2023

RELATÓRIO DE ESINF - TRABALHO SPRINT 3

José Sá, 1220612

Mariana Correia, 1211883

Rafael Araújo, 1201804

João Pinto, 1221694

Vasco Sousa, 1221700

Índice

Objetivo

Para este trabalho, é pedido que se desenvolva de forma mais adequada um conjunto de funcionalidades que permitam através da interface Graph gerir uma rede de distribuição de cabazes de produtos agrícolas.

A rede deve ser constituída por vários vértices que representam localidades onde poderão existir Hubs de distribuição e cada localidade está a X distancia de qualquer outra localidade. Os cabazes são transportados em veículos elétricos, com autonomia limitada (em km) e a distribuição dos produtos é condicionada ao horário de funcionamento do hub.

Algoritmo

Estrutura de Dados Inicial

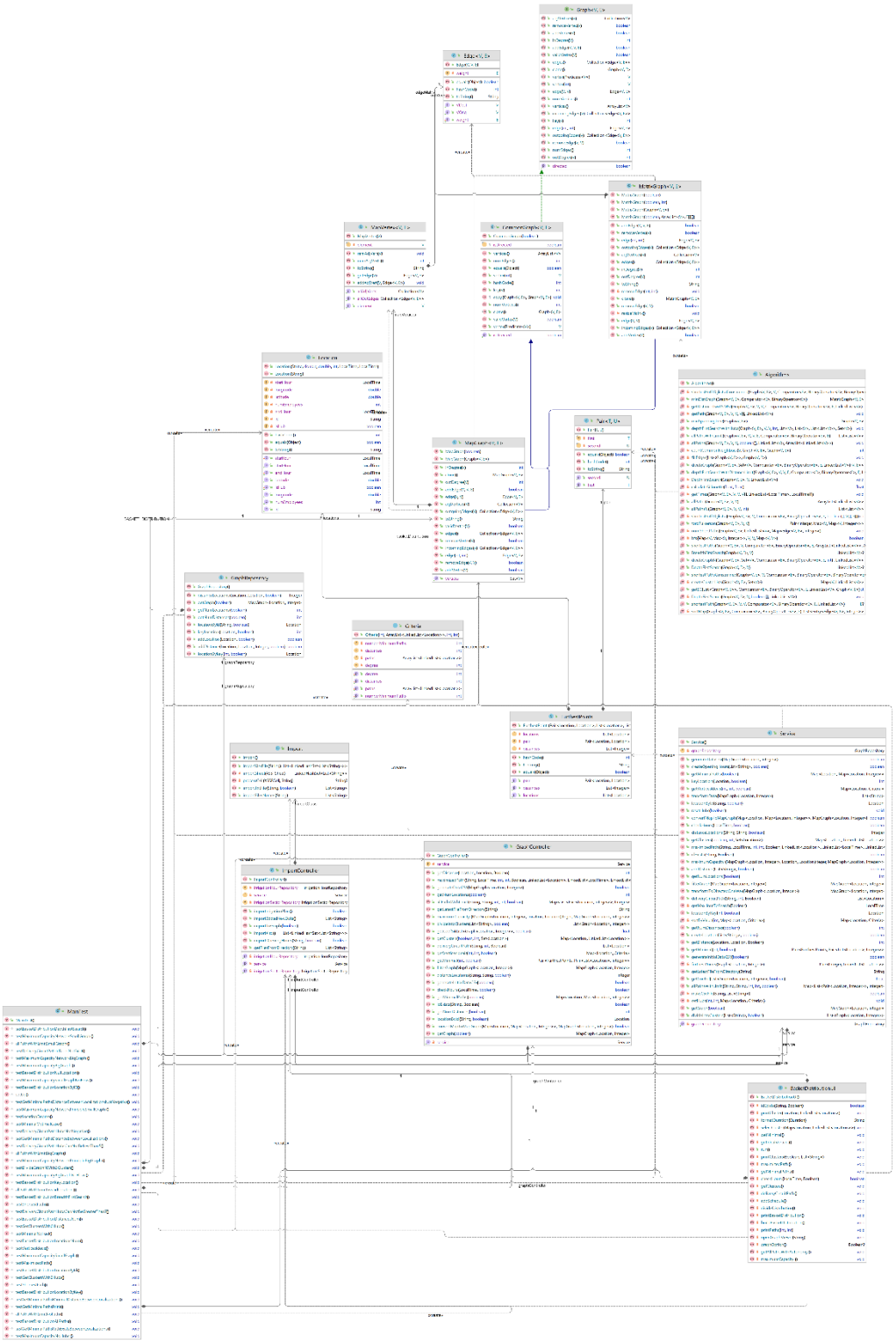
De maneira a inicializarmos a implementação dos Grafos e respetivos algoritmos, primeiramente foi necessário extrair os dados existentes nos ficheiros fornecidos. Tal como nos foi pedido, para executar os testes deve-se utilizar os ficheiros grandes, sendo assim optamos por extrair apenas esses dados. Após executada a extração, os dados ficam armazenados em duas Lists, uma de localizações e uma de distâncias, tal como é possível verificar nas imagens que se seguem:

```
public List<String> importTxtFile(String filePath, boolean skip) {

    InputStream stream = getClass().getClassLoader().getResourceAsStream(filePath);
    List<String> output = new ArrayList<>();
    Scanner sc = new Scanner(stream);

    if (skip)
        sc.nextLine();
    while (sc.hasNext()) {
        output.add(sc.nextLine());
    }
    sc.close();

    return output;
}
```

```
List<String> locations = importClass.importTxtFile( filePath: "esinf/locais_big.csv",  skip: true);
List<String> distances = importClass.importTxtFile( filePath: "esinf/distancias_big.csv",  skip: true);
```

Diagrama de Classes

Consoante o que fomos desenvolvendo e que iremos explicar mais à frente, o diagrama de classes final é o que se apresenta na seguinte imagem (devido ao tamanho do diagrama, a imagem fica muito pequena e com pouca qualidade, por isso incluiremos a mesma no README do projeto):



US EI06

Problema: Encontrar 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, ou seja, não considerando carregamentos no percurso.

