# PROJETO INTEGRADOR

2DA
1211016 - Joao Rodrigues (2DB)
1211061 - Gustavo Jorge
1211063 - Joao Leitao
1211073 - Guilherme Sousa
1211076 - Pedro Monteiro

isep
Instituto Superior de Engenharia do Porto
DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

# Conteúdo

# Introdução

No âmbito do Projeto Integrador, a equipa tem como objetivo desenvolver uma solução informática que apoie a gestão de uma empresa responsável pela gestão de uma instalação agrícola em modo de produção biológico. Existindo um conjunto de aspetos necessários para a gestão de uma exploração agrícola em MPB, que serão desenvolvidos ao longo de 2 Sprints, seguindo um processo de desenvolvimento iterativo e incremental, seguindo a metodologia Ágil. Toda esta solução é composta por três aplicações, uma em PL/SQL, outra em C/Assembly e outra em Java.

Especificamente para a Unidade Curricular de ESINF, a aplicação desenvolvida é em Java. Para o Sprint 1, é requerida a realização de um conjunto de 5 user stories. Todas estas vão ser devidamente explicadas e exploradas ao longo deste relatório.

Como objetivo principal do desenvolvimento desta aplicação temos a criação de algo que permita gerir uma **rede de distribuição de cabazes entre agricultores e clientes.** Nesta rede temos um fluxo de dados no qual os **Produtores** disponibilizam diariamente à rede de distribuição os **produtos** e respetivas quantidades que têm para vender e os **Clientes** (**Particulares** ou **Empresas**) colocam encomendas (**cabazes** de produtos agrícolas) à **rede de distribuição.** A **rede** gere distribuição dos **produtos** dos **produtores** de modo a satisfazer os **cabazes** a serem entregues em **Hubs** para posterior levantamento pelos clientes. Um **Hub** é localizado numa **Empresa** e cada **Cliente** (**Particular** ou **Empresa**) recolhe as suas encomendas no **Hub** mais próximo.

De forma a garantir que todos os objetivos eram cumpridos, de forma eficiente, a equipa desenvolveu uma série de algoritmos e recorreu a diversas classes, como tal, é pertinente fazer uma análise dos algoritmos e métodos utilizados de forma a refletir sobre a eficiência dos mesmos. Esta análise está presente ao longo de todo o documento associado à explicação dos diferentes métodos, no entanto, gostávamos de realçar a análise de dois algoritmos principais utilizados pela equipa em diferentes momentos. Por esse motivo, passamos agora à análise dos Algoritmos de Dijkstra e Kruskal e posteriormente à análise de cada funcionalidade em si.

# Funcionamento da Aplicação

## Como utilizar a aplicação – Passo a passo

De forma a cumprir todos os requisitos propostos, desenvolvemos uma aplicação.

Esta pode ser utilizada através da sua Interface Gráfica, no qual tudo é mais bem trabalhado e as funcionalidades apresentam um carácter mais "User Friendly". Existe também um modo de consola, que deve apenas ser utilizado como um "backup", no caso de algo de errado se passar com a Interface gráfica ou com alguma funcionalidade em específico.

Começamos então por fazer um pseudo tutorial passo a passo de como utilizar a aplicação através da interface gráfica.

Se por acaso errar em algum passo, ou fechar algo por engano a melhor solução é repetir o processo.

### 1º Passo – Extrair o ZIP:

Proceder à extração do ficheiro .zip com todo o conteúdo do projeto.

