Instituto Superior de **Engenharia** do Porto

Relatório - Projeto Integrador
Sprint III

Estruturas de Informação

Realizado por :

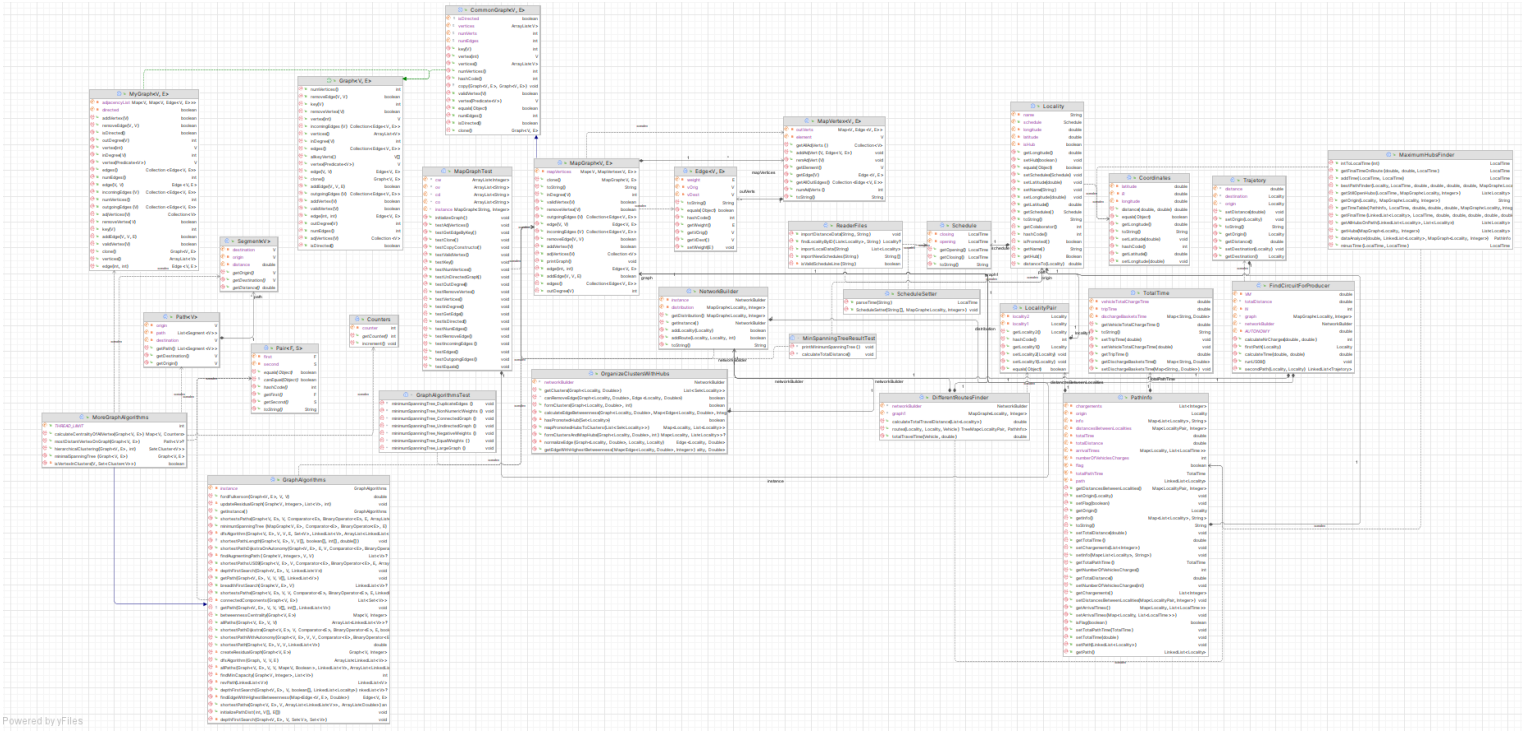
Lourenço Guimarães, 1220766
Diogo Ribeiro, 1220812
Francisco Silveira, 1220813
Rodrigo Brito, 1220842

Turma : 2DH G04 - 2023/2024

O enunciado do trabalho prático é composto por 5 alíneas, que foram distribuídos pelos 4 elementos do grupo da seguinte forma :

Lourenço - exercícios 6;
Diogo - exercícios 7 e 11;
Francisco - exercício 8;
Rodrigo - exercício 9.