Resolução: Para a realização desta Us necessitámos de pesquisar no grafo criado anteriormente, todos os caminhos possível dado um local de partida e um local de destino e uma dada autonomia.

```
public static <V, E> List<List<V>> allPathsWithLimit(Graph<V, E> g, V vOrig, V vDest, E distanceLimit,
        Comparator<E> ce, BinaryOperator<E> sum, E zero) {
    List<List<V>> paths = new ArrayList<>();

    depthFirstSearchWithDistanceLimit(g, vOrig, vDest, distanceLimit, ce, sum, zero,
            new ArrayList<>(Collections.singletonList(vOrig)), zero,
            paths, new HashSet<>());
    return paths;
}
```

Para tal usamos um método recursivo de depthFirstSearch, adaptado para que quando a distância do percurso em questão seja maior que a autonomia do veículo esse caminho não seja considerado, evitando assim o peso de passar por toda a profundidade de um grafo.

```
public static <V, E> void depthFirstSearchWithDistanceLimit(Graph<V, E> g, V v, V vDest, E distanceLimit,
        Comparator<E> ce,
        BinaryOperator<E> sum, E weight,
        List<V> path, E pathWeight, List<List<V>> paths, Set<V> visited) {
    if (ce.compare(pathWeight, distanceLimit) > 0) {
        return;
    }

    if (v.equals(vDest)) {
        paths.add(new ArrayList<>(path));
    }

    visited.add(v); // Mark vertex as visited

    for (V vAdj : g.adjVertices(v)) {
        if (!visited.contains(vAdj)) { // Skip if vertex has been visited
            E edgeWeight = g.edge(v, vAdj).getWeight();
            E newPathWeight = sum.apply(pathWeight, edgeWeight);
            path.add(vAdj);
            depthFirstSearchWithDistanceLimit(g, vAdj, vDest, distanceLimit, ce, sum, weight, path, newPathWeight,
                    paths, new HashSet<>(visited)); // Pass a copy of visited set
            path.remove(path.size() - 1);
        }
    }
}
```

Neste algoritmo, podemos notar que o caminho vai sendo criado, tendo em consideração as localidades já visitadas e o limite de distância que o veículo consegue percorrer.

```
List<List<Location>> paths = Algorithms.allPathsWithLimit(graph,
        graphRepository.locationById(vOrigin, bigGraph),
        graphRepository.locationById(vDest, bigGraph), autonomy, Integer::compare,
        Integer::sum, zero:0);
Map<List<Pair<Location, Integer>>, Integer> output = new HashMap<>();

paths.forEach(path -> {
    List<Pair<Location, Integer>> pathWithDistance = new ArrayList<>();
    int pathDistance = 0;
    for (int i = 0; i < path.size() - 1; i++) {
        Location loc1 = path.get(i);
        Location loc2 = path.get(i + 1);
        Integer distance = graph.edge(loc1, loc2).getWeight();
        pathDistance += distance;
        pathWithDistance.add(new Pair<>(loc1, distance));
    }
    pathWithDistance.add(new Pair<>(path.getLast(), second:0));
    output.put(pathWithDistance, pathDistance);
```

Por fim, com todos os caminhos já determinados é calculada as distâncias entre todas as localidades pertencentes ao caminho e também a distância total do mesmo, identificando, ainda, caso uma das localidades seja um Hub.

Complexidade: Visto que este algoritmo se baseia numa pesquisa DepthFirstSearch, cuja complexidade é de O(V+E), e sabendo ainda que temos em consideração os vértices visitados a complexidade passa para O(V*(V+E)). No fim da execução do algoritmo principal ainda é necessário calcular as distâncias de todas as localidades tendo de passar por todos os vértices do output, logo:

$$O(V\text{^}2(V+E))$$

US EI07

Problema: Encontrar para um produtor que parte de um local de origem o percurso de entrega que maximiza o número de hubs pelo qual passa, tendo em consideração o horário de funcionamento de cada hub, o tempo de descarga dos cestos em cada hub, as distâncias a percorrer, a velocidade média do veículo e os tempos de carregamento do veículo.