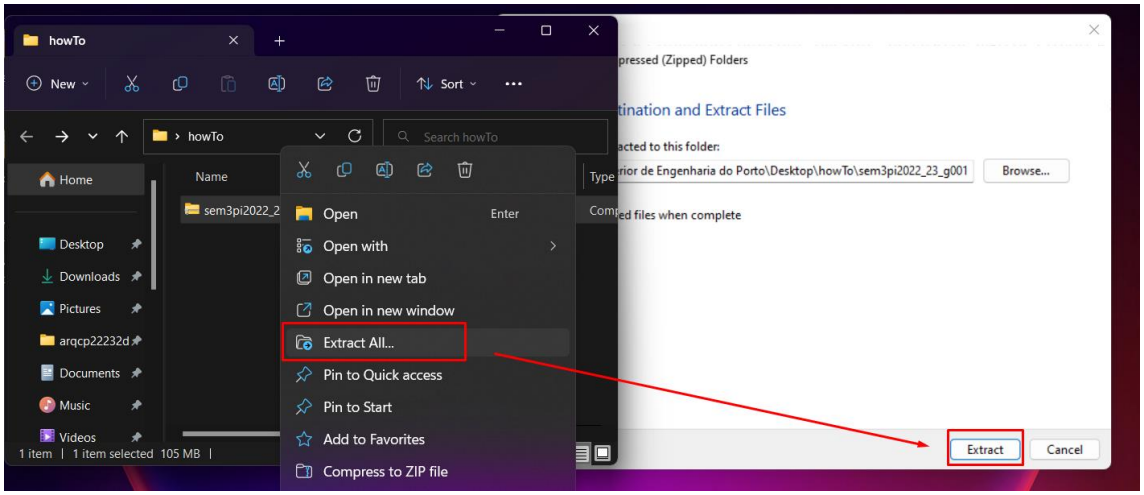


*Figura 1 – Demonstração da Extração o zip*

### 2º Passo – Abrir o projeto:

Utilizando a ferramenta **IntelliJ IDEA,** clicar em **File** no canto superior esquerdo -> **Open** -> Escolher a pasta do projeto. É importante que escolha a pasta **sem3pi2022_23_g001** que se encontra **dentro** de outra com o nome igual, isto pode alterar o funcionamento da aplicação. Caso exista apenas uma pasta com o nome apresentado basta escolher essa.

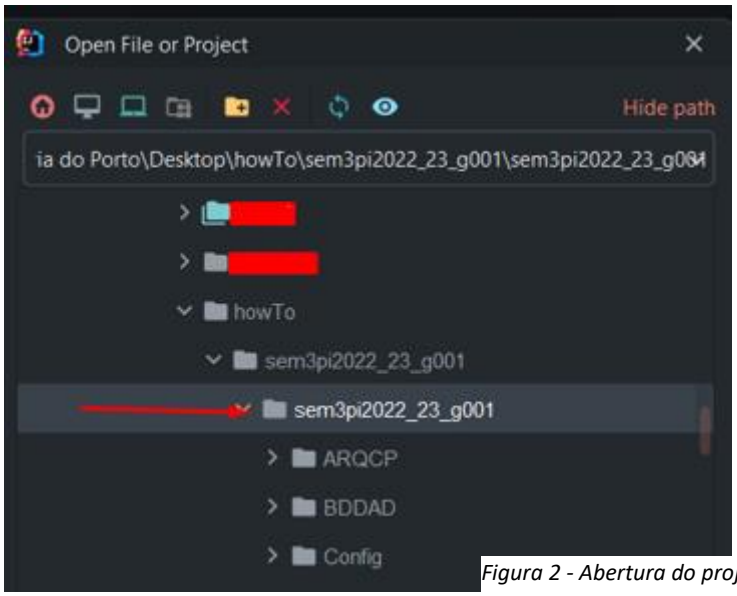


*Figura 2 - Abertura do projeto*

## 3º Passo – "Ativar" o projeto e as suas configurações:

Estamos quase lá… De forma a automatizar todo o processo de configuração é apenas necessário clicar no botão "**Load**". No caso de ter fechado o popUp apresentado na imagem, terá de repetir todo o processo ou procurar pelo ficheiro pom.xml e ativar o maven.