Diagrama de classes :



Análise e complexidade dos métodos

User storie 6

Na user storie 6, é pedido que se encontre, para um produtor, os diferentes percursos que consegue fazer entre um local de origem e um hub limitados pelos Kms de autonomia do seu veículo elétrico. Assim, foram implementados 3 métodos na classe DifferentRoutesFinder:

```
4 usages
public double totalTravelTime(Vehicle vehicle, double totalDistance) {
    return totalDistance / vehicle.getAverageSpeed();
}

3 usages
public double calculateTotalTravelDistance(List<Locality> routes) {
    double totalDistance = 0;
    for (int i = 0; i < routes.size() - 1; i++) {
        Locality origin = routes.get(i);
        Locality destiny = routes.get(i + 1);

        totalDistance += graph1.edge(origin, destiny).getWeight();
    }
    return totalDistance;
}
```

totalTravelTime é um método que através da velocidade média de um veículo e da distância total viajada, calcula o tempo total de viagem.

calculateTotalTravelDistance é utilizado para calcular a distância total percorrida, a partir de uma lista de Hubs. Ao iterar sobre a lista, o método obtém 2 hubs e adiciona a sua distância à distância total percorrida.

```
no usages
public TreeMap<LocalityPair, PathInfo> routes(Locality origin, Locality hub, Vehicle vehicle) {
    TreeMap<LocalityPair, PathInfo> result = new TreeMap<>();

    List<LinkedList<Locality>> allPaths = dfsAlgorithm(graph1, origin, hub, vehicle.getAutonomy());

    for (LinkedList<Locality> path : allPaths) {
        double totalDistance = calculateTotalTravelDistance(path);
        double totalTravelTime = totalTravelTime(vehicle, totalDistance);

        // Use LocalityPair as the key for the TreeMap
        LocalityPair localityPair = new LocalityPair(path.getFirst(), path.getLast());

        result.put(localityPair, new PathInfo(totalDistance, totalTravelTime));
    }

    return result;
}
```

O método principal é o routes. Inicialmente, é criado um TreeMap para armazenar os resultados em que a key é um LocalityPair (um par de hubs), e o value é um objeto do tipo PathInfo, que armazena a distância percorrida e o tempo total de viagem. Depois, utilizando o algoritmo de procura DFS, é criada uma lista de LinkedLists de Hubs, que será, na prática, a lista com os diferentes possíveis caminhos. De seguida, o método itera sobre a lista e, para cada LinkedList, obtém a distância total percorrida e o tempo total de viagem através dos respetivos métodos, cria um objeto do tipo LocalityPair com a origem e destino dessa viagem. Depois disso, adiciona o LocalityPair e o PathInfo (com distância e tempo) ao mapa de resultados e no final, retorna este mapa.

A complexidade do método routes(), será aproximadamente, O(V + E + P*N), em que V é o número de vértices, E é o número de arestas, P é o número de caminhos encontrados e N é o número de localidades num caminho.

User storie 7

Na user storie 7 é pedido que seja encontrado o percurso de entrega que maximize o número de hubs pelo qual passa, partindo de um local origem e tendo em conta o horário de funcionamentos de cada hub, o tempo de descarga dos cestos, as distâncias a percorrer, a velocidade e o tempo de carregamento do veículo.

Para esse efeito, é usado o algoritmo **shortestPathWithAutonomy**, cuja complexidade é **O((V+E)*log(V))**, sendo V o número de vértices e E o número de arestas, a parte log(v) advém do uso da estrutura de dados para manter e acessar aos vértices ainda não visitados com distâncias mínimas.

