

PROJETO INTEGRADOR SPRINT 3

Estruturas de Informação Projeto desenvolvido por:

1221219 Diogo Araújo

1221023 João Monteiro

1220780 Tiago Alves

1221003 Tiago Santos

Data: 03/01/2024

Conteúdo

[Introdução 3](#_Toc155217290)

[Análise de Complexidade 3](#_Toc155217291)

[USEI06 3](#_Toc155217292)

[USEI07 5](#_Toc155217293)

[USEI08 7](#_Toc155217294)

[USEI09 9](#_Toc155217295)

# Introdução

O intuito deste relatório, no âmbito da disciplina de Estruturas de Informação, é explicitar o estudo da complexidade dos métodos desenvolvidos para as User Stories propostas no Sprint 3 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

## USEI06

Nesta funcionalidade é pretendido 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. Devem ser devolvidos os locais de passagem, distância entre todos os locais do percurso, distância total e tempo total de percurso.

public static ArrayList<Route<Node>> routesToHub (String originNodeLabel, String destinationHubLabel, float autonomy, int averageSpeed, Graph<Node, Weight> graph){

É inicializado o método com as variáveis necessárias, o vértice de origem e o destino, uma autonomia e uma velocidade média assim como o grafo em causa.

if(autonomy <= 0 || averageSpeed <= 0 || originNodeLabel == null || destinationHubLabel == null || graph == null)  
 return null;

Aqui fazemos verificações para eventuais atributos nulos.  
  
 Node originNode = Utils.*getNodeFromLabel*(graph, originNodeLabel);  
 Node destinationNode = Utils.*getNodeFromLabel*(graph,destinationHubLabel);  
  
 return *allPaths*(graph,originNode,destinationNode,autonomy,averageSpeed);   
}

Este método retorna todos os caminhos entre os dois vértices limitados pela autonomia sendo que a complexidade desta funcionalidade é principalmente determinada pela que é chamada na linha do return, “allPaths”.

public static ArrayList<Route<Node>> allPaths(Graph<Node, Weight> g, Node vOrig, Node vDest,float autonomy,int averageSpeed) {  
 LinkedList<POI<Node>> path = new LinkedList<>();  
 ArrayList<Route<Node>> paths = new ArrayList<>();  
  
 boolean[] visited = new boolean[g.numVertices()];  
  
 if (g.validVertex(vOrig) && g.validVertex(vDest))  
 *allPaths*(g, vOrig, vDest,autonomy, visited, path, paths,0,0,averageSpeed);  
 return paths;  
}

Neste método é verificado se os vértices existem no grafo e se tal for verdade, chamará um método allPaths que calcula todos os caminhos entre o vértice inicial e final.

private static void allPaths(Graph<Node, Weight> g, Node vOrig, Node vDest,float autonomy, boolean[] visited, LinkedList<POI<Node>> path,  
 ArrayList<Route<Node>> paths, int totalDistance,int distanceToNextPOI,int averageSpeed) {  
  
 int vKey = g.key(vOrig);  
  
 if (visited[vKey])  
 return;  
  
 if (vOrig.equals(vDest)) {  
 LinkedList<POI<Node>> pathCopy = new LinkedList<>(path);  
  
 pathCopy.addFirst(new POI<>(vDest,false,distanceToNextPOI));  
  
 Collections.*reverse*(pathCopy);  
  
 float totalTime = (totalDistance / (float)(averageSpeed \* 1000));  
 paths.add(new Route<>(pathCopy,totalDistance,totalTime) );  
 return;  
 }

Verifica se o vértice já foi visitado, se é o destino, cria um caminho invertido e o adiciona aos resultados.

path.push(new POI<>(vOrig,false,distanceToNextPOI));  
  
 visited[vKey] = true;  
  
 for (Node vAdj : g.adjVertices(vOrig)) {  
 Edge<Node, Weight> edge = g.edge(vOrig, vAdj);  
 Weight w = edge.getWeight();  
 if (autonomy > w.getDistance()) {  
 *allPaths*(g, vAdj, vDest, (autonomy - w.getDistance()), visited, path, paths,(totalDistance += w.getDistance()),w.getDistance(),averageSpeed);  
 } else return;  
 }  
  
 path.pop();  
  
 visited[vKey] = false;  
}

Com isto tudo em conta, e considerando V como o número de vértices e E como o número de arestas no grafo, a complexidade final pode ser representada por O (V + E).