Resolução: Para esta US começamos por pedir ao utilizador a autonomia do veículo, a velocidade média, a localização de origem e a hora de partida. Após serem dados estes valores, os mesmo irão ser utilizados como parâmetros do método "shortestPathDijkstraConstrained". Este método irá iterar por todos os vértices até todos serem visitados e durante estas iterações, o método irá calcular os diferentes tempos, entre viagem, carregamentos e descargas, no caso destas duas é preciso que a autonomia durante a viagem seja menor do que o peso associado ao vértice e que este seja um hub para ambas acontecerem, no entanto uma não depende da outra. Por fim, este método determina os melhores caminhos para cada vértice tendo como critério os tempos de chegada a cada ponto.

```
private static <V, E> void shortestPathDijkstraConstrained(Graph<V, E> g, V vOrig,
        Comparator<E> ce, BinaryOperator<E> sum, BinaryOperator<E> subtract, E zero,
        boolean[] visited, V[] pathKeys, LocalTime[] arriveTimes, LocalTime[] departTimes,
        LocalTime[] afterChargeTimes, LocalTime[] descargaTimes, E[] dist, E autonomy,
       LocalTime time, double velocity, LocalTime maxHour, E[] autonomies) {

    E duringAutonomy = autonomy;
    double gainedAutonomyPerMinute = 1016.6666666667;
    int vertices = g.numVertices();
    for (int i = 0; i < vertices; i++) {
        dist[i] = null;
        pathKeys[i] = null;
        visited[i] = false;
    }
    dist[g.key(vOrig)] = zero;
    arriveTimes[g.key(vOrig)] = time;
    departTimes[g.key(vOrig)] = time;
    descargaTimes[g.key(vOrig)] = null;

    while (vOrig != null) {
        visited[g.key(vOrig)] = true;
        LocalTime currentTime = departTimes[g.key(vOrig)];

        for (V vAdj : g.adjVertices(vOrig)) {
            if (vAdj.equals(vOrig)) {
                continue;
            }
            time = currentTime;
            LocalTime pathTime = LocalTime.of( hour: 0,  minute: 0);
            LocalTime pathTimeMax = LocalTime.of( hour: 20,  minute: 0);
            E edgeWeight = g.edge(vOrig, vAdj).getWeight();
            if (ce.compare(edgeWeight, autonomy) > 0) {
                continue;
            }
            E resetAutonomy = duringAutonomy;
            if (ce.compare(edgeWeight, duringAutonomy) > 0) {
            time = time.plus(Duration.ofMinutes((long) (Math.round((float) ((Integer) subtract.apply(autonomy,
                    duringAutonomy)).intValue()) / gainedAutonomyPerMinute)));
                pathTime = pathTime.plus(Duration.ofMinutes((long) (Math.round((float) ((Integer)
                        subtract.apply(autonomy, duringAutonomy)).intValue()) / gainedAutonomyPerMinute)));
                duringAutonomy = autonomy;
```

```
            }
            LocalTime timeAfterCharge = time;
            time = time.plus(Duration.ofMinutes((long) (60 * ((double) ((Integer) edgeWeight).intValue()
                    / 1000 / velocity))));
            pathTime = pathTime.plus(Duration.ofMinutes((long) (60 * ((double) ((Integer) edgeWeight).intValue()
                    / 1000 / velocity))));
            LocalTime endHour = ((Location) vAdj).getEndHour();
            if ((endHour != null && time.isAfter(endHour)) || time.isAfter(maxHour)
                    || pathTime.isAfter(pathTimeMax)) {
                duringAutonomy = resetAutonomy;
                time = currentTime;
                continue;
            }
            LocalTime arrivedTime = time;
            if (((Location) vAdj).isHub() && ((Location) vAdj).getEndHour().isAfter(time)) {
                time = time.plus(Duration.ofMinutes(new Random().nextInt( bound: 10) + 1));
                pathTime = pathTime.plus(Duration.ofMinutes(new Random().nextInt( bound: 10) + 1));
            }
            LocalTime descargasTime = time;
            E oldDist = dist[g.key(vAdj)];
            E newDist = sum.apply(dist[g.key(vOrig)], edgeWeight);

            if (oldDist == null || ce.compare(newDist, oldDist) < 0) {
                dist[g.key(vAdj)] = newDist;
                pathKeys[g.key(vAdj)] = vOrig;
                arriveTimes[g.key(vAdj)] = arrivedTime;
                afterChargeTimes[g.key(vAdj)] = timeAfterCharge;
                descargaTimes[g.key(vAdj)] = descargasTime;
                departTimes[g.key(vAdj)] = time;

                duringAutonomy = subtract.apply(duringAutonomy, edgeWeight);
                autonomies[g.key(vAdj)] = duringAutonomy;
            }
        }

        vOrig = null;
        LocalTime minArriveTime = null;
        for (V v : g.vertices()) {
            if (visited[g.key(v)] == false && arriveTimes[g.key(v)] != null) {
                if (vOrig == null || arriveTimes[g.key(v)].isBefore(minArriveTime)
                        && arriveTimes[g.key(v)].isAfter(arriveTimes[g.key(vOrig)])) {
                    minArriveTime = arriveTimes[g.key(v)];
                    vOrig = v;
                }
            }
        }

        if (vOrig != null) {
            duringAutonomy = autonomies[g.key(vOrig)];
            time = departTimes[g.key(vOrig)];
        }
    }
}
```

De seguida, todos os caminhos que foram determinados irão ser analisados e o caminho que tiver o maior número de hubs será o selecionado, sendo todos os seus diferentes tempos e as suas localizações, tal como a distância, armazenados e retornados ao utilizador.

```java
public int maximizedPath(String idOrigem, LocalTime time, int autonomy, int velocity, Boolean bigGraph,
                         LinkedList<Location> topPath, LinkedList<LocalTime> topArriveTimes,
                         LinkedList<LocalTime> topDepartTimes, LinkedList<LocalTime> topAfterChargeTimes,
                         LinkedList<LocalTime> topDescargaTimes) {
    Location curLocation = graphRepository.locationById(idOrigem, bigGraph);
    int topDistance = 0;
    Graph<Location, Integer> graph = getGraph(bigGraph);
    ArrayList<LinkedList<Location>> locationPaths = new ArrayList<>();
    ArrayList<Integer> locationDistance = new ArrayList<>();
    ArrayList<LinkedList<LocalTime>> arriveTimes = new ArrayList<>();
    ArrayList<LinkedList<LocalTime>> departTimes = new ArrayList<>();
    ArrayList<LinkedList<LocalTime>> afterChargeTimes = new ArrayList<>();
    ArrayList<LinkedList<LocalTime>> descargaTimes = new ArrayList<>();
    BinaryOperator<Integer> subtract = (a, b) -> a - b;
    LocalTime maxHour = getMaxHourToSearch(bigGraph);
    Algorithms.shortestPathsConstrained(graph, curLocation, Integer::compare, Integer::sum, subtract, zero: 0,
            locationPaths, locationDistance, arriveTimes, departTimes, afterChargeTimes, descargaTimes, autonomy,
            time, velocity, maxHour);
    int maxHubs = -1;
    for (int i = 0; i < locationPaths.size(); i++) {
        int totalHubs = 0;
        for(int j = 0; j < locationPaths.get(i).size(); j++){
            if(locationPaths.get(i).get(j).isHub()){
                totalHubs++;
            }
        }
        if(totalHubs > maxHubs){
            maxHubs = totalHubs;
            topPath.clear();
            topArriveTimes.clear();
            topDepartTimes.clear();
            topAfterChargeTimes.clear();
            topDescargaTimes.clear();
            topDistance = 0;
            for(int j = 0; j < locationPaths.get(i).size(); j++){
                topPath.add(locationPaths.get(i).get(j));
                topArriveTimes.add(arriveTimes.get(i).get(j));
                topDepartTimes.add(departTimes.get(i).get(j));
                topAfterChargeTimes.add(afterChargeTimes.get(i).get(j));
                topDescargaTimes.add(descargaTimes.get(i).get(j));
                topDistance = locationDistance.get(i);
            }
        }
    }
    return topDistance;
}
```

Complexidade: Nesta US temos dois métodos que influenciam a complexidade do método principal que os chama, sendo esses o "shortestPathsConstrained" com uma complexidade de O((V + E) log V) e o "getMaxHourToSearch" com uma complexidade de O(n). Existe ainda três ciclos for dentro do método principal cuja complexidade é $O(n^3)$. Tudo considerado, a complexidade final deste método é a seguinte:

$$O\big((V + E)\log V + n^3 + n\big)$$

US EI08

Problema: Encontrar para um produtor o circuito de entrega que parte de um local de origem, passa por N hubs com maior número de colaboradores uma só vez e volta ao local de origem minimizando a distância total percorrida. Considere como número de hubs: 5, 6 e 7.