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

    shortPath.clear();
    int numVerts = g.numVertices();
    boolean[] visited = new boolean[numVerts];
    V[] pathKeys = (V[]) new Object[numVerts];
    E[] dist = (E[]) new Object[numVerts];
    initializePathDist(numVerts, pathKeys, dist);

    shortestPathDijkstraOnAutonomy(g, autonomy, vOrig, ce, sum, zero, visited, pathKeys, dist);

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

    if (lengthPath != null) {
        getPath(g, vOrig, vDest, pathKeys, shortPath);
        return lengthPath;
    }

    return null;
```

O método dataAnalyze() tem como intuito analisar um caminho representado por uma linkedList de localidades no grafo , tendo em consideração a autonomia do veículo. O método retorna um objeto do tipo PathInfo que contém informações sobre o caminho analisado. A complexidade é de O(N), visto que no método se percorrem (no ciclo for each) a LinkedList, cujo N representa o número de elementos dentro da mesma.

```java
public static PathInfo dataAnalyze (double autonomy, LinkedList<Locality> path, MapGraph<Locality, Integer> graph){
    ArrayList<Integer> chargementsIndexes = new ArrayList<>();
    LinkedList<Locality> path1 = path;
    boolean flag = true;
    double distance = 0, battery = autonomy;
    if(path1 != null){
        for (int i = 0; i < path1.size(); i++) {
            double distanceBetweenHubs = graph.edge(path1.get(i), path1.get(i + 1)).getWeight();
            distance += distanceBetweenHubs;
            if(distanceBetweenHubs / 1000 > battery){
                if(distanceBetweenHubs / 1000 <= autonomy){
                    chargementsIndexes.add(i);
                    battery = autonomy;
                }else {
                    flag = false;
                }
            }else{
                battery -= distanceBetweenHubs / 1000;
            }
        }
    }else{
        flag = false;
        return new PathInfo(distance, path1, chargementsIndexes, flag);
    }
    return new PathInfo(distance, path1, chargementsIndexes, flag);
```

Os métodos acima explicitados são depois utilizados no método principal da classe MaximumHubsFinder, bestPathFinder(), cujo principal intuito é encontrar o melhor caminho, considerando um local de partida, o horário de início do percurso, a autonomia do veículo, a velocidade média, o tempo de carregamento do veículo, o tempo de descarga nos hubs e o grafo. Como o método chama o algoritmo shortestPathWithAutonomy, este herda a sua complexidade, logo tem complexidade O((V+E)*log(V)), a mesma que a do algoritmo.

```
public static PathInfo bestPathFinder(Locality origin, LocalTime startTime, double autonomy, double averageSpeed,
                                      double chargeTime, double unloadingTime, MapGraph<Locality, Integer> graph) {
    int numberOfOpenHubs = 0;
    Locality locality = origin;
    LocalTime hour = startTime, remainingTime = LocalTime.of( hour: 0,  minute: 0), inicialHour = startTime;
    List<Locality> visitedLocalities = new ArrayList<>();
    LinkedList<Locality> actualPath = new LinkedList<>(), bestPath = new LinkedList<>();

    while (remainingTime != null) {
        remainingTime = null;
        for (Locality hub : getHubs(graph)) {
            if (!visitedLocalities.contains(hub)) {
                LinkedList<Locality> donePath = new LinkedList<>();
                //GraphAlgorithms.shortestPathWithAutonomy(graph, origin, hub, Comparator.naturalOrder(), Integer::sum, 0, donePath, autonomy);
                if (getFinalTime(donePath, inicialHour, autonomy, averageSpeed, chargeTime, unloadingTime,
                        getAllHubsOnPath(donePath, visitedLocalities).size(), graph).isBefore(hub.getSchedules().getClosing()) &&
                        getFinalTime(donePath, inicialHour, autonomy, averageSpeed, chargeTime, unloadingTime,
                                getAllHubsOnPath(donePath, visitedLocalities).size(), graph).isAfter(hub.getSchedules().getOpening())) {
                    if (remainingTime == null ||
                            (getStillOpenHubs(getFinalTime(donePath, inicialHour, autonomy, averageSpeed, chargeTime, unloadingTime, getAllHubsOnPath(donePath, visitedL
                                    .size() > numberOfOpenHubs
                                    && minusTime(
                                    hub.getSchedules().getClosing(), getFinalTime(donePath, startTime, autonomy, averageSpeed, chargeTime, unloadingTime,
                                            getAllHubsOnPath(donePath, visitedLocalities).size(), graph)).isBefore(remainingTime))) {
                        actualPath = donePath;
                        remainingTime = minusTime(hub.getSchedules().getClosing(), getFinalTime(donePath, startTime, autonomy, averageSpeed, chargeTime, unloadingTime,
                                getAllHubsOnPath(donePath, visitedLocalities).size(), graph));
                        hour = getFinalTime(donePath, startTime, autonomy, averageSpeed, chargeTime, unloadingTime, getAllHubsOnPath(donePath, visitedLocalities).size()
                        numberOfOpenHubs = getStillOpenHubs(hour, graph).size();
                    }
```

