ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Instituto Superior de Engenharia do Porto

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA
Instituto Superior de Engenharia do Porto

Departamento de Engenharia Informática

Relatório ESINF
Trabalho 3

Grupo 322

Ricardo Gonçalves, 1221720

Diogo Espírito Santo, 1212039

Paulo Moreira, 1212044

Rafael Carolo, 1212047

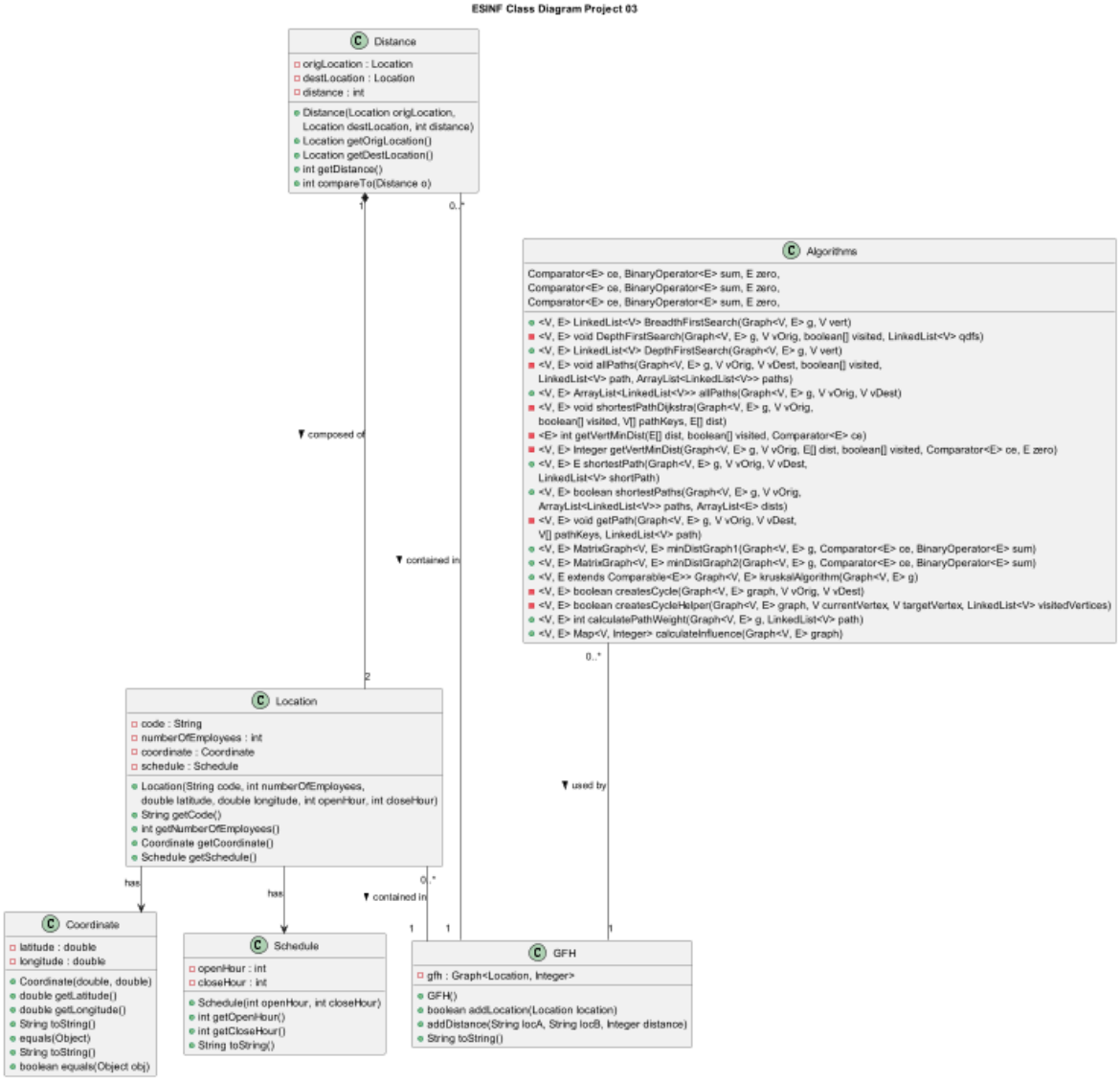
Orientador

Professor Telmo Matos

Porto, 26 de novembro de 2023

ÍNDICE

DIAGRAMA DE CLASSES



FUNCIONALIDADE 1

EXERCÍCIO

"Construir a rede de distribuição de cabazes a partir dos ficheiros (distancias xxx.csv e locais xxx.csv) com o formato disponibilizado. O grafo deve ser implementado usando a representação mais adequada para realizar de forma eficiente as funcionalidades pretendidas."

DESCRIÇÃO

A importação dos dados foi feita com leitura do csv, e alocação direta num MapGraph(). O objeto, mapa é depois reservado em memória, para ser utilizado durante a execução apenas.

ALGORITMO

Nome: importExcel

Input: caminho para locais.csv e distancias.csv

Output: boolean, true se concluído com sucesso e false se falhar

```
private boolean importExcel(String locaisPath, String distanciasPath)
{ // O(n)
    try {
        // Auxiliary path initializer
        File currentDirFile = new File("."); // O(1)
        String pathSufix = currentDirFile.getAbsolutePath(); // O(1)
        pathSufix = pathSufix.substring(0, pathSufix.length() - 1); //
O(1)

        // Path to locais Excel file
        String excelLocaisPath = pathSufix + locaisPath; // O(1)

        // Run method for Excel file
        importLocais(excelLocaisPath); // O(n)

        // Auxiliary path initializer
        currentDirFile = new File("."); // O(1)
        pathSufix = currentDirFile.getAbsolutePath(); // O(1)
        pathSufix = pathSufix.substring(0, pathSufix.length() - 1); //
O(1)

        // Path to distancias Excel file
        String excelDistanciasPath = pathSufix + distanciasPath; //
O(1)

        // Run method for Excel file