Resolução: Para resolver este problema, iremos necessitar de pedir ao utilizador para indicar o ID do local de origem pelo qual pretende realizar entrega e o número de hubs (N) que irá passar, sendo este 5,6 ou 7.

```
private void deliveryCircuitPath(){
    Boolean bigGraph = graphOption();
    if (bigGraph == null) {
        return;
    }
    String idOrigem;
    int nHubs;
    do {
        idOrigem = Utils.readLineFromConsole( prompt: "Escreva o ID da localização de origem: (CT**)").toUpperCase();
    } while ((!idOrigem.contains("CT")) || (!idExists(idOrigem, bigGraph)));
    do {
        nHubs = Utils.readIntegerFromConsole( prompt: "Escreva o número de hubs (5, 6 e 7): ");
    } while (nHubs < 5 || nHubs > 7);
    int autonomy = Utils.readIntegerFromConsole( prompt: "Qual a autonomia do veículo?(km)");
    int velocity = Utils.readIntegerFromConsole( prompt: "Qual a velocidade média do veículo?(km/h)");
    int numCharges = 0;
    List<Location> bestPath = graphController.deliveryCircuitPath(idOrigem, nHubs, bigGraph);
    System.out.println("Localização de Origem: " + idOrigem);
    System.out.println("Número de Hubs: " + nHubs);
    int totalDistance = 0;
    int numCollaborations = 0;
    for (int i = 0; i < bestPath.size(); i++) {
        System.out.println("+--------------------+");
        System.out.println("| ID: " + bestPath.get(i).getId() + "            |");
        System.out.println("| Colaboradores: " + bestPath.get(i).getNumEmployees() + "   |");
        if (i < bestPath.size() - 1) {
            System.out.println("| Distância: " + graphController.getDistance(bestPath.get(i), bestPath.get(i+1),
                    bigGraph) + " m |");
            Integer distance = graphController.getDistance(bestPath.get(i), bestPath.get(i+1), bigGraph);
            if (distance == null) {
                System.out.println("Não existe caminho entre " + bestPath.get(i).getId() + " e " +
                        bestPath.get(i+1).getId());
                return;
            }
            totalDistance += distance;
        }
        System.out.println("+--------------------+");
        numCollaborations += bestPath.get(i).getNumEmployees();
        if (i < bestPath.size() - 1) {
            System.out.println("  |");
            System.out.println("  v");
        }
    }
    Pair<Duration, Integer> charge = graphController.getChargeDuration(bestPath, bigGraph, autonomy: autonomy * 1000);
    Duration chargeDuration = charge.getFirst();
    numCharges = charge.getSecond();
    Duration travDuration = graphController.getTravDuration(bestPath, bigGraph, velocity);
    Duration fullDuration = chargeDuration.plus(travDuration);
    System.out.println("\nNúmero de Colaboradores: " + numCollaborations);
    System.out.println("\nDistância Total: " + totalDistance + "m");
    System.out.println("Número de Carregamentos: " + numCharges);
    System.out.println("Tempo Total: " + formatDuration(fullDuration) + "\n");
    System.out.println("Tempo de Carregamento: " + formatDuration(chargeDuration) + "\n");

}
```

Deste modo, no método deliveryCircuitPath() verificamos, primeiramente, se o grafo é grande ou não através do método graphOption() e, depois, solicitamos ao utilizador as informações acima mencionadas, tendo estas que ser consideradas válidas. De seguida, chamamos o método ***graphController.deliveryCircuitPath(idOrigem,nHubs,bigGraph)***, que irá calcular o melhor caminho com base na localização de origem e no número de hubs, armazenando o resultado numa lista.

Neste método, imprimimos informações sobre cada local, que inclui o ID do mesmo, o número de colaboradores e a distância entre locais, calculando a distância total e o número total de colaboradores ao longo do caminho. Além disso, apresentamos estatísticas relacionadas com o número de carregamentos e a duração total da viagem através dos métodos ***getChargeDuration***, no qual calculamos a duração total e carregamentos e número de vezes que se para realizar um carregamento ao longo do melhor caminho fornecido, e ***getTravDuration***, no qual calculamos a duração total da viagem ao longo do melhor caminho fornecido.

```java
public List<Location> deliveryCircuitPath(String idOrigem, int nHubs, Boolean bigGraph) {
    return service.deliveryCircuitPath(idOrigem, nHubs, bigGraph);
}
```

```java
public List<Location> deliveryCircuitPath(String idOrigem, int nHubs, Boolean bigGraph) {
    if(!idExists(idOrigem, bigGraph)){
        System.out.println("A localização de origem não existe!");
        return null;
    }
    if(nHubs < 0 || nHubs > 7 || nHubs < 5){
        System.out.println("O número de hubs tem de ser entre 5 e 7!");
        return null;
    }
    try {
        List<List<Location>> paths = Algorithms.allPathsN(graphRepository.getGraph(bigGraph),
                graphRepository.locationById(idOrigem, bigGraph), graphRepository.locationById(idOrigem, bigGraph),
                nHubs);
        int maxCollaborations = 0;
        int minDistance = Integer.MAX_VALUE;
        List<Location> bestPath = new ArrayList<>();
        for (List<Location> path : paths) {
            int totalCollaborations = 0;
            int totalDistance = 0;
            for (int i = 0; i < path.size() - 1; i++) {
                if (i < path.size() - 1) {
                    Location location1 = path.get(i);
                    Location location2 = path.get(i + 1);
                    totalCollaborations += location1.getNumEmployees();
                    totalDistance += graphRepository.distanceLocations(location1,location2,bigGraph);
                }
            }
            if (totalCollaborations > maxCollaborations) {
                maxCollaborations = totalCollaborations;
                minDistance = totalDistance;
                bestPath = path;
            } else if (totalCollaborations == maxCollaborations && totalDistance < minDistance) {
                minDistance = totalDistance;
                bestPath = path;
            }
        }
        return bestPath;
    }catch (Exception e){
        System.out.println("Não foi possível encontrar um caminho :" + e.getMessage());
        return null;
    }
}
```

```java
public Integer distanceLocations(Location location1, Location location2, boolean bigGraph) {
    Edge<Location, Integer> distance = null;
    if (bigGraph) {
        if (this.basketDistribution.edge(location1, location2) == null) {
            return null;
        }
        distance = this.basketDistribution.edge(location1, location2);
    } else {
        if (this.smallGraph.edge(location1, location2) == null) {
            return null;
        }
        distance = this.smallGraph.edge(location1, location2);
    }

    return distance != null ? distance.getWeight() : null;
}
```