```
if (remainingTime != null) {
    if (bestPath.isEmpty()) {
        bestPath.addAll(actualPath);
    } else if (!actualPath.isEmpty()) {
        bestPath.removeLast();
        bestPath.addAll(actualPath);
    }

    if (!bestPath.isEmpty()) {
        locality = bestPath.getLast();
        for (int i = 0; i < getAllHubsOnPath(actualPath, visitedLocalities).size(); i++) {
            visitedLocalities.add(getAllHubsOnPath(actualPath, visitedLocalities).get(i));
        }
    } else {
        break;
    }

    for (int i = 0; i < getAllHubsOnPath(actualPath, visitedLocalities).size(); i++) {
        visitedLocalities.add(getAllHubsOnPath(actualPath, visitedLocalities).get(i));
    }
    startTime = hour;
    remainingTime = null;
}

PathInfo pathInfo = dataAnalyze(autonomy, bestPath, graph);
pathInfo.setArrivalTimes(getTimeTable(pathInfo, startTime, averageSpeed, chargeTime, unloadingTime, graph));
return pathInfo;
```

User storie 8

Na user storie 8 é pedido que se encontre um produtor o circuito de entrega que parte de um local origem, passa por N hubs com maior número de colaboradores uma só vez e volta ao local origem minimizando a distância total percorrida.

Para esse efeito, implementam-se 4 métodos, um para o caminho de volta, outro para o caminho de volta, um para calcular a distância total e outro para calcular o número de carregamentos.

O método **firstPath** encontra um caminho num grafo direcionado. Começa numa determinada localidade (currentLocation). O objetivo é percorrer N vértices ao escolher a próxima localidade a visitar com base no maior número de colaboradores. A complexidade do algoritmo depende da estrutura e do tamanho do grafo. Se o grafo tiver V vértices e E arestas, a complexidade seria aproximadamente O(N * (V + E)), pois, em cada iteração, é necessário percorrer as arestas de saída da localidade atual.

```
public Locality firstPath(Locality currentLocation) {
    System.out.println("First Path: ");
    List<Edge<Locality, Integer>> newOrigin;

    int maxCollab;
    int counter = 0;

    Locality maxLocal = null;
    List<Locality> alreadyPassed = new ArrayList<>();

    while (counter < N) {
        newOrigin = (List<Edge<Locality, Integer>>) graph.outgoingEdges(currentLocation);
        maxCollab = 0;

        for (Edge<Locality, Integer> edge : newOrigin) {
            int collaborator = edge.getVDest().getColaborator();

            if (collaborator > maxCollab) {
                if (!alreadyPassed.contains(edge.getVDest())) {
                    maxCollab = collaborator;
                    maxLocal = edge.getVDest();
                }
            }
        }
```

```
            System.out.println(currentLocation.getName() + " -> " + maxLocal.getName());
            System.out.println(" Distance: " + graph.edge(maxLocal, currentLocation).getWeight());
            totalDistance = totalDistance + graph.edge(maxLocal, currentLocation).getWeight();
            currentLocation = maxLocal;
            counter++;
            alreadyPassed.add(currentLocation);
        }
        System.out.println("-------------------------------------");
        return maxLocal;
    }
```

O método **secondPath** calcula o caminho mais curto entre duas localidades num grafo usando o algoritmo de caminho mais curto