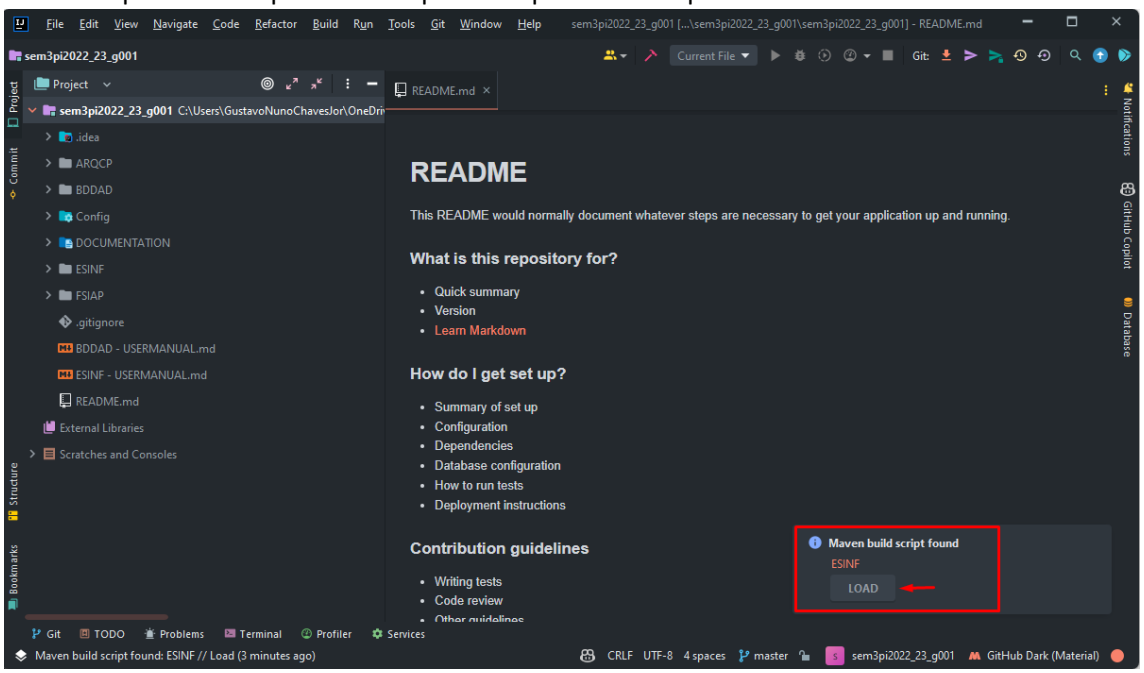


*Figura 3 – Configuração do Projeto*

## 4º Passo – Correr a aplicação:

Abrir a pastas **ESINF** -> **src** -> **main** -> **java** e clicar com o botão do lado direito no ficheiro ***RunApplication*** e escolher a opção **Run**.

** É natural que seja lançado um WARNING pela consola do Intellij, no entanto isso não se trata de nada problemático e deve ser ignorado.

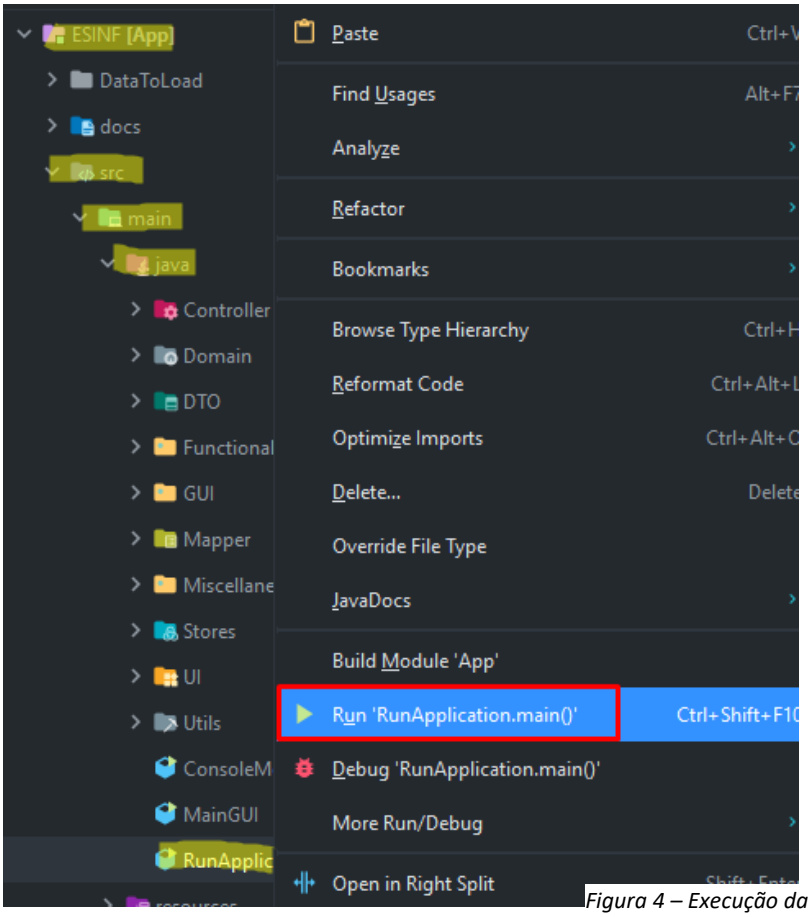


*Figura 4 – Execução da aplicação*

## 5º Passo – Login:

Trata-se de um requisito não funcional do projeto integrador ter um login, daí quando corre a aplicação será necessário efetuá-lo.
Para poder testar todas as funcionalidades no âmbito de ESINF, basta logar com as seguintes credenciais:

**Username: EsinfUser**
**Password: esinfPwd**

## 6º Passo – Utilização da aplicação:

De forma a assegurar uma experiência cómoda e simples do utilizador, inicialmente é pedido ao mesmo que carregue algum tipo de dados para poder prosseguir. Existe sempre a possibilidade de carregar dados **automaticamente**, de forma a facilitar o processo.

Ainda com vista a simplicidade e compreensão de todas as funcionalidades, será possível encontrar, diversas vezes, ao longo da aplicação desenvolvida o símbolo de um **olho**, este trata-se de um botão que tem sempre o objetivo de providenciar mais informação sobre aquilo a que se encontra associado.

## Em caso de problema:

Se por acaso a partir do passo nº 4 ou durante a utilização da aplicação ocorrer algum problema inesperado, existe a opção de correr o programa através do modo consola, para tal basta correr a classe ***ConsoleMode***.

Qualquer outro tipo de questão ou problema, basta enviar um email a qualquer um dos membros da equipa.

