Instituto Superior de Engenharia do Porto

PROJETO INTEGRADOR SPRINT 2

Estruturas de Informação

Projeto desenvolvido por:

1221219 Diogo Araújo

1221023 João Monteiro

1220780 Tiago Alves

1221003 Tiago Santos

Data: 26/11/2023

Índice

Introdução

Este relatório tem como objetivo analisar a complexidade dos métodos desenvolvidos para as User Stories propostas no Sprint 2 do Projeto Integrador.

O ponto central desta análise incidirá sobre a estrutura de informação que são os grafos, os quais estão associados à distribuição de produtos agrícolas no sistema em desenvolvimento no âmbito do projeto.

Análise de Complexidade

USEI01

Nesta funcionalidade é pretendido desenvolver a infraestrutura de entrega de cabazes utilizando os ficheiros (distâncias xxx.csv e locais xxx.csv) no formato fornecido. A estrutura do grafo deve ser criada utilizando a representação mais apropriada para executar eficientemente as operações desejadas.

Dependendo da necessidade das funcionalidades será utilizado um método diferente para criar a estrutura de dados, um grafo de matrizes ou um grafo de mapas. Ambos os métodos eventualmente chamam a mesma função “fillGraph”. Quando utilizada uma matriz, o método “addVertex” irá se apresentar com uma complexidade de v^2 sendo que v representa o número de vértices. Este método executado V vezes podemos então concluir que esta funcionalidade terá uma complexidade v^3 no pior caso.

```
private static void fillGraph(Graph<Node, Weight> graph, String edges_file,
String vertices_file) {
   ArrayList<String[]> data_vertices = Utils.readCSV(vertices_file);
   ArrayList<String[]> data_edges = Utils.readCSV(edges_file);

   for (String[] s : data_vertices)
       graph.addVertex(new Node(s[0], new Coords(Utils.toDouble(s[1]),
Utils.toDouble(s[2]))));

   Node vOrig = null;
   Node vDest = null;

   for (String[] s : data_edges) {
       for (Node v : graph.vertices()) {
           if (v.getLabel().equals(s[0]))
               vOrig = v;
           if (v.getLabel().equals(s[1]))
               vDest = v;
```

```
        }
        Weight weight = new Weight(Utils.toInt(s[2]), "m");
        graph.addEdge(vOrig, vDest, weight);
    }
}
```

USEI02

Identificar os vértices ideais para posicionar N centros de distribuição a fim de otimizar a rede de distribuição com base em diferentes critérios:

- Influência: Vértices com maior grau.
- Proximidade: Vértices mais próximos dos demais.
- Centralidade: Vértices com o maior número de caminhos mínimos que passam por eles.

Critério de Aceitação: Apresentar todas as localidades e seus critérios associados, ordenados de forma decrescente com base na centralidade e influência.

Esta funcionalidade utiliza o método “hubsLocation”, a complexidade deste é principalmente determinada pelos dois ciclos “for” pelos quais irá iterar, percorrendo todos os vértices para cada vértice, resultando destes uma complexidade v^2. A somar-se a esta situação o método “shortestPath” que utiliza o algoritmo de Dijkstra que tem complexidade $(v + e) * log(v)$.

Para além deste, ainda é utilizado um outro bloco “for” que no pior caso será percorrido V vezes. Assim sendo, a complexidade geral desta funcionalidade apresenta-se como $v^3 * (v + e) * log(v)$.

```
  public static List<HubStatus<Node>> hubsLocation(Graph<Node, Weight>
graph) {

        List<HubStatus<Node>> listStatus = new ArrayList<>();

        LinkedList<Node> res;
        Weight zero = new Weight(0, "m");

        int[] influence = new int[graph.numVertices()];
        int[] proximity = new int[graph.numVertices()];
        int[] centrality = new int[graph.numVertices()];
```

```
        for (Node n1: graph.vertices()) {
            for (Node n2: graph.vertices()) {
                if (n1.equals(n2))
                    continue;
                res = new LinkedList<>();
                Algorithms.shortestPath(graph.clone(), n1, n2,
Weight::compare, Weight::sum, zero, res);

                if (res.size() == 0)
                    return new ArrayList<>();

                for (int i = 0; i < res.size() - 1; i++) {
                    centrality[graph.key(res.get(i))]++;
                    proximity[graph.key(n1)] += graph.edge(res.get(i),
res.get(i + 1)).getWeight().getDistance();
                }
                centrality[graph.key(res.get(res.size() - 1))]++;
            }
            influence[graph.key(n1)] = graph.outDegree(n1);
        }

        for (int i = 0; i < graph.numVertices(); i++)
            listStatus.add(new HubStatus<>(graph.vertex(i), influence[i],
proximity[i], centrality[i]));

        listStatus.sort(new InfluenceComparator());
        listStatus.sort(new CentralityComparator());
        Collections.reverse(listStatus);

        return listStatus;
    }
}
```

USEI03

Dado um veículo e a sua autonomia e considerando que os carregamentos só podem ocorrer em localidades específicas, é preciso calcular a rota mais curta entre os dois pontos mais distantes na rede de distribuição.