```
public LinkedList<Trajetory> secondPath(Locality start, Locality end) {
    System.out.println("Second Path: ");
    LinkedList<Locality> shortPath = new LinkedList<>();
    LinkedList<Trajetory> trajetory = new LinkedList<>();

    GraphAlgorithms.shortestPath(graph, start, end, shortPath);
    int counter = 0;
    for (Locality local : shortPath) {
        if (counter == 0) {
            counter++;
            continue;
        }

        Double distanceBeetween = Double.valueOf(graph.edge(start, local).getWeight());
        totalDistance = totalDistance + graph.edge(start, local).getWeight();


        trajetory.add(new Trajetory(start, local, distanceBeetween));
        start = local;


    }
    for (int i = 0; i < trajetory.size(); i++) {
        System.out.println(trajetory.get(i).toString());
    }
    System.out.println("-------------------------------------");
    return trajetory;
}
```

A complexidade deste método depende da implementação específica do algoritmo de caminho mais curto (GraphAlgorithms.shortestPath). A complexidade seria O(V * E), onde V é o número de vértices e E é o número de arestas no grafo.

```java
public double calculateTime(double totalDistance, double VM) {
    double totalDistanceInKm;
    totalDistanceInKm = totalDistance / 1000.00;
    return (totalDistanceInKm / VM);
}
```

A função do método **calculateTime** é converter a distância total para quilômetros e, em seguida, calcula o tempo de viagem com base na velocidade média do veículo. A unidade de medida do resultado final será em horas.

```java
public int calculateNrCharges(double totalDistance, double AUTONOMY) {
    int nrCharges = (int) (totalDistance / AUTONOMY);
    double nrChargesDouble = totalDistance / AUTONOMY;

    if (nrChargesDouble > nrCharges) {
        nrCharges++;
    }

    return nrCharges;
}
```

O método **calculateNrCharges** recebe a distância total a percorrer (totalDistance) e a autonomia do veículo (AUTONOMY). Com base nesses parâmetros, o método calcula o número de cargas necessárias para completar a viagem, levando em consideração a autonomia do veículo. Ele realiza a divisão da distância total pela autonomia para obter um valor inicial, e em seguida, verifica se há uma fração não utilizada que exige uma carga adicional. Se necessário, o número de cargas é arredondado para cima. O resultado final é o número total de cargas necessárias para percorrer a distância total com o veículo.

User storie 9

Na user story 9, é pedido que se implementem um algoritmo para organizar as localidades de um grafo em N clusters, garantindo a presença de apenas um hub

por cluster. O processo de formação dos clusters é iterativo e baseia-se na remoção de arestas que possuem o maior número de caminhos mais curtos entre localidades, até que cada cluster fique isolado. O objetivo é fornecer uma solução que apresente a lista de hubs e o conjunto de localidades associadas a cada cluster, respeitando a condição de ter um hub por cluster. Este algoritmo visa otimizar a organização das localidades de forma eficiente, destacando hubs significativos para melhorar a conectividade e a eficiência da rede de localidades. O critério de aceitação inclui a entrega de uma lista de hubs e seus respectivos conjuntos de localidades para cada cluster formado.

```
public class OrganizeClustersWithHubs {
    1 usage
    NetworkBuilder networkBuilder = NetworkBuilder.getInstance();
    Rodrigo Brito
    public MapGraph<Locality, Integer> getGraph() {
        return networkBuilder.getDistribution();
    }

    2 usages   Rodrigo Brito *
    public static boolean formClusters(Graph<Locality, Double> g, int desiredNumClusters) {