## USEI07

A tarefa consiste em determinar a rota de entrega a partir do local de origem para um produtor otimizando o número de hubs visitador. Nesse processo, é necessário levar em conta o horário de funcionamento de cada hub, o tempo de descarga nos hubs, as distâncias a serem percorridas assim como a velocidade média e os tempos de carregamento do veículo. O intuito é encontrar a trajetória mais eficiente que maximize a eficácia da entrega, considerando todas as variáveis referidas.

public static MostHubsPath<Node> maxHubs(Graph<Node, Weight> graph, Node vOrig, LocalTime time, int velocity, int autonomy, int chargerTime, int unloadTime) {  
 ArrayList<LinkedList<Node>> allPaths = new ArrayList<>();  
 LinkedList<Node> path = new LinkedList<>();  
 LinkedList<POI\_2<Node>> list = new LinkedList<>();  
  
 if (velocity < 0 || autonomy < 0 || graph == null || vOrig == null)  
 return null;

É inicializado o método com as variáveis necessárias, sendo estas o grafo em estudo, o vértice de origem do caminho, o tempo inicial do caminho, a velocidade do veículo, a sua autonomia, assim como o tempo de carga do veículo e descarga do tempo de descarga dos produtos nos hubs.

for (Edge<Node, Weight> e : graph.edges())  
 if (e.getWeight().getDistance() > autonomy)  
 graph.removeEdge(e.getVOrig(), e.getVDest());

De seguida retiramos todas as arestas com peso maior do que a autonomia uma vez que o veículo não possui capacidade para efetuar o trajeto.

Complexidade de O (E), sendo E as arestas do grafo.

for (Node n1 : graph.vertices())  
 if (!n1.equals(vOrig))  
 allPaths.addAll(Algorithms.*allPaths*(graph, vOrig, n1));

De seguida obtemos os todos os caminhos possíveis do vértice de origem para todos os outros.

Complexidade de O (V^2 \* E)

int nHubs;  
int maxHubs = 0;  
int numVertex = 0;  
  
for (LinkedList<Node> l : allPaths) {  
  
 if ((nHubs = Utils.*getNumOfHubs*(graph, l, time, velocity, autonomy, chargerTime, unloadTime)) > maxHubs && numVertex < l.size()) {  
  
 numVertex = l.size();  
 maxHubs = nHubs;  
 path = l;  
 }  
}

Depois, todos esses caminhos são percorridos de modo a obter o primeiro caminho que possua o maior número de hubs e o menor número de vértices.

Complexidade de O (V^2)

int totalDist = 0;  
int chargingTimes = 0;  
  
int chargingTimeTotal = 0;  
int pathTimeTotal = 0;  
int unloadTimeTotal = 0;  
  
int autonomyNow = autonomy;  
int vel = Utils.*convertKMHtoMM*(velocity);  
int totalTime = 0;  
  
Node n1;  
Node n2 = null;  
POI\_2<Node> poi;

for (int i = 0; i < path.size() - 1; i++) {  
 n1 = path.get(i);  
 n2 = path.get(i + 1);  
  
 int dist = graph.edge(n1, n2).getWeight().getDistance();  
  
 int minTotal = dist / vel;  
  
 totalTime += minTotal;  
 time = time.plusMinutes(minTotal);  
 pathTimeTotal += minTotal;  
  
 poi = new POI\_2<>(n1, n1.getIsHub(), time, time);  
  
  
 if (autonomyNow < dist) {  
 autonomyNow = autonomy;  
 totalTime += chargerTime;  
 chargingTimeTotal += chargerTime;  
 time = time.plusMinutes(chargerTime);  
 chargingTimes++;  
 }  
  
 autonomyNow -= dist;  
 totalDist += dist;  
  
 if (n1.getIsHub() && n1.isOpen(time)) {  
 totalTime += unloadTime;  
 unloadTimeTotal += unloadTime;  
 time = time.plusMinutes(unloadTime);  
 poi.setDepartureTime(time);  
 }  
 list.add(poi);  
}  
  
poi = new POI\_2<>(n2, n2.getIsHub(), time, time);

Por fim, analisamos o caminho obtido e percorremo-lo de modo a obter todos os valores necessários.