No método ***deliveryCircuitPath(idOrigem, nHubs, bigGraph)*** iremos obter uma lista de objetos do tipo "Location", que representa o caminho de entrega otimizado. Primeiro, verificamos se a localização inserida existe no grafo e se o número de hubs especificado se encontra dentro do intervalo permitido, que, neste caso, é entre 5 e 7. Posteriormente, calculamos todos os caminhos possíveis com até nHubs, a partir da localização de origem, através do algoritmo ***allPathsN***, obtendo uma lista dos caminhos. Após determinarmos a lista dos todos caminhos possíveis que se regem aquelas condições, calculamos o número total de colaboradores e a distância total para cada caminho, pois como se pretende obter o melhor caminho necessitamos de determinar o caminho com o maior número de colaboradores e, em caso de existirem dois caminhos com o mesmo número de colaboradores, retornamos o caminho com a menor distância total.

```
public static <V, E> List<List<V>> allPathsN(Graph<V, E> g, V vOrig, V vDest, int n) {
    List<List<V>> paths = new ArrayList<>();
    List<V> hubs = new ArrayList<>();
    for (V v : g.vertices()) {
        if (((Location) v).isHub() && !v.equals(vOrig)) {
            hubs.add(v);
        }
    }

    depthFirstSearchWithHubs(g, vOrig, vDest, n, hubs, new ArrayList<>(Collections.singletonList(vOrig)), paths,
            new HashSet<>());
    return paths;
}

public static <V, E> void depthFirstSearchWithHubs(Graph<V, E> g, V v, V vDest, int n, List<V> hubs,
        List<V> path, List<List<V>> paths, Set<V> visited) {
    if (path.size() > n) {
        return;
    }

    if (v.equals(vDest)) {
        paths.add(new ArrayList<>(path));
    }

    for(V vAdj : g.adjVertices(v)) {
        if (!visited.contains(vAdj)) {// Skip if vertex has been visited
            visited.add(vAdj);
            if (hubs.contains(vAdj)) {
                path.add(vAdj);
                depthFirstSearchWithHubs(g, vAdj, vDest, n, hubs, path, paths, new HashSet<>(visited));
                path.remove( index: path.size() - 1);
            } else {
                depthFirstSearchWithHubs(g, vAdj, vDest, n, hubs, path, paths, new HashSet<>(visited));
            }
        }
    }
}
```

O algoritmo apresentado implementa um algoritmo de busca em profundidade modificado, no qual se pretende encontrar todos os caminhos possíveis de um ponto de origem para um destino num grafo, que, neste caso, representam a mesma localização, e com uma limitação do número de hubs (pontos de conexão) permitidos ao longo do percurso, sendo este número especificado pelo parâmetro 'n'.

Na função ***allPathsN***, são inicializadas listas para armazenar os caminhos encontrados e os hubs do grafo e, depois, invocamos o método ***depthFirstSearchWithHubs*** para realizar a busca em profundidade restrita pelos hubs, obtendo a lista que contem todos os caminhos encontrados. Inicialmente, verificamos se o tamanho do caminho excede o número máximo de hubs permitidos e, de seguida, se o vértice atual coincide com o destino, o caminho é adicionado à lista de caminhos encontrados. Posteriormente, iteramos os vértices adjacentes ao vértice atual, em que caso um vértice adjacente ainda não tenha sido visitado, é marcado como visitado. Se o vértice adjacente é um hub, é adicionado ao caminho, continuando a busca recursiva. Se não é um hub, a busca recursiva prossegue sem incluir o vértice no caminho. A utilização de um conjunto **visited** permite evitar a repetição de visitas aos mesmos vértices durante a busca recursiva. Por último, é removido o último vértice adicionado ao caminho, garantindo que o caminho seja ajustado corretamente.

Complexidade: A complexidade desta US é predominantemente influenciada pela complexidade do método "allPathsN" com uma complexidade de O(V + E), que é proporcional ao número de arestas do grafo. No entanto, este método apresenta um outro designado por "depthFirstSearchWithHubs", que se baseia numa pesquisa DepthFirstSearch, em que consideramos os vértices visitados. Assim, a complexidade final deste método é a seguinte:

$$O(V\text{^}2(V + E))$$

US EI09

Problema: Organizar as localidades do grafo em N clusters que garantam apenas 1 hub por cluster de localidades. Os clusters devem ser obtidos iterativamente através da remoção das ligações com o maior número de caminhos mais curtos entre localidades até ficarem clusters isolados. Não deverá fornecer soluções de clusters de localidades sem o respetivo hub.