## US301/US307 – Construir a rede de distribuição de cabazes

Esta funcionalidade consiste na construção automática da rede de distribuição, através de informação fornecida pelos ficheiros. Para isto, foi implementada uma classe ***Distribution Network*** que implementa a interface ***Graph*** e utiliza como representação de um grafo, a classe ***MapGraph***, simulando assim uma estrutura de mapa de adjacências, no qual cada vértice (vertex) tem associado a si um mapa de arestas (edge). Um grafo pode ser direcionado ou não direcionado, sendo que esta propriedade determina se a direção de uma aresta é ou não relevante, neste caso específico, trata-se de um grafo não direcionado.

De forma a tornar toda esta implementação possível, foi necessário que cada Cliente (Particular ou Empresa) e cada Produtor fosse representado como um vértice do grafo, e a ligação entre dois deles uma aresta. Para isto criamos a classe ***Participant*** para ser utilizada como superclasse, ou seja, de forma que qualquer participante da rede, seja ele um Cliente ou um Produtor, possa ser representado como um vértice no grafo. Para demonstrar a relação acima descrita aqui está um diagrama muito simples que a representa e aprofunda.

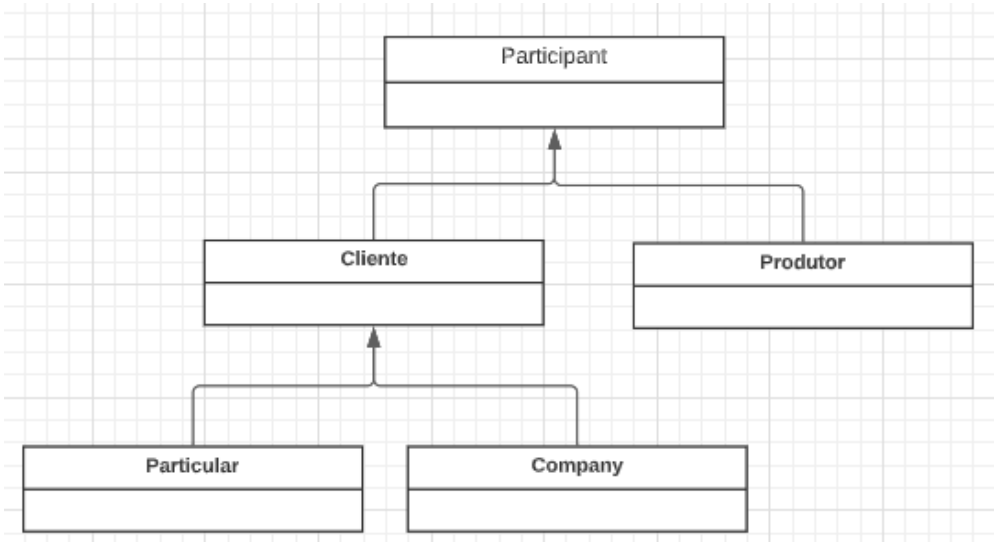


*Figura 1 – Representação da hierarquia de participantes*

Com isto, a estrutura principal da rede fica assegurada, no entanto, de forma a ser possível associar cabazes à rede, foi criada uma classe ***DistributionNetworkData***. Nesta existe uma instância de uma ***DistributionNetwork***, bem como duas listas, uma de produtos (***Products***) disponíveis na mesma, bem como uma de encomendas (***BasketOrder***). Com esta classe já é possível representar toda a rede de distribuição. No entanto, faltam-nos formas de a gerir, bem como, de preencher.

De forma a passar da representação à própria construção da rede, é disponibilizada ao utilizador a opção de carregar toda a informação referente à rede através de ficheiros.

## Carregar a informação que constrói o grafo

A informação de um grafo encontra-se dividida em dois ficheiros, um ficheiro com todos os participantes, e um ficheiro com a informação referente às distâncias entre os participantes.

No primeiro ficheiro encontra-se então os participantes, identificados pelo seu ID e caracterizados pelo ID da sua localização bem como as coordenadas desta, no segundo encontram-se, por linha, dois ID's de duas localizações diferentes e a distância entre ambos.

Para cada linha de um ficheiro que contém os participantes é criado um Participante, com a sua devida informação e por fim, se o mesmo já não estiver presente na ***DistributionNetwork***, é adicionado à mesma.

Já para cada linha de um ficheiro com distâncias, dado que um Participante tem associado a si um ID de uma localização, é possível associar uma distância, que serve de aresta no grafo, entre os dois.