Isto envolve identificar o menor caminho possível e indicar quantas paradas serão necessárias para recarregar o veículo.

Critério de Aceitação: Fornecer a rota entre os dois pontos mais distantes na rede de distribuição, incluindo o local de origem, os pontos de passagem (com indicação dos locais onde ocorreu o carregamento do veículo), a distância entre os locais do percurso, o destino, a distância total percorrida e o número total de carregamentos.

A funcionalidade autonomyCheck tem a sua complexidade principalmente determinada pela utilização do método "minDistGraph" sendo essa v^3. Para além deste, como no exercício anterior, está presente o método "shortestPath" e a complexidade deste é $(v + e) * log(v)$. Posto isto concluimos que a complexidade final será o resultante da soma destas complexidades, $v^3 + (v + e) * log(v)$.

```
public static Route<Node> autonomyCheck(Graph<Node, Weight> graph, int
autonomy) {

    Graph<Node, Weight> graphMinDist = Algorithms.minDistGraph(graph,
Weight::compare, Weight::sum);

    Weight nullWeight = new Weight(0, "m");
    Weight maxWeight = nullWeight;
    Node vOrig = null;
    Node vDest = null;

    for (Edge<Node, Weight> e : graphMinDist.edges()) {
        if (Weight.compare(e.getWeight(), maxWeight) == 1) {
            maxWeight = e.getWeight();
            vOrig = e.getVOrig();
            vDest = e.getVDest();
        }
    }

    Graph<Node, Weight> copy_graph = graph.clone();

    for (Edge<Node, Weight> edge : copy_graph.edges()) {
        if (Weight.compare(edge.getWeight(), new Weight(autonomy, "m"))
== 1)
            copy_graph.removeEdge(edge.getVOrig(), edge.getVDest());
    }
```

```java
        LinkedList<Node> res = new LinkedList<>();
        Algorithms.shortestPath(copy_graph, vOrig, vDest, Weight::compare,
Weight::sum, nullWeight, res);

        if (res.size() == 0)
            return new Route<>(null, 0, 0);

        LinkedList<POI<Node>> poiList = new LinkedList<>();
        int totalChargingTimes = 0;
        int totalDistance = 0;
        int autonomyNow = autonomy;
        int distance_edge;
        boolean charged;

        for (int i = 0; i < res.size() - 1; i++) {
            distance_edge = copy_graph.edge(res.get(i), res.get(i +
1)).getWeight().getDistance();
            totalDistance += distance_edge;
            charged = false;

            if (autonomyNow < distance_edge) {
                totalChargingTimes++;
                autonomyNow = autonomy;
                charged = true;
            }
            autonomyNow -= distance_edge;
            poiList.add(new POI<>(res.get(i), charged, distance_edge));
        }
        poiList.add(new POI<>(res.get(res.size() - 1), false, 0));

        return new Route<>(poiList, totalChargingTimes, totalDistance);
    }
```

USEI04

Encontrar a rede que conecta todas as localidades com a menor distância total.

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

Para tal aplicamos o método “Kruskal” que recebe um grafo, criado na US01, um comparador de custos e um grafo vazio que será a árvore geradora mínima retornada.

```java
public static MST<Node, Weight> minCostNetwork(Graph<Node, Weight> graph) {

    Graph<Node, Weight> mst = new MatrixGraph<>(false);

    Algorithms.kruskal(graph, Weight::compare, mst);

    int totalDist = 0;
    for (Edge<Node, Weight> e: mst.edges())
        totalDist += e.getWeight().getDistance();

    return new MST<>(mst, totalDist);
}
```

A complexidade mais relevante nesta US encontra-se neste mesmo método. Começamos com um loop que itera todos os vértices presentes no grafo inicial (complexidade O(E)). De seguida serão iterados todos os ramos presentes nesse mesmo grafo por ordem de custo (O(E)), mas para cada iteração obteremos uma lista que provém do método “DepthFirstSearch” (O(V+E)). Sendo assim teremos que a complexidade do método de “Kurskal” é O(E*(V+E))