Complexidade de O (V)

poi = new POI\_2<>(n2, n2.getIsHub(), time, time);  
  
 if (n2.getIsHub() && n2.isOpen(time)) {  
 totalTime += unloadTime;  
 unloadTimeTotal += unloadTime;  
 time = time.plusMinutes(unloadTime);  
 poi.setDepartureTime(time);  
 }  
 list.add(poi);  
  
 return new MostHubsPath<>(list, totalDist, chargingTimes, totalTime, chargingTimeTotal, pathTimeTotal, unloadTimeTotal);  
}

Com isto tudo em conta, e considerando V como o número de vértices e E como o número de arestas no grafo, a complexidade final pode ser representada por O (V^2 \* E).

## USEI08

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

Critério de Aceitação: Devolver o local de origem do circuito, os locais de passagem, a distância entre todos os locais do percurso, a distância total, o número de carregamentos e o tempo total do circuito.

Este for loop retira as arestas para reduzir a eficiência geral do algoritmo.

*for* (Edge<Node, Weight> e : graph.edges())  
 *if* (e.getWeight().getDistance() > autonomy)  
 graph.removeEdge(e.getVOrig(), e.getVDest());

Complexidade de O (E)

Este loop calcula todos os caminhos desde o ponto de origem ate todos os N hubs.

*for* (Node n1 : hubList)  
 allPaths.addAll(Algorithms.allPaths(graph, vOrig, n1));  
Complexidade de O (n \* paths)

Este loop seleciona apenas os caminhos que contem todos os hubs na lista de N hubs

*for* (LinkedList<Node> allPath : allPaths) {  
 *if* (containsList(allPath, hubList))  
 newList.add(allPath);  
}

Complexidade de O (paths)

O seguinte codigo tem como funcao encontrar qual dos caminhos, o mais curto, de modo a ser o circuito mais curto.

*int* distance;  
*int* minDist = Integer.MAX\_VALUE;  
  
*for* (*int* i = 0; i < newList.size(); i++) {  
 *if* ((distance = getTotalDist(graph, newList.get(i))) < minDist) {  
 minDist = distance;  
 path = allPaths.get(i);  
 }  
}  
Complexidade de O (paths \* paths.size)

E utilizado o shortest path para calcular o caminho de volta a origem.

LinkedList<Node> list2 = *new* LinkedList<>();  
  
Algorithms.shortestPath(graph, path.get(path.size() - 1), vOrig, Weight::compare, Weight::sum, *new* Weight(0, "m"), list2);

E necessario remover o primeiro node do caminho de volta para não haver repeticoes

list2.remove(0);  
  
path.addAll(list2);

Esta funcao tem como objetivo calcular a distancia percorrida por um caminho.

*private static int* getTotalDist(Graph<Node, Weight> graph, LinkedList<Node> nodes) {  
 *int* sum = 0;  
  
 Node n1;  
 Node n2;  
  
 *for* (*int* i = 0; i < nodes.size() - 1; i++) {  
 n1 = nodes.get(i);  
 n2 = nodes.get(i + 1);  
  
 sum += graph.edge(n1, n2).getWeight().getDistance();  
 }  
 *return* sum;

}  
 Complexidade de O (nodes.size)

Esta funcao tem como objetivo verificar se uma lista contem a outra.  
 *private static boolean* containsList(LinkedList<Node> nodes, LinkedList<Node> hubList) {  
 *int* i = 0;  
  
 *for* (Node n : nodes)  
 *if* (hubList.contains(n))  
 i++;  
 *return* i == hubList.size();  
 }

Complexidade de O (nodes.size \* hubList.size)  
Esta funcao tem como objetivo retornar os N hubs com mais colaboradores.  
 *private static* LinkedList<Node> getNhubs(Graph<Node, Weight> graph, *int* n) {  
 ArrayList<Node> list = graph.vertices();  
 Collections.sort(list, *new* LabelComparator());  
  
 LinkedList<Node> finalList = *new* LinkedList<>();  
  
 *for* (*int* i = 0; i < n; i++)  
 finalList.add(list.get(i));  
  
 *return* finalList;  
 }  
}

Complexidade de O (V log V + n)

## USEI09

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.