## Carregar os produtos e cabazes da rede

A informação referente aos produtos e cabazes da rede encontra-se toda no mesmo ficheiro, no qual, para cada linha, é possível encontrar um ID de um participante bem como várias colunas com quantidades referentes a cada produto. Cada produto é identificado pelo seu nome, que consta no header do ficheiro

## Análise da complexidade temporal dos métodos utilizados

➔ Ficheiro de participantes (vértices)

O método utilizado para a leitura de participantes trata-se de um método determinístico, uma vez que a sua complexidade temporal não varia dependendo do tipo de input. Passamos agora a uma análise mais detalhada do método:

| | Line Complexity |
|---|---|
| `public boolean readFromFile(String path) {` | |
| `BufferedReader reader = new BufferedReader(new FileReader(path));` | O(1) |
| `String line = reader.readLine();` | O(1) |
| `while ((line = reader.readLine()) != null) {` | O(V) |
| `String[] participantInfo = line.split(",");` | O(1) |
| `if (participantInfo.length != 4) {` | O(1) |
| `return false;}` | O(1) |
| `Coordinates coordinates = new Coordinates(Double.parseDouble(participantInfo[1]), Double.parseDouble(participantInfo[2]));` | O(1) |
| `Location location = new Location(participantInfo[0], coordinates);` | O(1) |
| `String participantType = participantInfo[3].substring(0,1);` | O(1) |
| `Participant participant = switch (participantType) {` | O(1) |
| `case Participant.PARTICULAR -> new Particular(participantInfo[3], location);` | O(1) |
| `case Participant.COMPANY -> new Company(participantInfo[3], location);` | O(1) |
| `case Participant.PRODUCER -> new Producer(participantInfo[3], location);` | O(1) |
| `default -> null;}` | O(1) |
| `if (participant == null) {` | O(1) |
| `return false;}` | O(1) |
| `if (!distributionNetworkData.getDistributionNetwork().containsParticipant(participant)){` | O(V) |
| `distributionNetworkData.getDistributionNetwork().addParticipant(participant);}` | O(1) |
| `return true;` | O(1) |
| | O(V) |

➔ Ficheiro de distâncias (arestas)

O método utilizado para a leitura de distâncias trata-se também de um método determinístico, uma vez que a sua complexidade temporal não varia dependendo do tipo de input. Passamos agora a uma análise mais detalhada do método:

| | Line Complexity |
|---|---|
| `public boolean readFromFile(String path) {` | |
| `BufferedReader reader = new BufferedReader(new FileReader(path));` | O(1) |
| `String line = reader.readLine();` | O(1) |
| `while ((line = reader.readLine()) != null) {` | O(N) |
| `String[] lineData = line.split(",");` | O(1) |
| `if (lineData.length != 3) {` | O(1) |
| `return false; }` | O(1) |
| `String origin = lineData[0];` | O(1) |
| `String destination = lineData[1];` | O(1) |
| `double distance = Double.parseDouble(lineData[2]);` | O(1) |
| `Participant originParticipant = distributionNetworkData.getDistributionNetwork().participantByLocationId(origin);` | O(N) |
| `Participant destinationParticipant = distributionNetworkData.getDistributionNetwork().participantByLocationId(destin` | O(N) |
| `if (originParticipant == null \|\| destinationParticipant == null) {` | O(1) |
| `System.out.println("You can't add a route between two participants, if one don't exist in the network");` | O(1) |
| `return false;}` | O(1) |
| `distributionNetworkData.getDistributionNetwork().addRoute(originParticipant, destinationParticipant, distance);}` | O(1) |
| `return true;` | O(1) |
| | O(N) |

# Algoritmos

## Algoritmo Kruskal

O algoritmo de Kruskal é aplicado num contexto específico, sendo este, a procura de árvores geradoras.

Árvores geradoras podem ser entendidas como esqueletos desse grafo, no sentido em que ligam todos os vértices do grafo e são minimais para esta condição. Vários problemas recorrem às árvores geradoras do grafo.

No entanto, o problema que abordamos é o de encontrar a árvore geradora de um grafo ponderado cuja soma dos pesos das suas arestas seja miníma (soma dos pesos das arestas é minímo).

Para resolver este problema, apresentamos o algoritmo de Kruskal que encontra a árvore geradora miníma de um grafo ponderado recorrendo a uma estratégia particular, isto porque, é um algoritmo considerado guloso, no sentido em que, para cada iteração, faz a melhor escolha.

Além disto, é importante referir que nem sempre um algoritmo guloso produz a solução ótima, não sendo o caso do algoritmo de Kruskal.

Este algoritmo tem por base ir escolhendo sempre a aresta de menor custo que ainda não se encontra na árvore geradora em construção e que não forma um ciclo com nenhum conjunto de arestas dessa árvore. O algoritmo termina após termos escolhido v – 1 arestas, onde v é o número de vértices do grafo, tornado-o assim um algoritmo não determinístico.