        importDistancias(excelDistanciasPath); // O(n)

        return true;
```

```
    } catch (Exception e) {
        e.printStackTrace(); // O(1)
        return false;
    }
}
```

ANÁLISE COMPLEXIDADE

1. importLocais(excelLocaisPath); // O(n)
2. importLocais(excelLocaisPath);

Complexidade do algoritmo = **O(n) + O(n) = O(n)**

****

ALGORITMO

Nome: importLocais(String excelLocaisPath)

Input: caminho para locais.csv

Output: void

```
private void importLocais(String excelLocaisPath){ // O(n)
    try {
        // load the data from file
        Scanner sc = new Scanner(new
FileReader(excelLocaisPath)).useDelimiter("\n"); // O(1)

        // checking each line, adding it to string and each of this to
arraylist
        sc.next(); // O(1)
        while (sc.hasNext()) { // O(n)
            String[] array = sc.next().split(","); // O(n)

            // get data to variables
            String code = array[0]; // O(1)
            int numberOfEmployees =
Integer.parseInt(array[0].substring(2)); // O(1)
            double latitude = Double.parseDouble(array[1]); // O(1)
            double longitude = Double.parseDouble(array[2]); // O(1)
            int openHour; // O(1)
            int closeHour; // O(1)
            if (numberOfEmployees <= 105) { // O(1)
                openHour = 9; // O(1)
                closeHour = 14; // O(1)
            } else if (numberOfEmployees <= 215) { // O(1)
                openHour = 11; // O(1)
                closeHour = 16; // O(1)
            } else { // O(1)
                openHour = 12; // O(1)
                closeHour = 17; // O(1)
            }

            Location location = new Location(code, numberOfEmployees,
latitude, longitude, openHour, closeHour); // O(1)
```

```
            // add to matrix
            gfh.addLocation(location); // O(1)
        }
    } catch (IOException e) {
        e.printStackTrace(); // O(1)
    }
}
```

ANÁLISE COMPLEXIDADE

1. while (sc.hasNext()) { // O(n)

Complexidade do algoritmo = **O(n)**

****

ALGORITMO

Nome: importDistancias(String excelDistanciasPath)

