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

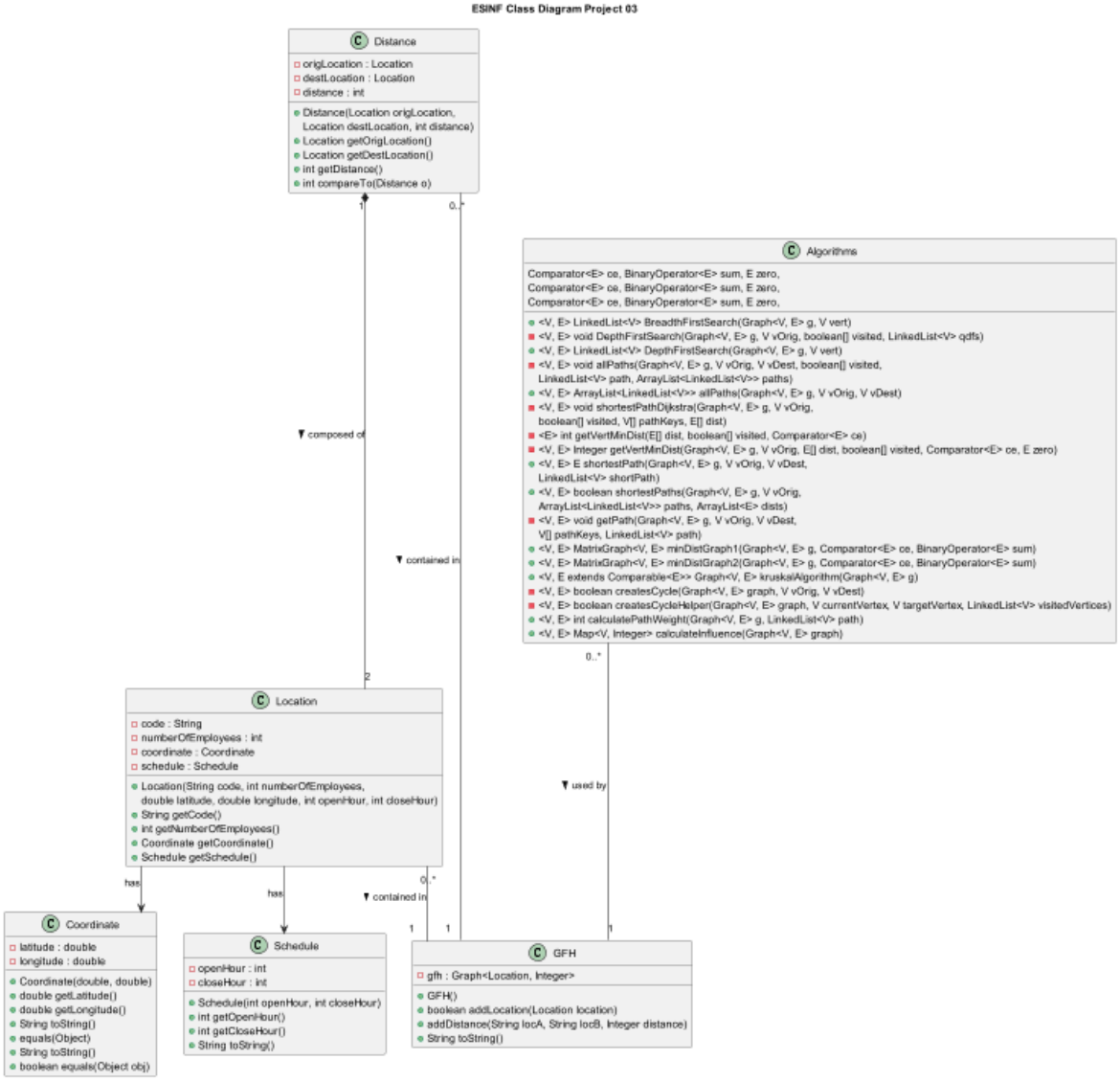
Orientador

Professor Telmo Matos

Porto, 26 de novembro de 2023

ÍNDICE

DIAGRAMA DE CLASSES



FUNCIONALIDADE 6

EXERCÍCIO

"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."

DESCRIÇÃO

De forma a encontrar todos os percursos possíveis entre um local de origem e um hub, recorreu-se ao algoritmo "allPaths" que devolve todos os caminhos possíveis entre dois vértices num grafo. Neste caso, como os percursos estão limitados pelos km's de autonomia, ajustou-se esse algoritmo, passando a chamar-se "allPathsAutonomy" que retira os percursos cuja soma dos pesos das arestas excedem a autonomia máxima do veículo elétrico.

ALGORITMO

Nome: allPathsAutonomy

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

Output: ArrayList de todos os caminhos possíveis (LinkedList's de vértices)

```
public static <V, E> ArrayList<LinkedList<V>>
allPathsAutonomy(Graph<V, E> g, V vOrig, V vDest, int autonomy) { //
O(V*E)
    if (g.numVertices() <= 0) { // O(1)
        return null; // O(1)
    }

    if (autonomy < 0) { // O(1)
        return null; // O(1)
    }

    boolean[] visited = new boolean[g.numVertices()]; // O(1)
    LinkedList<V> path = new LinkedList<>(); // O(1)
    ArrayList<LinkedList<V>> paths = new ArrayList<>(); // O(1)

    allPathsAutonomy(g, vOrig, vDest, visited, path, paths, autonomy);
// O(V*E)

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

