosseguir, o carro "percorre" esse, guardando as informações, eliminando o hub como possibilidade de destino e analisando os novos caminhos, agora através do hub alcançado.

O maior problema deste código é a criação dos vários caminhos. Apesar de ele efetuar este processo apenas uma vez, e depois usufruir de uma lista com todas as informações a seu dispor, o cálculo inicial é extenso e a sua complexidade temporal é afetada negativamente pelo número de "hubs" inicial. Após diversos testes, concluímos que com 5 hubs, este demora 2 segundos, com 10 hubs demora 8 segundos e com 20 hubs demora 30 segundos.

Analisando então a complexidade do algoritmo, o problema principal reside, como dito anteriormente, no método "createOptimalHubPaths", em que são calculados os melhores caminhos entre hubs. Em primeiro lugar, temos um loop que repete H vezes, sendo H o número de Hubs. Dentro desse loop temos outro loop que repete, outra vez, H vezes para fazer a ligação do primeiro hub com os outros todos. Apesar de ele repetir este loop H vezes, ele só repete o seu conteúdo H-1, visto que se o hub do loop interior for igual ao do loop exterior, então este não faz nada. Dentro do loop interior, temos o método "createOptimalPath" que usufriu de uma adaptação ao método Dijkstra para ter em conta a autonomia do carro. A complexidade do método Dijkstra com arestas positivas é O(E*log(V)), contudo, a versão adaptada tem a complexidade de O(E*log(V)+E) visto que tem de testar não só o caminho com menor peso global, mas sim que esteja de encontro a autonomia do carro, o que pode provocar que sejam escolhidas arestas de valores menores para obter este resultado.

Portanto, conseguimos que a complexidade do método "createOptimalHubPaths" é O(H^2*(E*log(V)+E)), o que significa que a complexidade de pior caso é O(N^2*(N*log(N)+N)).

Os restantes algoritmos não apresentam uma complexidade significativa para ser considerada. O máximo que cada um pode ter é percorrer o número de Hubs, tendo isso uma complexidade de O(N), que adicionando ao valor anterior, será sempre desconsiderado.

USEI08 -

```java
public static LinkedList<Localidade> travellingSalesman(int n, String startLoc, double autonomy) throws FileNotFoun
    ImportData.importData();
    int indexHub;
    LinkedList<Localidade> circuito = new LinkedList<>();
    Comparator<Double> comparator = Double::compare;
    BinaryOperator<Double> sumOperator = Double::sum;

    MapGraph<Localidade, Double> graph = ImportData.getGraph();


    Localidade startingPoint = graph.vertex(p -> p.getIdLocalidade().equals(startLoc));
    Localidade previousHub = startingPoint;

    ArrayList<Localidade> hubs = ImportData.getNBiggestHubs(n);
    circuito.add(startingPoint);
    for (int i = 0; i < n; i++) {
        indexHub=-1;
        LinkedList<Localidade> shortestPath = new LinkedList<>();

        double shortest = 999999999;
        LinkedList<Localidade> shortPath = new LinkedList<>();

        for (int j = 0; j < hubs.size(); j++) {
            shortPath = new LinkedList<>();
            Double[] distances = new Double[graph.numVertices()];
            shortPath = shortestPathWithSingleAutonomy(graph, startingPoint, hubs.get(j),  autonomy, distances);
```

```java
            if (pathLength(shortPath, graph) < shortest) {
                indexHub=j;
                shortest = pathLength(shortPath, graph);
                shortestPath = new LinkedList<>(shortPath);
            }
        }

        startingPoint = hubs.get(indexHub);
        hubs.remove(indexHub);
        shortestPath.removeFirst();
        circuito.addAll(shortestPath);
    }
    LinkedList<Localidade> shortPath = new LinkedList<>();
    shortestPath(graph,startingPoint,graph.vertex(p -> p.getIdLocalidade().equals(startLoc)),comparator, sumOperat
    shortPath.removeFirst();
    circuito.addAll(shortPath);

    return circuito;
}

4 usages  Alfredo
public static double pathLength(LinkedList<Localidade> path, MapGraph<Localidade, Double> graph){
    double total=0;
    for (int i = 0; i < path.size()-1; i++) {
        total += Double.parseDouble(String.valueOf(graph.edge(path.get(i),path.get(i+1)).getWeight()));
    }

    return total;
}
```