Resolução: Para resolver este problema, primeiro o programa pede o número de clusters que o utilizador pretende obter. Depois, o programa obtém N hubs e os põe numa lista, sendo que N é igual ao número de clusters pretendidos.

```
public static <V, E> Map<V, LinkedList<V>> divideGraphN(Graph<V, E> g, Set<V> hubList, Comparator<E> ce,BinaryOperator<E> sum, E zero, int numClusters){
    List<Map.Entry<Edge<V, E>, Integer>> sortedMap = sortMap(g, ce, sum, zero);

    for (Map.Entry<Edge<V, E>, Integer> entry : sortedMap){
        Graph<V, E> gBackup = g.clone();

        g.removeEdge(entry.getKey().getVOrig(), entry.getKey().getVDest());

        LinkedList<V> visitados = new LinkedList<>();

        boolean containsHub = true;

        int numClustersAtuais = 0;

        for(V vertice : g.vertices()){
            if(!visitados.contains(vertice)) {
                LinkedList<V> cluster = new LinkedList<>();
                DepthFirstSearch(g, vertice, cluster);
                if(cluster.size() == g.numVertices()){
                    break;
                } else {
                    if(Collections.disjoint(cluster, hubList)){
                        containsHub = false;
                        break;
                    } else {
                        numClustersAtuais++;
                        visitados.addAll(cluster);
                    }
                }
            }
        }
        if(!containsHub){
            g = gBackup.clone();
        }
        if(numClustersAtuais == numClusters){
            break;
        }
    }

    return createClusterLists(g, hubList);
}
```

O algoritmo primeiro cria uma lista de todas as arestas do grafo, organizadas de forma decrescente pelo número de número de caminhos mais curtos. Os algoritmos que criam essa lista são os seguintes:

```java
private static <V, E> List<Map.Entry<Edge<V, E>, Integer>> sortMap(Graph<V, E> g, Comparator<E> ce,BinaryOperator<E> sum, E zero){
    Map<Edge<V, E>, Integer>  numShortPaths = new LinkedHashMap<>();
    for (Edge<V, E> e : g.edges()){
        numShortPaths.put(e, 0);
    }

    for (V vertice : g.vertices()){
        ArrayList<LinkedList<V>> verticePaths = new ArrayList<>();
        ArrayList<E> verticeDistance = new ArrayList<>();
        shortestPaths(g, vertice, ce, sum, zero, verticePaths, verticeDistance);
        for (LinkedList<V> list : verticePaths){
            numShortPaths(g, list, numShortPaths);
        }
    }
    List<Map.Entry<Edge<V, E>, Integer>> sortedMap = new ArrayList<>(numShortPaths.entrySet());
    sortedMap.sort(Map.Entry.comparingByValue(Integer::compareTo));
    sortedMap = sortedMap.reversed();

    return sortedMap;
}

1 usage   José Sá
private static <V, E> void numShortPaths(Graph<V, E> g, LinkedList<V> listVertices, Map<Edge<V, E>, Integer>  numShortPaths){
    for (int i = 0; i < listVertices.size() - 1; i++){
        V vOrigin = listVertices.get(i);
        V vDest = listVertices.get(i + 1);
        int currentNum = numShortPaths.get(g.edge(vOrigin, vDest));
        numShortPaths.put(g.edge(vOrigin, vDest), currentNum + 1);
    }
}
```

Como se pode ver, para cada vértice no grafo é criada a lista de todos os caminhos curtos que começam nesse vértice, e depois para cada aresta que está nessa lista é adicionado um ao valor no mapa que guarda o número de caminhos curtos que passam por cada aresta. Desse mapa é criada uma lista de entradas em ordem decrescente do valor.

Voltando ao algoritmo principal, após obter a lista organizada das arestas, o algoritmo passa por cada uma delas em ordem e as tira do grafo. A seguir, o algoritmo vai passar por todos os vértices, e se esse vértice não estiver na lista de vértices visitados criada no início do loop, então a função preforma uma Depth First Search, que determina todos os vértices ligados ao vértice inicial. Como o DFS só chega aos vértices que estão ligados ao vértice inicial, assim podemos obter uma lista de vértices que estão num determinado cluster. A lista de vértices visitados é guardada numa lista temporária denominada “cluster”, e o algoritmo faz dois ifs. O primeiro determina se o tamanho do cluster é igual ao tamanho do grafo inicial. Se for, isso significa que o grafo ainda não está segmentado, ou seja, é preciso retirar mais arestas, por isso o loop dá break logo de seguida. O segundo if vê se o cluster criado contém pelo menos um hub. Se não tiver, a variável “containsHub” é dada como falsa e o loop dá break. Se o cluster tiver um hub, então é dado como valido e o número de hubs aumenta um valor, e a lista temporária “cluster” é adicionada à lista de vértices visitados.

Após todos os vértices terem sido visitados, antes de remover outra aresta o algoritmo faz dois if checks: Primeiro, vê se a variável containsHub é falsa. Se for, isso significa que a última aresta removida criou um cluster sem hubs, e a função restaura essa aresta. O segundo if vê se a instância atual do loop criou a quantidade de clusters desejados. Se sim, então não é necessário retirar mais arestas, e o loop para.

Por fim, passa-se o grafo final para um algoritmo que o transforma num mapa de hubs (chave) e lista de vértices (valor), que será o valor de retorno.

```java
private static <V, E> Map<V, LinkedList<V>> createClusterLists(Graph<V, E> g, Set<V> hubList){
    Map<V, LinkedList<V>> returnMap = new HashMap<>();
    for (V hub : hubList){
        LinkedList<V> cluster = new LinkedList<>();
        DepthFirstSearch(g, hub, cluster);
        returnMap.put(hub, cluster);
    }
    return returnMap;
}
```

O algoritmo é simples, simplesmente vê quais são os vértices ligados a um hub, e cria uma entrada do mapa com esse hub como chave, e a lista de vértices ligados como valor.

Complexidade: Primeiro vemos a complexidade da função sortMap. Esta função usa o algoritmo de Dijkstra para obter os caminhos curtos. Como o algoritmo é chamado para cada vértice do grafo, a complexidade é O(((V + E) * log(V)) * V), onde V é o número de vértices no grafo e E é o número de arestas.