```
private static <V, E> void allPathsAutonomy(Graph<V, E> g, V vOrig, V vDest,
boolean[] visited, LinkedList<V> path, ArrayList<LinkedList<V>> paths, double
autonomy) { // O(V*E)

    path.add(vOrig); // O(1)
    visited[g.key(vOrig)] = true; // O(1)

    for (V vAdj : g.adjVertices(vOrig)) { // O(V*E)
        Edge edge = g.edge(vOrig, vAdj); // O(1)
        Integer edgeWeight = (Integer) edge.getWeight(); // O(1)

        if (autonomy >= edgeWeight) { // O(1)
            autonomy -= edgeWeight; // O(1)

            if (vAdj == vDest) { // O(1)
                path.add(vDest); // O(1)
                LinkedList<V> pathCopy = new LinkedList<>(path); // O(V)
                paths.add(pathCopy); // O(1)
                path.removeLast(); // O(1)
            } else {
                if (!visited[g.key(vAdj)]) { // O(1)
                    allPathsAutonomy(g, vAdj, vDest, visited, path, paths,
autonomy); // O(V*E)
                }
            }

            autonomy += edgeWeight; // O(1)
        }
    }
    visited[g.key(vOrig)] = false; // O(1)
    path.removeLast(); // O(1)
}
```

ANÁLISE COMPLEXIDADE

Ciclo For/Corpo do método: O(V)

Chamada recursiva a allPathsAuthonomy: O(E)

Complexidade do algoritmo = **O(V*E)**

****

FUNCIONALIDADE 7

EXERCÍCIO

"Encontrar para um produtor que parte de um local origem o percurso de entrega que maximiza o número de hubs pelo qual passa, tendo em consideração o horário e 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"

DESCRIÇÃO

Para otimizar a identificação do caminho mais eficiente entre locais de origem e hubs, foi implementado um algoritmo baseado em travessia de grafos. O método "findOptimizedPath" utiliza uma abordagem de busca em profundidade (DFS) para determinar o caminho otimizado através do grafo. Além disso, a classe "calculateCentrality" calcula informações de centralidade para cada hub no caminho otimizado, incluindo horários de chegada e partida, distância total, número de carregamentos e tempo total

ALGORITMO

```
private List<Location> findOptimizedPath(Graph<Location, Integer> graph, Location startingLocation) {

    Map<Location, List<Location>> adjacencyList = getAdjacencyList(graph);

    // Initialize a stack to store the path
    Stack<Location> path = new Stack<>(); //O(1)
    path.push(startingLocation);

    List<Location> optimizedPath = new ArrayList<>();

    while (!path.isEmpty()) { //O(n)
        Location current = path.peek();

        if (adjacencyList.get(current) != null && !adjacencyList.get(current).isEmpty()) {
            // If there are remaining edges from the current vertex
            Location next = adjacencyList.get(current).remove( index: 0);
            path.push(next); //O(1)
        } else {
            // If no more edges from the current vertex, add it to the final path
            optimizedPath.add( index: 0, path.pop());//O(1)
        }
    }

    return optimizedPath;
}
```

ANÁLISE COMPLEXIDADE

ciclo while: O(n), Complexidade do Algoritmo = O(V)

FUNCIONALIDADE 8

EXERCÍCIO

"Encontrar para 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 voltar ao local origem minimizando a distância total percorrida."

DESCRIÇÃO

Na funcionalidade 8 tratamos de atribuir o número de funcionários a cada hub. Posteriormente encontramos o caminho hamiltoniano com o menor custo. Depois de verificarmos qual é o caminho com o menor custo através do algoritmo de Hamilton, calculámos o tempo que o circuito iria demorar a ser completado tendo em conta que o veículo se movimento a uma velocidade média de 60Km/h, demora 30min sempre que tem de recarregar e 20min em qualquer paragem para descarga. De ressaltar que a autonomia do veículo é inserida pelo utilizador.

ALGORITMO

```
public USEI08_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 USEI08_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 USEI08_DTO(!shortestPath.isEmpty(), shortestPath, minPathWeight[0]);
}
```

```java
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)

FUNCIONALIDADE 9

EXERCÍCIO

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.

DESCRIÇÃO

O objetivo é o de organizar as localidades de um grafo em N clusters, garantindo a presença de apenas um hub por cluster. A abordagem iterativa adotada utiliza a remoção de ligações com o maior número de caminhos mais curtos entre localidades, resultando na formação de clusters isolados. A função "clusterize" incorpora esse processo iterativo de remoção de arestas para identificar clusters no grafo, enquanto a função exploreCluster utiliza busca em largura (BFS) para explorar e identificar um cluster a partir de um vértice inicial.

ALGORITMO

```
public static <V, E> List<Cluster<V>> clusterize(Graph<V, E> graph) {
    List<Cluster<V>> clusters = new ArrayList<>();

    while (graph.numEdges() > 0) { //O(E)
        Edge<V, Integer> edgeToRemove = findEdgeToRemove((Graph<V, Integer>) graph);
        if (edgeToRemove != null) {
            V vOrig = edgeToRemove.getVOrig();
            V vDest = edgeToRemove.getVDest();
            graph.removeEdge(vOrig, vDest);
        } else {
            break; // No more edges to remove
        }
    }

    // Identify clusters
    Set<V> visitedLocations = new HashSet<>(); // O(1)
    for (V vertex : graph.vertices()) { // O(V)
        if (!visitedLocations.contains(vertex)) {
            Cluster<V> cluster = exploreCluster(graph, vertex); // O(V + E)
            if (cluster != null) {
                clusters.add(cluster);
            }
        }
    }

    return clusters;
}
```

ANÁLISE COMPLEXIDADE

O(V + E)