Critério de Aceitação: Devolver lista de Hubs e o respetivo conjunto de localidades que pertencem ao cluster desse Hub.

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.

Critério de Aceitação: Devolver lista de Hubs e o respetivo conjunto de localidades que pertencem ao cluster desse Hub.

Para tal primeiro vamos organizar num mapa todas as arestas e as suas respetivas vezes que são repetidas ao traçar caminhos mínimos entre todos os vértices do grafo na função “contarNumeroArestasMinimas”.

public static void contarNumeroArestasMinimas (Graph<Node, Weight> grafo,ArrayList<LinkedList<Node>> paths,HashMap<Edge<Node, Weight>, Integer> numeroDeArestasRepetidas){

ArrayList<Weight> dists = new ArrayList<>();

for (int k = 0; k < grafo.numVertices(); k++) {

Algorithms.*shortestPaths*(grafo, grafo.vertex(k), Weight::*compare*, Weight::*sum*, new Weight(0,"m"), paths, dists);

for (LinkedList<Node> path : paths) {

for (int j = 0; j < path.size() - 1; j++) {

Node origem = path.get(j);

Node destino = path.get(j + 1);

Edge<Node, Weight> chaveAresta;

if (origem.getLabel().compareTo(destino.getLabel()) < 0) {

chaveAresta = new Edge(origem, destino, 0);

} else {

chaveAresta = new Edge(destino, origem, 0);

}

Weight w = grafo.edge(origem, destino).getWeight();

chaveAresta.setWeight(w);

numeroDeArestasRepetidas.put(chaveAresta, numeroDeArestasRepetidas.getOrDefault(chaveAresta, 0) + 1);

}

}

}

}

De seguida vamos cortando as arestas mais repetidas e a cada vez que tal acontece verificamos se conseguimos criar um cluster, se conseguirmos cria-se um cluster e adicionamos á lista de clusters com suas respetivas localidades.

public static List<Cluster> organizarClusters(Graph<Node, Weight> grafo, List<HubStatus<Node>> hubs) {

ArrayList<LinkedList<Node>> paths = new ArrayList<>();

HashMap<Edge<Node, Weight>, Integer> numeroDeArestasRepetidas = new HashMap<>();

*contarNumeroArestasMinimas*(grafo,paths,numeroDeArestasRepetidas);

if (hubs.size() == 1){

grafo.removeVertex(hubs.get(0).getElement());

*clusters*.add(new Cluster(hubs.get(0),grafo.vertices()));

}

while (!numeroDeArestasRepetidas.isEmpty() || *numeroDeClustersCriados* != hubs.size()) {

Edge<Node, Weight> arestaMaxima = *calcularArestaMaxima*(numeroDeArestasRepetidas);

grafo.removeEdge(arestaMaxima.getVOrig(), arestaMaxima.getVDest());

if (!*isConnected*(grafo)) {

List<Graph<Node,Weight>> componentesConexos = *dividirEmComponentesConexos*(grafo);

*verificacoes*(hubs,grafo,arestaMaxima,componentesConexos,numeroDeArestasRepetidas);

}

numeroDeArestasRepetidas.remove(arestaMaxima);

}

return *clusters*;

}

Nesta user storie a função que causa a maior complexidade é a “contarNumeroArestasMinimas” já mostrada a cima visto que vamos calcular todos os caminhos entre todos os vértices do grafo e contar quantas vezes cada aresta aparece.

Contamos com 3 ciclos for sendo que no primeiro chamamos o método shortestPaths cuja complexidade é O(V^2) ficando assim a complexidade final O(V^5).