Input: caminho para distancias.csv

Output: void

```
private void importDistancias(String excelDistanciasPath){ // O(n)
    try {
        // load the data from file
        Scanner sc = new Scanner(new
FileReader(excelDistanciasPath)).useDelimiter("\n"); // O(1)

        // checking each line, adding it to string and each of this to
arraylist
        sc.next(); // O(1)
        while (sc.hasNext()) { // O(n)
            String[] array = sc.next().split(","); // O(1)

            // get data to variables
            String codeOrig = array[0]; // O(1)
            String codeDest = array[1]; // O(1)
            Integer distance =
Integer.parseInt(array[2].split("\r")[0]); // O(1)

            // add to matrix
            gfh.addDistance(codeOrig, codeDest, distance); // O(1)
        }
    } catch (IOException e) {
        e.printStackTrace(); // O(1)
    }
}
```

ANÁLISE COMPLEXIDADE

1. while (sc.hasNext()) { // O(n)

Complexidade do algoritmo = **O(n)**

FUNCIONALIDADE 2

EXERCÍCIO

Determinar os vértices ideais para a localização de N hubs de modo a otimizar a rede de distribuição segundo diferentes critérios:

-Influência: vértices com maior grau.

-Proximidade: vértices mais próximos dos restantes vértices.

-Centralidade: Vértices com maior número de caminhos mínimos que passam por eles.

DESCRIÇÃO

Nesta US, os algoritmos de grafos desempenham um papel fundamental na otimização da rede de distribuição. O algoritmo de Dijkstra calcula eficazmente os caminhos mais curtos entre localizações, tendo em conta factores como as distâncias. Os algoritmos de ordenação são utilizados para ordenar os vértices por ordem decrescente com base na centralidade, facilitando a identificação de localizações altamente centrais.

ALGORITMO

Nome: calculateInfluence

Input: grafo

Output: Mapa com que associa cada localização (vértice) com o seu valor de influência (grau), ordenado por ordem decrescente.

ANÁLISE COMPLEXIDADE

```
3 usages   1212039
public static <V, E> Map<V, Integer> calculateInfluence(Graph<V, E> graph) { //O(V)
    Map<V, Integer> influenceMap = new HashMap<>();

    for (V vertex : graph.vertices()) {
        influenceMap.put(vertex, graph.adjVertices(vertex).size());
    }

    return influenceMap;
}
```

ALGORITMO

Nome: calculateProximities

Input: grafo, vértice source de onde são calculadas as próximidades

Output: Lista de valores de proximidade para cada vértice, ordenado por ordem decrescente.

ANÁLISE COMPLEXIDADE

```
2 usages  1212039
public static List<Double> calculateProximities(Graph<Location, Integer> graph, Location sourceVertex) { // O(n^2)
    List<Double> proximities = new LinkedList<>();

    for (Location vertex : graph.vertices()) {
        if (!vertex.equals(sourceVertex)) {
            double proximity = Algorithms.shortestPath(graph, sourceVertex, vertex, Comparator.naturalOrder(), Integer::sum, zero: 0, new LinkedList<>());
            proximities.add(proximity);
        } else {
            // Proximity from the vertex to itself is 0
            proximities.add(0.0);
        }
    }

    return proximities;
}
```

ALGORITMO

Nome: buildCentralityInfoList

Input: grafo, Lista de distâncias do vértice source para todos os outros vértices

Output: Lista de USEI02_DTO que contém informação dos vértices e a sua centralidade (número de caminhos mínimos que passam por si)

ANÁLISE COMPLEXIDADE

```
1 usage  1212039
private List<USEI02_DTO> buildCentralityInfoList(Graph<Location, Integer> graph, ArrayList<Integer> distances) { //O(n)
    List<USEI02_DTO> centralityInfoList = new ArrayList<>();
    for (Location vertex : graph.vertices()) {
        int index = graph.key(vertex);
        centralityInfoList.add(new USEI02_DTO(vertex, distances.get(index)));
    }

    // Order the list by centrality in descending order
    centralityInfoList.sort(Comparator.comparingInt(USEI02_DTO::getCentrality).reversed());

    return centralityInfoList;
}
```

FUNCIONALIDADE 3

EXERCÍCIO

"Dado um veículo, a sua autonomia e atendendo a que os carregamentos só podem ser feitos nas localidades, determinar o percurso mínimo possível entre os dois locais mais afastados da rede de distribuição, indicando o número de paragens necessárias para carregamentos do veículo."