Vamos agora ver a sua complexidade na notação de “big-Oh”.

| | # Line Complexity |
|---|---|
| void ascendingSortEdges(List<KruskalObject<V, E>> kruskalObjectL | |
| { kruskalObjectList.sort((o1, o2) -> { | O(E log(E)) |
| if ((Double) o1.getWeight() - (Double) o2.getWeight() | O(1) |
| if ((Double) o1.getWeight() - (Double) o2.getWeight() | O(1) |
| return 0; } | O(1) |
| } | O(E log(E)) |

| | # Line Complexity |
|---|---|
| void kruskalAlgorithm(Graph<V, E> mst, List<KruskalObject<V, E>> kruskalList, Map<V, List<V>> c, Map<V, List<V>> p, List<KruskalObject<V, E>> l) | |
| { ascendingSortEdges(kruskalObjectList); | O(E log(E)) |
| for (KruskalObject<V, E> kruskalObject : kruskalObjectList) { | O(E) |
| V originParent = vParentMap.get(kruskalObject.getvOrig()).get(vParentMap.get(kruskalObject.getvOrig())); | O(1) |
| V destinationParent = vParentMap.get(kruskalObject.getvDest()).get(vParentMap.get(kruskalObject.getvDest())); | O(1) |
| if (originParent != destinationParent) { | O(1) |
| vConnectionMap.get(originParent).addAll(vConnectionMap.get(destinationParent)); | O(1) |
| mst.addEdge(kruskalObject.getvOrig(), kruskalObject.getvDest(), kruskalObject.getWeight()); | O(1) |
| listToPrint.add(kruskalObject); | O(1) |
| for (int position = 0; position < vConnectionMap.get(destinationParent).size(); position++) | O(log(E)) |
| vParentMap.get(vConnectionMap.get(destinationParent).get(position)).add(originParent); | O(1) |
| } | O(E log(E)) |

Como foi evidenciado, o algoritmo tem uma complexidade O(E log(E)), onde “E” é o número de arestas do grafo.

## Algoritmo Dijkstra

O algoritmo do Dijkstra é usado para descobrir o caminho mais curto entre um vértice de origem e um outro qualquer, num grafo não orientado, conexo e com pesos positivos nas arestas.

A ideia básica deste algoritmo é ir determinando sucessivamente passeios de menor distância (custo).

O nosso algoritmo devolverá então uma lista com todos os vértices, as respetivas distâncias do vértice de origem até ele mesmo e um vértice procedente associado (que representa o vértice anterior ao caminho de menor custo).

| | Line Complexity |
|---|---|
| `public static <V, E extends Number> void shortestPathDijkstra(Graph<V, E> graph, V vOrig, Map<V, Boolean> visited, Map<V, DijkstraObject<V, E>> dijkstraObjectMap) {` | |
| `if (!graph.validVertex(vOrig))` | O(1) |
| `  return;` | O(1) |
| `int vertCounter = 0;` | O(1) |
| `while (vertCounter < graph.numVertices()) {` | O(V) |
| `  if (vOrig == null)` | O(1) |
| `  return;` | O(1) |
| `  for (Edge<V, E> edge: graph.outgoingEdges(vOrig)) {` | O(E) |
| | O(1) |
| `  if (!visited.get(edge.getVDest())) {` | O(1) |
| `  Double sum = dijkstraObjectMap.get(vOrig).getDistance().doubleValue() + edge.getWeight().doubleValue();` | O(1) |
| `  E current = dijkstraObjectMap.get(edge.getVDest()).getDistance();` | O(1) |
| `  if (current == null || Double.compare(sum, current.doubleValue()) < 0) {` | O(1) |
| `  dijkstraObjectMap.get(edge.getVDest()).setDistance((E) sum);` | O(1) |
| `  dijkstraObjectMap.get(edge.getVDest()).setPredecessor(vOrig); } } }` | O(1) |
| `  visited.put(vOrig, true);` | O(1) |
| `  vertCounter++;` | O(1) |
| `  vOrig = getVertexWithLeastDistance(dijkstraObjectMap, visited);` | O(V) |
| `}` | O(VE) |

## Algoritmo BreadthFirstSearch

A pesquisa em largura, também conhecida como BFS, encontra os caminhos mais curtos de um determinado vértice de origem para todos os outros vértices, em termos do número de arestas nos caminhos.