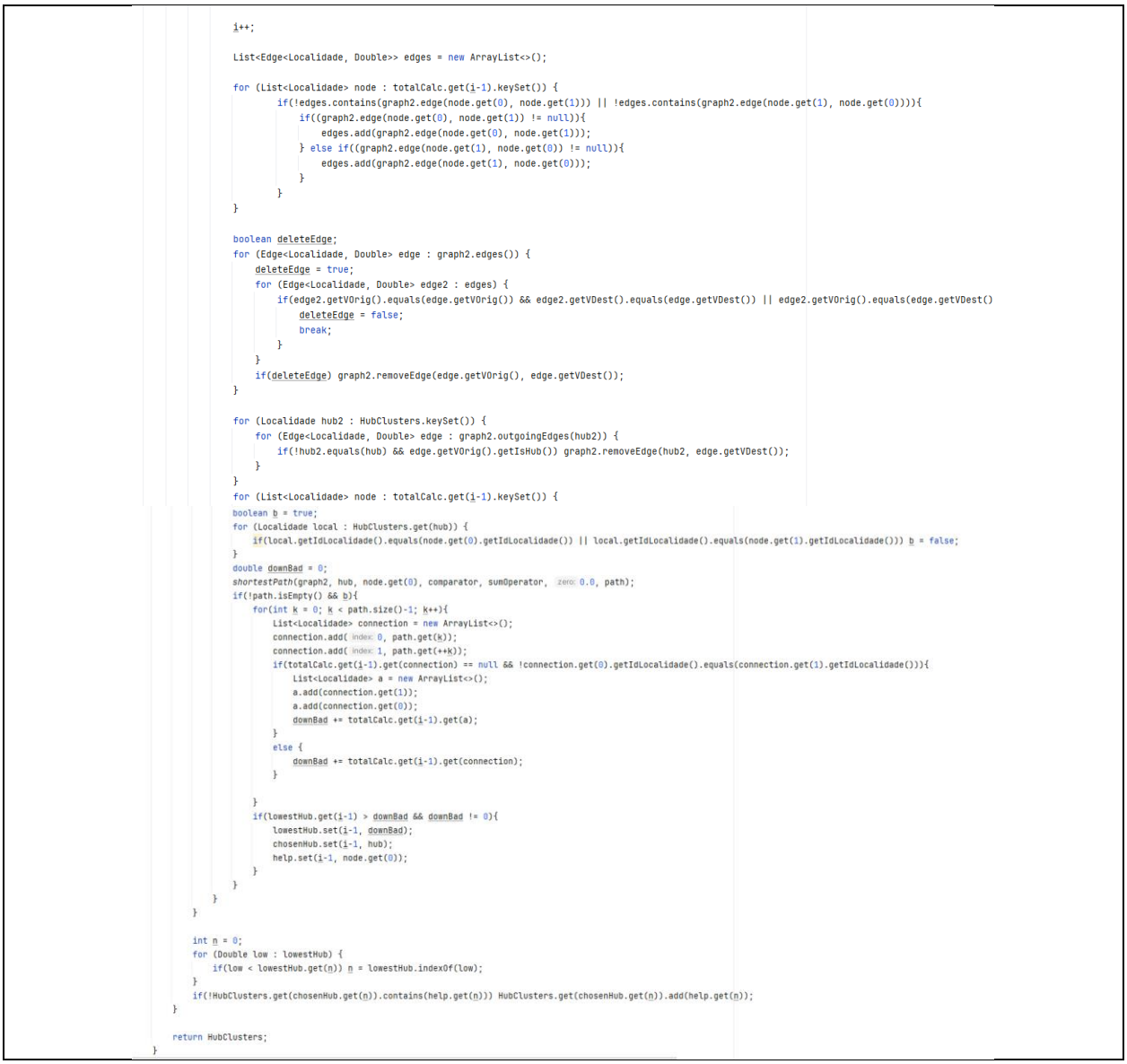
A USBD08 é um caso típico do “Travelling Salesman Problem”, este problema é um problema NP, haviam várias maneiras de encontrar o **circuito** que começasse e acabasse numa dada localidade, ou vértice de um grafo, passando por V hubs uma só vez.