DESCRIÇÃO

Para esta US foram implementados métodos que utilizam os algoritmos de Dijkstra e Floyd Warshall para determinar o caminho mais curto entre os dois vértices mais afastados de um grafo. A partir do grafo obtido através dos ficheiros csv, determinou-se o grafo com os caminhos mais curtos (Floyd Warshall). Escolheu-se o maior desses caminhos guardando o vértice de origem e destino. Através desses vértices, obtemos uma lista com os locais todos desse caminho curto (Dijkstra).

ALGORITMO

Nome: getBiggestShortestPathData

Input: Autonomia do veículo

Output: O caminho pretendido, os locais em que o veículo precisa de ser carregado, a distância total da viagem, a quantidade de carregamentos e a autonomia mínima necessária para completar a viagem.

```
public USEI03_DTO getBiggestShortestPathData(Integer autonomiaMax) {
//O(V^3)
    if (autonomiaMax == null) { //O(1)
        return null; //O(1)
    }
    MatrixGraph<Location, Integer> matrixGraphConverted = new
MatrixGraph<>(gfh); //O(1)
    MatrixGraph<Location, Integer> matrixGraph =
Algorithms.minDistGraph1(matrixGraphConverted, Integer::compare,
Integer::sum); //O(V^3)

    Integer maxDistance = 0; //O(1)
    Location origin = null; //O(1)
    Location destination = null; //O(1)
    Collection<Edge<Location, Integer>> allDistances =
matrixGraph.edges(); //O(V^2)
    for (Edge<Location, Integer> distance : allDistances) { //O(V^2)
        if (distance.getWeight() > maxDistance) { //O(1)
            maxDistance = distance.getWeight(); //O(1)
            origin = distance.getVOrig(); //O(1)
```

```
            destination = distance.getVDest(); //O(1)
        }
    }

    USEI03_DTO dto = getShortestPathData(origin, destination,
autonomiaMax); //O(V^2)

    return dto; //O(1)
}
```

ANÁLISE COMPLEXIDADE

minDistGraph1(): O(V^3)

Logo,

Complexidade do algoritmo = **O(V^3)**

****

ALGORITMO

Nome: getShortestPathData

Input: Vértice de origem, vértice de destino e autonomia do veículo

Output: O mesmo output que o algoritmo acima (getBiggestShortestPathData), no entanto, é neste método que os dados que são passados para o DTO são calculados.

```
 public USEI03_DTO getShortestPathData(Location origin, Location
destination, int autonomiaMax) { //O(V^2)
    if (origin == null || destination == null) { //O(1)
        return null; //O(1)
    }
    LinkedList<Location> shortPath = new LinkedList<>(); //O(1)
    Integer pathLength = Algorithms.shortestPath(gfh, origin,
destination, Integer::compare, Integer::sum, 0, shortPath); //O(V^2)
    int autonomiaAtual = autonomiaMax; //O(1)
    int numberOfCharges = 0; //O(1)
    int distance = 0; //O(1)
    int minimumAutonomy = 0; //O(1)
    boolean tripIsPossible = true; //O(1)
    ArrayList<Location> chargeLocations = new ArrayList<>(); //O(1)


    for (int i = 0; i < shortPath.size() - 1; i++) {
        Location location = shortPath.get(i); //O(1)
        Location nextLocation = shortPath.get(i + 1); //O(1)
        distance = gfh.edge(location, nextLocation).getWeight();
//O(1)
        if (distance > autonomiaMax) { //O(1)
            tripIsPossible = false; //O(1)
            if (distance > minimumAutonomy) { //O(1)
```

```
                minimumAutonomy = distance; //O(1)
            }

        } else {
            if (autonomiaAtual - distance <= 0) { //O(1)
                chargeLocations.add(location); //O(1)
                numberOfCharges++; //O(1)
                autonomiaAtual = autonomiaMax; //O(1)
            } else {
                autonomiaAtual -= distance; //O(1)
            }
        }
    }
    USEI03_DTO dto = new USEI03_DTO(tripIsPossible, shortPath,
chargeLocations, pathLength, numberOfCharges, autonomiaMax,
minimumAutonomy); //O(1)
    return dto; //O(1)
}
```