| | Line Complexity |
|---|---|
| `public static <V, E> LinkedList<V> BreadthFirstSearch(Graph<V, E> graph, V vert) {` | |
| | |
| `  if (!graph.validVertex(vert))` | O(1) |
| `  return null;` | O(1) |
| `  LinkedList<V> qSearch = new LinkedList<>();` | O(1) |
| `  LinkedList<V> qHelper = new LinkedList<>();` | O(1) |
| `  boolean[] visited = new boolean[graph.numVertices()];` | O(1) |
| `  qSearch.add(vert);` | O(1) |
| `  qHelper.add(vert);` | O(1) |
| `  int vertKey = graph.key(vert);` | O(1) |
| `  visited[vertKey] = true;` | O(1) |
| `  while (!qHelper.isEmpty()) {` | O(V) |
| `  vert = qHelper.remove();` | O(1) |
| `  for (V vertAdj : graph.adjVertices(vert)) {` | O(V) |
| `  vertKey = graph.key(vertAdj);` | O(1) |
| `  if (!visited[vertKey]) {` | O(1) |
| `  qSearch.add(vertAdj);` | O(1) |
| `  qHelper.add(vertAdj);` | O(1) |
| `  visited[vertKey] = true; } } }` | O(1) |
| `  return qSearch;` | O(1) |
| `}` | $O(V^2)$ |

# US302 - Obter o Par de Vértices mais afastados

## Funcionalidades

### Objetivo

Verificar se o grafo carregado é conexo e calcular o número de ramos entre os dois utilizadores mais afastados, ou seja, o diâmetro do grafo.

### Abordagem

O objetivo desta funcionalidade é, tal como referido em cima, obter o par que apresenta a maior distância entre si. Posto isto existem duas abordagens possíveis:

- Maior distância por peso
- Maior distância por conexões

Ora, tendo em conta o enunciado, interpretamos que devíamos calcular o número de ramos através do número de conexões.

O primeiro passo é verificar se o grafo é conexo, pois caso contrário, esta funcionalidade não correrá.

Para isto, usamos o método ***isConnected*** que por sua vez usa o algoritmo ***BreadFirstSearch***.

***BreadthFirstSearch*** irá fazer uma visita a todos os vértices, ou seja, se o tamanho da lista que este devolve for do tamanho do número de vértices que o grafo apresenta, isto significada que o grafo é *conexo.*

Tendo o algoritmo de Dijkstra, a nossa abordagem para obter este par passa por aplicar este algoritmo a todos os *vértices* e, de seguida, ver para cada iteração qual o vértice que se apresenta mais *distante* de cada vértice.  Ou seja, para os vértices C1, C2 e C3 vamos ver qual o vértice mais distante de cada e, de seguida, vamos ver qual o par que apresenta a maior distância.

O primeiro ***for*** trata de obter todas as distâncias a partir de um vértice (Dijkstra) e adicionar à lista ***longestParticipants*** o par que apresenta a maior distância nessa iteração.

Finalmente, temos um ***Iterator*** e um ***While*** que têm como objetivo iterar sobre a ***List longestParticipants*** e ver qual é o par que apresenta a maior distância entre os pares mais distantes.

# Análise de Complexidade

## IsConnected

| | Line Complexity |
|---|---|
| `public boolean isConnected() {` | |
| `if (vertices.size() == 0)` | O(1) |
| `return false;` | O(1) |
| `LinkedList<V> verticesFound = Algorithms.BreadthFirstSearch(this, this.vertices.get(0));` | $O(V^2)$ |
| `return verticesFound.size() == vertices.size();` | O(1) |
| `}` | $O(V^2)$ |

## getLongestPath

| | Line Complexity |
|---|---|
| `public DijkstraObjectDTO<Participant, Double> getLongestPath() {` | |
| `List<DijkstraObject<Participant, Double>> longestParticipants = new ArrayList<>();` | O(1) |
| `for (Participant p : distributionNetwork.getDistributionNetwork().vertices()) {` | O(V) |
| `dijkstraSetUp(p);` | O(1) |
| `Algorithms.shortestPathDijkstra(distributionNetwork.getDistributionNetwork(), p, visited, dijkstraObjectMap);` | O(V × E) |
| `longestParticipants.add(Algorithms.getVertexWithMostConnections(p, dijkstraObjectMap)); }` | $O(V^2)$ |
| `Iterator<DijkstraObject<Participant, Double>> iterator = longestParticipants.iterator();` | O(1) |
| `DijkstraObject<Participant, Double> longest = iterator.next();` | O(1) |
| `while (iterator.hasNext()) {` | O(V) |
| `DijkstraObject<Participant, Double> current = iterator.next();` | O(1) |
| `if (current.getDistance() > longest.getDistance())` | O(1) |
| `longest = current;}` | O(1) |
| `return DijkstraObjectMapper.toDTO(longest);` | O(1) |
| `}` | $O(V^2 \times E)$ |