A partir da pesquisa feita foi escolhido o método de Branch and Bound NSS, este método é uma adaptação do método de "Nearest Neighbor Search" adaptando a filosofia de branch and bound para isolar os caminhos com hubs que já foram visitados, visitando consecutivamente os vértices mais próximos do ponto atual, tendo sempre em conta a autonomia do veículo.

Sendo "V" representativo da quantidade de "hubs" e N representativo da quantidade de vértices, temos a seguinte complexidade: $O(T) = O(V! \cdot N^3)$, isto é, de de esperar visto que esta user story é um caso clássico do "Travelling Salesman Problem", um dos mais comuns problemas NP em programação.

USEI09 -

```
public Map<Localidade, List<Localidade>> gervanNewman(MapGraph<Localidade, Double> graph){
    Map<Localidade, List<Localidade>> HubClusters = new HashMap<>();

    MapGraph<Localidade, Double> graph2 = new MapGraph<>( directed: false);
    graph2.copy(graph, graph2);

    for (Edge<Localidade, Double> edge : graph.edges()) {
        if(edge.getVOrig().getIsHub()) HubClusters.put(edge.getVOrig(), new ArrayList<>());
    }

    List<Map<List<Localidade>, Double>> totalCalc = new ArrayList<>(HubClusters.size());

    int i = 0;
    for (Localidade hub : HubClusters.keySet()) {
        graph2.copy(graph, graph2);

        i++;

        for (Localidade hub2 : HubClusters.keySet()) {
            for (Edge<Localidade, Double> edge : graph2.outgoingEdges(hub2)) {
                if(!hub2.equals(hub) && edge.getVOrig().getIsHub()) graph2.removeEdge(hub2, edge.getVDest());
            }
        }

        Map<List<Localidade>, Double> verticeCalc = new HashMap<>();
        for (Localidade vertice : graph2.vertices) {
            verticeCalc = BreadthFirstSearchWithBetweeness(graph2, vertice);
            for (List<Localidade> local : verticeCalc.keySet()) {
                if (totalCalc.size() == 0 || totalCalc.size() < i){
                    totalCalc.add(verticeCalc);
                }else if(!checkEquals(totalCalc.get(i-1), local)){
                    totalCalc.get(i-1).put(local, verticeCalc.get(local));
                }else {
                    if(totalCalc.get(i-1).get(local) == null && !local.get(0).getIdLocalidade().equals(local.get(1).getIdLocalidade())){
                        List<Localidade> a = new ArrayList<>();
                        a.add(local.get(1));
                                a.add(local.get(0));
                                totalCalc.get(i-1).replace(local, totalCalc.get(i-1).get(a) + verticeCalc.get(local));
                            }
                            else totalCalc.get(i-1).replace(local, totalCalc.get(i-1).get(local) + verticeCalc.get(loc
                        }
                    }
                }
            }

            i = 0;
            double lowest = 0;
            Comparator<Double> comparator = Double::compare;
            BinaryOperator<Double> sumOperator = Double::sum;
            LinkedList<Localidade> path = new LinkedList<>();
            List<Double> lowestHub = new ArrayList<>();
            List<Localidade> chosenHub = new ArrayList<>();
            List<Localidade> help = new ArrayList<>();
            for (Localidade hub : HubClusters.keySet()) {
                chosenHub.add(hub);
                lowestHub.add(999999999.9);
                help.add(hub);
            }

            for (int j = 0; j < graph2.numVerts; j++){
                i = 0;
                for (Localidade hub : HubClusters.keySet()) {
                    chosenHub.set(i, hub);
                    lowestHub.set(i, 999999999.9);
                    help.set(i, hub);
                    i++;
                }
                i=0;
                for (Localidade hub : HubClusters.keySet()) {
                    graph2.copy(graph, graph2);
```



```
            i++;

            List<Edge<Localidade, Double>> edges = new ArrayList<>();

            for (List<Localidade> node : totalCalc.get(i-1).keySet()) {