A complexidade do algoritmo principal, no melhor caso, é O((N-1) * V * (V + E)), onde N é o número de clusters pretendidos e V é o número de vértices. Isto é o caso porque o número mínimo de arestas para teoreticamente dividir um grafo em N clusters é N – 1, logo se as primeiras N – 1 arestas removidas serem as únicas necessárias para obter os clusters, então esse é o melhor caso (para além disso, para cada aresta removida o algoritmo DFS é rodado pelo menos V vezes, ou seja, V * (V + E), onde (V + E) é a complexidade de DFS.

No pior caso, é O(N * E * V * (V + E)), já que teríamos de remover todas as arestas para criar os clusters.

Para a criação da lista, a complexidade é O((V + E) * N), já que o algoritmo DFS é rodado N (número de clusters) vezes.

No final, a complexidade do algoritmo todo é O((((V + E) * log(V)) * V) + ((N-1) * V * (V + E)) + ((V + E) * N))) no melhor caso, e O((((V + E) * log(V)) * V) + (N * E * V * (V + E)) + ((V + E) * N))) no pior caso.

US EI010

Problema: Considere uma rede constituída por N hubs definidos segundo o critério de centralidade e admita que nesta rede entre dois hubs circula sempre o mesmo veículo com capacidade limitada a X cabazes de produtos. Pretende-se para um hub origem e um hub destino determinar a rede que permita transportar o número máximo de cabazes.

Resolução: Para a realização desta US, começamos por gerar um grafo de hubs a partir de um dos grafos iniciais com hubs previamente designados pelo critério de centralidade.

```
Boolean bigGraph = graphOption();
if (bigGraph == null) {
    return;
}
MapGraph<Location, Integer> graph = graphController.getGraph(bigGraph);
MapGraph<Location, Integer> hubGraph = graphController.filterGraph(graph);
if (hubGraph.vertices().isEmpty()) {
    System.out.println(x:"Erro ao filtrar o grafo");
    return;
}
```

Através do grafo de hubs e dados os pontos de origem e destino, aplicamos o algoritmo de Ford-Fulkerson. Este algoritmo permite-nos descobrir o fluxo máximo numa rede, assim como o fluxo em cada uma das suas arestas.

```
String idOrigem;
String idDestino;
do {
    idOrigem = Utils.readLineFromConsole(prompt:"Escreva o ID do HUB de origem: (CT**)").toUpperCase();
} while ((!idOrigem.contains(s:"CT")) || (!hubGraph.validVertex(new Location(idOrigem))));
do {
    idDestino = Utils.readLineFromConsole(prompt:"Escreva o ID do HUB de destino: (CT**)").toUpperCase();
} while ((!idDestino.contains(s:"CT")) || (!hubGraph.validVertex(new Location(idDestino))));

Pair<Integer, MapGraph<Location, Integer>> result = graphController.maximumCapacity(hubGraph, new Location(idOrigem), new Location(idDestino));
```

Este algoritmo utiliza a função “bfs”, uma adaptação do algoritmo “Breadth-First Search” para encontrar um caminho da origem para o destino no grafo residual com capacidade positiva. A cada iteração, o valor do fluxo máximo e o grafo residual são atualizados.

```java
public static <V, E> Pair<Integer, Map<V, Map<V, Integer>>> fordFulkerson(Graph<V, E> g, V source, V sink) {
    int maxFlow = 0;

    Map<V, Map<V, Integer>> residualGraph = new HashMap<>();
    for (V vertex : g.vertices()) {
        residualGraph.put(vertex, new HashMap<>());
        for (V adjVertex : g.adjVertices(vertex)) {
            Edge<V, E> edge = g.edge(vertex, adjVertex);
            int weight = (Integer) edge.getWeight();
            residualGraph.get(vertex).put(adjVertex, weight);
            if (!residualGraph.containsKey(adjVertex)) {
                residualGraph.put(adjVertex, new HashMap<>());
            }
            residualGraph.get(adjVertex).put(vertex, weight);
        }
    }

    Map<V, V> parent = new HashMap<>();

    while (bfs(residualGraph, source, sink, parent)) {
        int pathFlow = Integer.MAX_VALUE;
        List<V> path = new ArrayList<>();
        for (V v = sink; v != source; v = parent.get(v)) {
            V u = parent.get(v);
            pathFlow = Math.min(pathFlow, residualGraph.get(u).get(v));
            path.add(index:0, v);
        }
        path.add(index:0, source);

        for (V v = sink; v != source; v = parent.get(v)) {
            V u = parent.get(v);
            int residualCapacity = residualGraph.get(u).get(v);
            int reverseCapacity = residualGraph.get(v).get(u);
            if (pathFlow <= residualCapacity) {
                residualGraph.get(u).put(v, residualCapacity - pathFlow);
                residualGraph.get(v).put(u, reverseCapacity + pathFlow);
            }
        }

        maxFlow += pathFlow;
    }

    return new Pair<>(maxFlow, residualGraph);
}
private static <V> boolean bfs(Map<V, Map<V, Integer>> residualGraph, V source, V sink, Map<V, V> parent) {
    Set<V> visited = new HashSet<>();
    Queue<V> queue = new LinkedList<>();
    queue.add(source);
    visited.add(source);

    while (!queue.isEmpty()) {
        V vertex = queue.poll();
        for (Map.Entry<V, Integer> entry : residualGraph.get(vertex).entrySet()) {
            V adjVertex = entry.getKey();
            Integer capacity = entry.getValue();
            if (!visited.contains(adjVertex) && capacity > 0) {
                queue.add(adjVertex);
                visited.add(adjVertex);
                parent.put(adjVertex, vertex);
                if (adjVertex.equals(sink)) {
                    return true;
                }
            }
        }
    }

    return false;
}
```