        Map<Edge<Locality, Double>, Integer> edgeBetweenness = new HashMap<>();
        calculateEdgeBetweenness(g, edgeBetweenness);
        if(HubDefiner.numHubs((MapGraph<Locality, Double>) g)<desiredNumClusters){
            return false;
        }
        while (getClusters(g).size() < desiredNumClusters) {
            Edge<Locality, Double> edgeToRemove = getEdgeWithHighestBetweenness(edgeBetweenness);
            if (edgeToRemove != null) {
                edgeBetweenness.remove(edgeToRemove);
                if (canRemoveEdge(g, edgeToRemove)) {
                    g.removeEdge(edgeToRemove.getVOrig(), edgeToRemove.getVDest());
                }
            } else {
                break;
            }
        }
        return true;}
```

Este excerto de código pertence a uma classe que organiza clusters em um grafo, garantindo a presença de hubs. O código faz o seguinte:
Obtém o grafo e calcula a centralidade de betweenness para suas arestas.
Verifica se há hubs suficientes para formar o número desejado de clusters.
Em um loop, remove iterativamente as arestas com maior centralidade de betweenness até atingir o número desejado de clusters.
A complexidade depende das implementações específicas de funções como o cálculo de betweenness e a verificação de remoção de arestas.

```java
public static List<Set<Locality>> getClusters(Graph<Locality, Double> g) {
    boolean[] visited = new boolean[g.numVertices()];
    List<Set<Locality>> clusters = new ArrayList<>();

    for (Locality vert : g.vertices()) {
        if (!visited[g.key(vert)]) {
            LinkedList<Locality> clusterVertices = new LinkedList<>();
            GraphAlgorithms.depthFirstSearch(g, vert, visited, clusterVertices);

            Set<Locality> cluster = new HashSet<>(clusterVertices);
            clusters.add(cluster);
        }
    }
    return clusters;
}
```

1 usage Rodrigo Brito +1

```java
static boolean canRemoveEdge(Graph<Locality, Double> g, Edge<Locality, Double> edge) {
    Graph<Locality, Double> clone = g.clone();
    clone.removeEdge(edge.getVOrig(), edge.getVDest());
    List<Set<Locality>> clusters = getClusters(clone);
    for (Set<Locality> cluster : clusters) {
        if (!hasPromotedHub(cluster)) {
            return false;
        }
    }
    return true;
}
```

Este excerto de código consiste em duas funções:
-getClusters: Utiliza DFS para identificar clusters de localidades não visitadas. Retorna uma lista de conjuntos (clusters) de localidades.
-canRemoveEdge: Verifica se é possível remover uma aresta específica sem violar a condição de ter um hub por cluster. Cria um clone do grafo, obtém clusters usando getClusters, e verifica se cada cluster possui pelo menos um hub promovido.

Complexidade:
-getClusters: O(V + E), onde V é o número de vértices e E é o número de arestas.
-canRemoveEdge: Depende da implementação específica de hasPromotedHub e da complexidade de getClusters.

```java
private static boolean hasPromotedHub(Set<Locality> cluster) {
    for (Locality locality : cluster) {
        if (locality.isPromoted()) {
            return true;
        }
    }
    return false;
}
```

1 usage Rodrigo Brito *

```java
public static void calculateEdgeBetweenness(Graph<Locality, Double> g, Map<Edge<Locality, Double>, Integer> edgeBetweenness) {
    Comparator<Double> ce = Comparator.naturalOrder();
    BinaryOperator<Double> sum = Double::sum;
    List<Locality> vertices = g.vertices();
    int numVertices = vertices.size();

    for (Edge<Locality, Double> edge : g.edges()) {
        Edge<Locality, Double> reverseEdge = g.edge(edge.getVDest(), edge.getVOrig());
        if (reverseEdge != null && !edgeBetweenness.containsKey(reverseEdge)) {
            edgeBetweenness.putIfAbsent(edge, 0);
        }
    }
```

```
for (int i = 0; i < numVertices; i++) {
    Locality source = vertices.get(i);
    ArrayList<LinkedList<Locality>> allPaths = new ArrayList<>();
    ArrayList<Double> allDists = new ArrayList<>();
    GraphAlgorithms.shortestPathsUS09(g, source, ce, sum, zero: 0.0, allPaths, allDists);
    for (int j = i + 1; j < numVertices; j++) {
        LinkedList<Locality> path = allPaths.get(j);
        if (path != null && path.size() > 1) {
            Iterator<Locality> it = path.iterator();
            Locality v1 = it.next();