ANÁLISE COMPLEXIDADE

shortestPath(): O(V^2)

Logo,

Complexidade do Algoritmo = **O(V^2)**

****

ALGORITMO

Nome: showShortestPath

Input: O caminho pretendido, os locais de carregamento, a distância de viagem, a quantidade de carregamentos, o local de origem, o local de destino e a autonomia mínima necessária à viagem em questão.

Output: O caminho com todos os locais por onde passa o veículo. Imprime os dados e também o trajeto.

```
public LinkedList<Location> showShortestPath(boolean tripIsPossible,
LinkedList<Location> shortPath, ArrayList<Location> chargeLocations, Integer
pathLength, int numberOfCharges, Location origin, Location destination, int
minimumAutonomy) { //O(V)
    if (!tripIsPossible) { //O(1)
        Utils.showMessageColor("\nO veículo não tem autonomia suficiente para
efetuar a viagem. Precisava de ter pelo menos " + minimumAutonomy + "m de
autonomia.", AnsiColor.RED); //O(1)
        shortPath = null; //O(1)
        return shortPath; //O(1)
    }
    Utils.showMessageColor("Local de origem: ", AnsiColor.BLUE); //O(1)
    System.out.println(origin.getCode()); //O(1)
    Utils.showMessageColor("\nLocal de destino: ", AnsiColor.BLUE); //O(1)
    System.out.println(destination.getCode()); //O(1)
```

```
    Utils.showMessageColor("\nTrajeto: ", AnsiColor.BLUE); //O(1)

    for (int i = 0; i < shortPath.size() - 1; i++) { //O(V)
        Location location = shortPath.get(i); //O(1)
        Location nextLocation = shortPath.get(i + 1); //O(1)
        int weight = gfh.edge(location, nextLocation).getWeight(); //O(1)
        if (chargeLocations.contains(location)) { //O(1)
            if (i == shortPath.size() - 2) //O(1)
                System.out.print(location.getCode() + " (Efetuou carregamento)
--- " + weight + " ---> " + nextLocation.getCode()); //O(1)
            else
                System.out.print(location.getCode() + " (Efetuou carregamento)
--- " + weight + " ---> "); //O(1)
        } else if (i == shortPath.size() - 2) { //O(1)
            System.out.print(location.getCode() + " --- " + weight + " ---> "
+ nextLocation.getCode()); //O(1)
        } else {
            System.out.print(location.getCode() + " --- " + weight + " --->
"); //O(1)
        }


    }
    Utils.showMessageColor("\n\nDistância total: ", AnsiColor.BLUE); //O(1)
    System.out.println(pathLength + "m"); //O(1)
    Utils.showMessageColor("\nNúmero de carregamentos: ", AnsiColor.BLUE);
//O(1)
    System.out.println(numberOfCharges); //O(1)
    Utils.showMessageColor("\nLocais de carregamento: ", AnsiColor.BLUE);
//O(1)
    for (Location location : chargeLocations) { //O(V)
        System.out.println(location.getCode()); //O(1)
    }

    return shortPath; //O(1)
}
```

ANÁLISE COMPLEXIDADE

Ciclo for: O(V)

Logo,

Complexidade do Algoritmo = **O(V)**

****

ALGORITMO

Nome: shortestPath

Input: Um grafo, vértice de origem, vértice de destino, comparator, binary operator, um valor nulo, e uma lista para guardar os locais do caminho.