Após a aplicação do algoritmo, obtemos não apenas o valor do fluxo máximo, mas também o grafo residual, que representa as capacidades não utilizadas nas arestas. No serviço através da função “maximumCapacity”, realizamos a subtração entre o grafo original e o residual, resultando no grafo de fluxo máximo. Esta operação destaca as arestas e fluxos que compõem a rota mais eficiente para o transporte de cabazes entre os hubs.

```java
public Pair<Integer, MapGraph<Location, Integer>> maximumCapacity(MapGraph<Location, Integer> graph,
        Location origin, Location destination) {

    Pair<Integer, Map<Location, Map<Location, Integer>>> result = Algorithms.fordFulkerson(graph, origin,
            destination);

    Map<Location, Map<Location, Integer>> residualGraph = result.getSecond();

    MapGraph<Location, Integer> maxFlowGraph = new MapGraph<>(directed:true);

    for (Location vertex : residualGraph.keySet()) {
        for (Location adjVertex : residualGraph.get(vertex).keySet()) {
            Edge<Location, Integer> edge = graph.edge(vertex, adjVertex);
            if (edge != null) {
                Integer originalCapacity = edge.getWeight();
                Integer residualCapacity = residualGraph.get(vertex).get(adjVertex);
                if (originalCapacity != null && residualCapacity != null) {
                    Integer flowOnEdge = originalCapacity - residualCapacity;
                    if(flowOnEdge > 0) {
                        maxFlowGraph.addVertex(vertex);
                        maxFlowGraph.addVertex(adjVertex);
                        maxFlowGraph.addEdge(vertex, adjVertex, flowOnEdge);
                    } else if (flowOnEdge < 0) {
                        maxFlowGraph.addVertex(adjVertex);
                        maxFlowGraph.addVertex(vertex);
                        maxFlowGraph.addEdge(adjVertex, vertex, -flowOnEdge);
                    }
                }
            }
        }
    }

    return new Pair<>(result.getFirst(), maxFlowGraph);
}
```

Em seguida, para fins de análise, transformamos esse grafo de fluxo máximo em um arquivo CSV utilizando a função “`generateDataCSV`”

```java
System.out.println("Hub de Origem: " + idOrigem);
System.out.println("Hub de Destino: " + idDestino);
if (result == null || result.getFirst() == 0) {
    System.out.println(x:"----------------------");
    System.out.println(x:"| Não existe fluxo ! |");
    System.out.println(x:"----------------------");
    return;
}
System.out.println("Capacidade Máxima: " + result.getFirst());
if (Utils.confirm(message:"Deseja ver o grafo?:")) {
    graphController.generateDataCSV(result.getSecond());
    String filePath = "output/" + graphController.getLatestFileFromDirectory(string:"esinf/output/");
    openGraphViewer(filePath);
}
```

Complexidade: A complexidade do algoritmo Ford-Fulkerson é de O((VxE)xf⋆) e o resto do método principal também contém dois ciclos “for” cuja complexidade é de $O(n^2)$. Sendo assim, a complexidade total desta US é:

$$O(((V \times E) \times f\star) + n^2)$$

US EI11

Problema: Como Product Owner pretendo carregar um ficheiro com os horários de funcionamento de uma lista de hubs.

Considere o seguinte exemplo:

CT1, 14:00, 17:00

CT214, 11:00, 15:30

Resolução: Nesta US, tal como nas anteriores, damos a opção de o utilizador escolher o grafo que pretende utilizar. Após isso o utilizador deve escolher qual o ficheiro de horários que pretende utilizar para definir os mesmos, sendo que estes ficheiros são identificados através da função "getFilesFromDirectory" que irá chamar a função "importFilesNames". Por fim é chamada a função "importOpeningHours".

```
public List<String> importFilesNames(String folderPath) {
    URL url = getClass().getClassLoader().getResource(folderPath);
    File folder = new File(url.getPath());
    String[] fileNames = folder.list();
    return Arrays.stream(fileNames).map(fileName -> folderPath + "/" + fileName).collect(Collectors.toList());
}
```

```
private void addSchedule() {
    try {
        Boolean bigGraph = graphOption();
        if (bigGraph == null) {
            return;
        }
        List<String> files = importController.getFilesFromDirectory( string: "esinf/schedules");
        int fileNumber = Utils.showAndSelectIndex(files,  header: "\n=========Lista de ficheiros=========");
        boolean check = importController.importOpeningHours(files.get(fileNumber).toString(), bigGraph);
        if (check) {
            System.out.println("Horários adicionados com sucesso");
        } else {
            System.out.println("Erro ao adicionar todos os horários");
        }

    } catch (IOException e) {
        System.out.println("Ocorreu um erro na pesquisa dos caminhos dos ficheiros: " + e.getMessage());
    }
}
```

Esta função é responsável por atribuir os horários a todos os ID's existentes no ficheiro de horários, filtrando através de um "if" apenas as que existem no grafo e que foram definidas como Hubs.

```
public boolean createOpeningHours(List<String> horarios, boolean bigGraph) throws Exception {
    boolean allAdded = true;
    for (String horario : horarios) {
        String[] line = horario.split( regex: ",");
        Location location = graphRepository.locationById(line[0], bigGraph);
        if (location != null && location.isHub() == true) {
            LocalTime startHour = LocalTime.parse(line[1]);
            LocalTime endHour = LocalTime.parse(line[2]);
            location.setStarHour(startHour);
            location.setEndHour(endHour);
            System.out.println("Horário da Localização com ID " + line[0] + " adicionada com sucesso");
        } else {
            allAdded = false;
            System.err.println("Horário da Localização com ID " + line[0] + " não existe ou não é um hub");
        }
    }
    return allAdded;
}
```

Complexidade: A complexidade deste método baseia-se única e exclusivamente na iteração do ficheiro e das localizações. Sendo assim, a sua complexidade é a seguinte:

Worst-case scenario: $O(V * N)$

Best-case scenario: $O(\log(V) * N)$

O pior caso acontece se existirem todos as localizações (V) no ficheiro de horários (N), já o melhor caso acontece caso só existam algumas localizações no ficheiro de horários.

Melhoramentos Possíveis

Após a finalização do trabalho, apercebemo-nos de que poderíamos ter criado algoritmos mais eficientes, no entanto fizemos o máximo de pesquisa possível e alargamos os nossos conhecimentos em algoritmos de grafos.

Conclusão

Em suma, foram elaborados com sucesso todos os exercícios propostos para a realização do Segundo Projeto e através do relatório foi atingido o objetivo de expor o trabalho realizado pelos membros da equipa na aplicação desenvolvida.

A aplicação foi concretizada com conhecimentos adquiridos na disciplina ESINF que foram alargados através de pesquisa utilizando diferentes recursos.