                    if(!edges.contains(graph2.edge(node.get(0), node.get(1))) || !edges.contains(graph2.edge(node.get(1), node.get(0)))){
                        if((graph2.edge(node.get(0), node.get(1)) != null)){
                            edges.add(graph2.edge(node.get(0), node.get(1)));
                        } else if((graph2.edge(node.get(1), node.get(0)) != null)){
                            edges.add(graph2.edge(node.get(1), node.get(0)));
                        }
                    }
            }

            boolean deleteEdge;
            for (Edge<Localidade, Double> edge : graph2.edges()) {
                deleteEdge = true;
                for (Edge<Localidade, Double> edge2 : edges) {
                    if(edge2.getVOrig().equals(edge.getVOrig()) && edge2.getVDest().equals(edge.getVDest()) || edge2.getVOrig().equals(edge.getVDest()
                        deleteEdge = false;
                        break;
                    }
                }
                if(deleteEdge) graph2.removeEdge(edge.getVOrig(), edge.getVDest());
            }

            for (Localidade hub2 : HubClusters.keySet()) {
                for (Edge<Localidade, Double> edge : graph2.outgoingEdges(hub2)) {
                    if(!hub2.equals(hub) && edge.getVOrig().getIsHub()) graph2.removeEdge(hub2, edge.getVDest());
                }
            }
            for (List<Localidade> node : totalCalc.get(i-1).keySet()) {
                boolean b = true;
                for (Localidade local : HubClusters.get(hub)) {
                    if(local.getIdLocalidade().equals(node.get(0).getIdLocalidade()) || local.getIdLocalidade().equals(node.get(1).getIdLocalidade())) b = false;
                }
                double downBad = 0;
                shortestPath(graph2, hub, node.get(0), comparator, sumOperator, zero: 0.0, path);
                if(!path.isEmpty() && b){
                    for(int k = 0; k < path.size()-1; k++){
                        List<Localidade> connection = new ArrayList<>();
                        connection.add( index: 0, path.get(k));
                        connection.add( index: 1, path.get(++k));
                        if(totalCalc.get(i-1).get(connection) == null && !connection.get(0).getIdLocalidade().equals(connection.get(1).getIdLocalidade())){
                            List<Localidade> a = new ArrayList<>();
                            a.add(connection.get(1));
                            a.add(connection.get(0));
                            downBad += totalCalc.get(i-1).get(a);
                        }
                        else {
                            downBad += totalCalc.get(i-1).get(connection);
                        }

                    }
                    if(lowestHub.get(i-1) > downBad && downBad != 0){
                        lowestHub.set(i-1, downBad);
                        chosenHub.set(i-1, hub);
                        help.set(i-1, node.get(0));
                    }
                }
            }
        }

        int n = 0;
        for (Double low : lowestHub) {
            if(low < lowestHub.get(n)) n = lowestHub.indexOf(low);
        }
        if(!HubClusters.get(chosenHub.get(n)).contains(help.get(n))) HubClusters.get(chosenHub.get(n)).add(help.get(n));
    }

    return HubClusters;
}
```

Nesta User Story foi pedido que seja dividida o grafo em N clusters, sendo N a quantidade de Hubs presente no grafo. Cada cluster tem, como obrigação, só 1 hub.

Na execução desta user story foi criada uma função que segue o princípio do algoritmo do Gervan Newman. No algoritmo, é primeiro feito um Breadth First Search de todos os vértices, para atribuir a cada vértice um valor. Através deste valor, é calculado o betweeness de cada edge e é removida a edge que possuí o maior valor. Este processo é repetido até nenhum Hub estar conectado a outro, isolando-os e criando clusters com só um Hub.

A complexidade deste algoritmo é de O(n*(n + e)*m) sendo n a quantidade de localidades no grafo, n+e as localidades + os edges do grafo e m a quantidade de hubs no grafo.