Output: Uma lista com os locais do caminho.

```
public static <V, E> E shortestPath(Graph<V, E> g, V vOrig, V vDest,
                                    Comparator<E> ce, BinaryOperator<E> sum, E
zero,
                                    LinkedList<V> shortPath) { // O(V^2)

    if (!g.validVertex(vOrig) || !g.validVertex(vDest)) { // O(1)
        return null; // O(1)
    }

    if (vOrig == vDest) { // O(1)
        shortPath.push(vOrig); // O(1)
        return zero; // O(1)
    }

    shortPath.clear(); // O(1)
    boolean[] visited = new boolean[g.numVertices()]; // O(1)


    E[] dist = (E[]) new Object[g.numVertices()]; // O(1)

    V[] pathKeys = (V[]) new Object[g.numVertices()]; // O(1)

    shortestPathDijkstra(g, vOrig, ce, sum, zero, visited, pathKeys, dist); //
O(V^2)

    getPath(g, vOrig, vDest, pathKeys, shortPath); // O(V)

    return shortPath.isEmpty() ? null : dist[g.key(vDest)]; // O(1)
}
```

ANÁLISE COMPLEXIDADE

shortestPathDijkstra(): O(V^2)

Logo,

Complexidade do Algoritmo = **O(V^2)**

****

ALGORITMO

Nome: minDistGraph1

Input: Um grafo, comparator e binary operator.

Output: Um grafo sob a forma de matriz com distâncias mínimas.

```
public static <V, E> MatrixGraph<V, E> minDistGraph1(Graph<V, E> g,
Comparator<E> ce, BinaryOperator<E> sum) { // O(V^3)
    int numVerts = g.numVertices(); // O(1)
    if (numVerts == 0) { // O(1)
        return null; // O(1)
    }

    Graph<V, E> g2 = g.clone(); // O(V^2)

    E[][] edges = (E[][]) new Object[numVerts][numVerts]; // O(1)

    for (int i = 0; i < numVerts; i++) { // O(V^2)
        for (int j = 0; j < numVerts; j++) { // O(V)
            Edge<V,E> edge = g2.edge(i, j); // O(1)
            if (edge != null){ // O(1)
                edges[i][j] = edge.getWeight(); // O(1)
            }
        }
    }

    for (int k = 0; k < numVerts; k++) { // O(V^3)
        for (int i = 0; i < numVerts; i++) { // O(V)
            if (i != k && edges[i][k] != null) { // O(1)
                for (int j = 0; j < numVerts; j++) { // O(V)
                    if (j != i && j != k && edges[k][j] != null) { //
O(1)
                        E s = sum.apply(edges[i][k], edges[k][j]); //
O(1)
                        if ((edges[i][j] == null) ||
ce.compare(edges[i][j], s) > 0) { // O(1)
                            edges[i][j] = s; // O(1)
                        }
                    }
                }
            }
        }
    }

    return new MatrixGraph<>(false, g.vertices(), edges); // O(V^3)
}
```

ANÁLISE COMPLEXIDADE

3 ciclos for: O(V^3)

Logo,

Complexidade do Algoritmo = **O(V^3)**

FUNCIONALIDADE 4

EXERCÍCIO

"Determinar a rede que liga todas as localidades com uma distância total mínima.

Critério de Aceitação: Devolver a rede de ligação mínima: locais, distância entre os locais e distância total da rede."

DESCRIÇÃO

Para esta US foram implementados métodos que utilizam o algoritmo HamiltonianPath para determinar o caminho mais curto que passasse em todos os vértices do grafo. A partir do grafo obtido através dos ficheiros csv, determinou-se os caminhos possíveis em que todos os vértices fossem visitados uma vez e escolhendo o mais curto.

ALGORITMO

Nome: findShortestHamiltonianPath

Input: Grafo

Output: Lista de caminho mais curso e o seu custo.