            while (it.hasNext()) {
                Locality v2 = it.next();
                Edge<Locality, Double> edge = g.edge(v1, v2);
                Edge<Locality, Double> reverseEdge = g.edge(v2, v1);

                if (edge != null) {
                    if (edgeBetweenness.containsKey(reverseEdge)) {
                        edgeBetweenness.put(reverseEdge, edgeBetweenness.get(reverseEdge) + 1);
                    } else {
                        edgeBetweenness.put(edge, edgeBetweenness.getOrDefault(edge, defaultValue: 0) + 1);
                    }
                }
                v1 = v2;
            }
        }
    }
}
```

Eis mais dois metodos da classe OrganizeClustersWithHubs:
-hasPromotedHub: Verifica se há pelo menos uma localidade promovida em um cluster.
-calculateEdgeBetweenness: Calcula a centralidade de betweenness para cada aresta no grafo, considerando caminhos mais curtos entre localidades.
Atualiza um mapa com a contagem de caminhos mais curtos para cada aresta.

Complexidade:
-hasPromotedHub: O(N), onde N é o número de localidades no cluster.
-calculateEdgeBetweenness: O(V + E), onde V é o número de vértices e E é o número de arestas, com dependência na complexidade da função GraphAlgorithms.shortestPathsUS09.

```java
no usages  Rodrigo Brito
private static Edge<Locality, Double> normalizeEdge(Graph<Locality, Double> g, Locality v1, Locality v2) {
    Edge<Locality, Double> directEdge = g.edge(v1, v2);
    Edge<Locality, Double> reverseEdge = g.edge(v2, v1);
    return (directEdge != null) ? directEdge : reverseEdge;
}
1 usage  Rodrigo Brito *
public static Edge<Locality, Double> getEdgeWithHighestBetweenness(Map<Edge<Locality, Double>, Integer> edgeBetweenness) {
    Edge<Locality, Double> maxEdge = null;
    int maxBetweenness = -1;
    String minEdgeName = null;

    for (Map.Entry<Edge<Locality, Double>, Integer> entry : edgeBetweenness.entrySet()) {
        Edge<Locality, Double> edge = entry.getKey();
        int betweenness = entry.getValue();
        String v1Name = edge.getVOrig().toString();
        String v2Name = edge.getVDest().toString();
        String edgeName = v1Name.compareTo(v2Name) < 0 ? v1Name + v2Name : v2Name + v1Name;
        if (betweenness > maxBetweenness || (betweenness == maxBetweenness && (minEdgeName == null || edgeName.compareTo(minEdgeName) < 0))) {
            maxEdge = edge;
            maxBetweenness = betweenness;
            minEdgeName = edgeName;
        }
    }
    return maxEdge;
}
```

Estes dois métodos sao responsáveis por:
- normalizeEdge: Retorna a aresta entre dois vértices em um grafo, priorizando a direção direta se existir, ou a reversa caso contrário.
-getEdgeWithHighestBetweenness: Encontra a aresta com a maior centralidade de betweenness em um mapa, resolvendo empates com base na ordem lexicográfica dos nomes dos vértices.

```java
1 usage  Rodrigo Brito
public static Map<Locality, List<Locality>> mapPromotedHubsToClusters(List<Set<Locality>> clusters) {
    Map<Locality, List<Locality>> promotedHubsToClusters = new HashMap<>();
    for (Set<Locality> cluster : clusters) {
        for (Locality locality : cluster) {
            if (locality.isPromoted()) {
                promotedHubsToClusters.put(locality, new ArrayList<>(cluster));
                break;
            }
        }
    }
    return promotedHubsToClusters;
}
```