Neste método percorremos os vértices pertencentes ao grafo, verificamos se o caminho é valido, verificando através do método recursivo shortestHamiltonianPathUtil, que vamos analisar de seguida. Posteriormente o custo é calculado e o menor caminho é retornado.

```java
private USEI04_DTO findShortestHamiltonianPath(Graph<Location, Integer> g) {
    //Get the nr of vertices in the graph
    int numVerts = g.numVertices(); // O(1)

    // If 0, returns a null list
    if (numVerts == 0) {
        return new USEI04_DTO( hasHamiltonianPath: false, new LinkedList<>(),  pathWeight: 0); // O(1)
    }

    // Array to check if the vertices have been visited
    boolean[] visited = new boolean[numVerts]; // O(|V|)

    //Array to save the min path cost
    int[] minPathWeight = { Integer.MAX_VALUE }; // O(1)

    // List to save the shortest path found
    LinkedList<Location> shortestPath = new LinkedList<>(); // O(1)

    // For loop to go thorugh all the vertices in the graph to find a valid path
    for (Location vertex : g.vertices()) { //O(|V|)
        LinkedList<Location> path = new LinkedList<>(); // O(1)
        path.addLast(vertex); // O(1)
        visited[g.key(vertex)] = true; // O(1)

        // If a path is found, calculate its cost
        if (shortestHamiltonianPathUtil(g, visited, path,  pos: 1, minPathWeight)) {
            int pathWeight = calculatePathWeight(g, path);
            if (pathWeight < minPathWeight[0]) {
                minPathWeight[0] = pathWeight; // O(1)
                shortestPath = new LinkedList<>(path); //O(|V|)
            }
        }

        //Resets for next loop
        Arrays.fill(visited,  val: false);
    }

    return new USEI04_DTO(!shortestPath.isEmpty(), shortestPath, minPathWeight[0]);
}
```

ANÁLISE COMPLEXIDADE

1 ciclo for: O(V)

Logo,

Complexidade do Algoritmo = **O(V)**

ALGORITMO

Nome: shortestHamiltonianPathUtil

Input: Grafo, visited, path, pos, minPathWeight

Output: boolean

Este método usa um algoritmo recursivo para encontrar o caminho hamiltoniano mais curto no grafo. O método explora possíveis caminhos no grafo, marcando os vértices visitados e recuando quando um caminho válido não pode ser continuado. Também garante que todos os vértices são visitados, indicando a descoberta de um caminho hamiltoniano.

```
2 usages  1212044
private boolean shortestHamiltonianPathUtil(Graph<Location, Integer> g, boolean[] visited,
                                            LinkedList<Location> path, int pos, int[] minPathWeight) {

    //If all vertices have been visited ends - the path has been found
    if (pos == g.numVertices()) { // O(1)
        return true;
    }

    // Get the last vertice of the current path
    Location lastVertex = path.getLast();

    for (Location v : g.vertices()) { //O(|V|)
        int key = g.key(v);

        //If the vertice is not visited yet, and there is a valid edge, goes thorugh that path
        if (!visited[key]) { // O(1)
            if (pos == 0 || g.edge(lastVertex, v) != null) {
                visited[key] = true;
                path.addLast(v);

                // Recursive to explore the path
                if (shortestHamiltonianPathUtil(g, visited, path,  pos: pos + 1, minPathWeight)) {
                    return true;
                }

                // Goes back if the current path is not valid
                visited[key] = false;
                path.removeLast();
            }
        }
    }
    return false;
}
```

ANÁLISE COMPLEXIDADE

1 ciclo for: O(V)

Logo,

Complexidade do Algoritmo = **O(V)**

ALGORITMO

Nome: calculatePathWeight

Input: Grafo, path

Output: weight

Neste método percorremos o caminho para calcular o seu custo.

ANÁLISE COMPLEXIDADE

```
1 usage   1212044
private int calculatePathWeight(Graph<Location, Integer> g, LinkedList<Location> path) {
    int weight = 0;
    // for loop that goes thorugh the path to calculte it's cost
    for (int i = 0; i < path.size() - 1; i++) { //O(n)
        Location currentLocation = path.get(i);
        Location nextLocation = path.get(i + 1);
        weight += g.edge(currentLocation, nextLocation).getWeight();
    }
    return weight;
}
```

1 ciclo for: O(n)

Logo,

Complexidade do Algoritmo = **O(n)**