```java
2 usages  Rodrigo Brito
public static Map<Locality, List<Locality>> formClustersAndMapHubs(Graph<Locality, Double> g, int desiredNumClusters) {
    boolean clustersFormed = formClusters(g, desiredNumClusters);
    if (!clustersFormed) {
        return null;
    }
    List<Set<Locality>> clusters = getClusters(g);
    return mapPromotedHubsToClusters(clusters);
}
```

Por último temos ainda os metodos mapPromotedHubsToClusters e formClustersAndMapHubs. O método mapPromotedHubsToClusters cria um mapa associando hubs promovidos aos seus clusters correspondentes, a partir de uma

lista de clusters. Tem uma complexidade de O(N * M), onde N é o número total de localidades nos clusters e M é o tamanho médio de um cluster.
Já o método formClustersAndMapHubs organiza um grafo em clusters desejados e associa hubs promovidos a esses clusters.
Complexidade: Depende da complexidade de formClusters e mapPromotedHubsToClusters, ambas O(V + E), onde V é o número de vértices e E é o número de arestas no grafo.

User storie 11

Na user storie 11 é pedido que se implementem métodos que permitam o carregamento de um ficheiro com horários de funcionamento de uma lista de hubs, sendo que caso os hubs presentes no ficheiro já existam, os horários devem ser redefinidos, caso contrário deve ser emitida uma mensagem de erro.

Para esse efeito, implementaram-se dois métodos na classe ReaderFiles do projeto, são eles:

```
6 usages  Diogo_1220812
public static String[] importNewSchedules(String fileName) throws IOException {
    List<String> dataList = new ArrayList<>();
    try (BufferedReader reader = new BufferedReader(new FileReader(fileName))) {
        String currentLine;
        while ((currentLine = reader.readLine()) != null) {
            if(isValidScheduleLine(currentLine)) {
                dataList.add(currentLine);
            }else{
                throw new IOException("Invalid schedule line: " + currentLine);
            }
        }
        return dataList.toArray(new String[0]);
    }
}

/**
 * Verifies if a given line extracted from the file is in the given format
 * @param line the content on the given line
 * @return true if the array with the line content has the pretended length or false, othewise.
 */
1 usage  Diogo_1220812
private static boolean isValidScheduleLine(String line) {
    String[] parts = line.split( regex: ",");
    return parts.length == 3;
}
```

O método importNewSchedules é responsável pela extração dos dados provenientes do ficheiro, retornando um array de Strings com

todos esses dados, em cada linha (id do hub, horário de abertura, horário de encerramento).

O método isValidSchedulesLines serve para verificação de cada linha extraída, por forma a confirmar que o ficheiro a carregar no sistema apresenta o formato correto.

O método abaixo, recebe por parâmetros os dados extraídos do ficheiro e o grafo preenchido, após isto dá-se a iteração do grafo e a verificação de que há ou não hubs definidos no ficheiro importado que façam parte do grafo, caso hajam, o horário a eles afeto será mudado de acordo com os dados extraídos e uma mensagem de sucesso é imprimida, caso contrário é emitida uma mensagem de erro.

Em termos de complexidade, o método apresenta O(|V| * |E|), sendo |V| o número de vértices do grafo e |E| o número de arestas do mesmo. Esta surge por contribuição:

- do ciclo foreach que percorre todas as linhas de entrada do array passado por parâmetro - O(N), sendo N o nº de linhas percorridas;
- do ciclo foreach que percorre todos os vértices do grafo, o que contribuí para uma complexidade O(V), sendo V o nº de vértices no grafo.

Ora, a complexidade total é portanto dada por O(N*V), que pode ser simplificada para O(|V| *|E|), visto que o número de arestas é proporcional ao número de vértices.

```java
public static void ScheduleSetter(String[] data, MapGraph<Locality, Integer> graph) throws IOException {
    for (String line : data) {
        String[] parts = line.split( regex: ",");
        if (parts.length == 3) {
            String id = parts[0].trim();
            String openingHour = parts[1].trim();
            String closingHour = parts[2].trim();

            boolean localityFound = false;

            for (Locality locality : graph.vertices()) {
                if (locality.getName().equals(id)) {
                    LocalTime openingTime = parseTime(openingHour);
                    LocalTime closingTime = parseTime(closingHour);

                    Schedule schedule = new Schedule(openingTime, closingTime);
                    locality.setSchedules(schedule);
                    localityFound = true;
                    System.out.println("The new schedule was successfully set for the locality: " + id);
                }
            }
            if (!localityFound) {
                System.out.println("The locality with ID: " + id + " isn't a hub");
            }
        }
    }
}
```