## getVertexWithMostConnections

| | Line Complexity |
|---|---|
| `public static <V, E extends Number> DijkstraObject<V, E> getVertexWithMostConnections(V vOrig, Map<V, DijkstraObject<V, E>> dijkstraObjectMap) {` | |
| `V predecessor = null;` | O(1) |
| `double maxConnections = 0;` | O(1) |
| `double connections;` | O(1) |
| `for (Map.Entry<V, DijkstraObject<V, E>> entry : dijkstraObjectMap.entrySet()) {` | O(V) |
| `connections = 0;` | O(1) |
| `V vertex = entry.getValue().getPredecessor();` | O(1) |
| `V currentVertex = entry.getKey();` | O(1) |
| `while (vertex != null) {` | O(V) |
| `vertex = dijkstraObjectMap.get(vertex).getPredecessor();` | O(1) |
| `connections++;}` | O(1) |
| `if (connections > maxConnections) {` | O(1) |
| `maxConnections = connections;` | O(1) |
| `predecessor = currentVertex;}}` | O(1) |
| `return new DijkstraObject<>(vOrig, (E) new Double(maxConnections), predecessor);` | O(1) |
| `}` | $O(V^2)$ |

## Possíveis Melhorias

Uma possível melhoria seria no método ***getVertexWithMostConnections***.

O objetivo deste método é ver qual o vértice que apresenta maior número de ligações entre si e um vértice de destino. Ora, para fazer isto, haverá inúmeros vértices que apareceram repetidos, pois para chegar ao vértice de destino os caminhos terão de passar pelos vértices à volta desse.

Posto isto, uma abordagem seria, ao construir o caminho, ver se há tínhamos passado por um certo vértice e usar a distância associada a este, ou seja, por exemplo:

C1 -> C2 || C1 -> C3 || C3 -> C4

O método irá executar-se para o vértice C2 e verá que o número de conexões mínimo é 1, o mesmo irá acontecer para o C3. Depois ao fazer para o vértice C4, já sabendo a distância do C3, poderíamos usar logo esta distância, poupando assim o trabalho de executar o resto do caminho.

Outra possível melhoria seria usar o método ***BreadFirstSearch*** em vez do método ***shortestPathDijkstra***, pois o objetivo do BFS é encontrar o mínimo caminho tendo em conta as ligações em vez do peso.

# US303 – Definir os Hubs da rede de distribuição

## Funcionalidades

- Definir os hubs da rede de distribuição

### Objetivo

O objetivo desta user story é, após carregar a rede de distribuição, determinar e definir os "N" hubs da mesma. De acordo com o input do utilizador e recorrendo por base ao algoritmo de Dijkstra acima explorado, é feito o cálculo de quais são as empresas mais próximas de todos os pontos da rede, sendo que deve ser considerada como medida a média do comprimento do caminho mais curto de cada empresa a todos os clientes e produtores agrícolas.

### Abordagem

Entrando em mais pormenor, esta funcionalidade recorre ao uso de dois métodos, o ***determineHubs*** no qual para cada empresas é então calculada a distância acima referida, através do método ***calculateDistanceAverage***, e depois de acordo com o número de hubs pedido é feito o top N destas, definindo assim as empresas referentes como Hub's e devolvendo uma lista com os ID's dos mesmos.

## Análise da complexidade

### *determineHubs*

| | Line Complexity |
|---|---|
| `public List<String> determineHubs(int numberOfHubs) {` | |
| `    List<Participant> companies = getCompanies();` | O(V) |
| `    List<Hub> hubs = new ArrayList<>();` | O(1) |
| `    double companyAverageDistance;` | O(1) |
| `    for (Participant company : companies) {` | O(V) |
| `        companyAverageDistance = calculateDistancesAverage(company);` | O(VE) |
| `        hubs.add(new Hub(company.getParticipantId(), companyAverageDistance));  }` | O(1) |
| `    Collections.sort(hubs);` | O(V) |
| `    List<String> hubIds = new ArrayList<>();` | O(1) |
| `    for (int i = 0; i < numberOfHubs; i++) {` | O(N) |
| `        String id = hubs.get(i).participantID();` | O(1) |
| `        hubIds.add(id);` | O(1) |
| `        Company newHub = (Company) getParticipantByID(id);` | O(1) |
| `        newHub.markAsHub();` | O(1) |
| `    return hubIds;` | O(1) |
| `}` | O(VE) |

## *calculateDistanceAverage*

| | Line Complexity |
|---|---|
| `private double calculateDistancesAverage(Participant company) {` | |
| `    double totalDistance = 0;` | O(1) |
| `    Map<Participant, Boolean> visited = new HashMap<>();` | O(1) |
| `    Map<Participant, DijkstraObject<Participant, Double>> dijkstraObjectMap = new HashMap<>();` | O(1) |
| `    DijkstraObject<Participant, Double> first = new DijkstraObject<>(company, 0.0, null);` | O(1) |
| | |
| `    dijkstraObjectMap.put(company, first);` | O(1) |
| `    for (Participant participant : distributionNetwork.vertices()) {` | O(V) |
| `        if (!participant.equals(company))` | O(1) |
| `            dijkstraObjectMap.put(participant, new DijkstraObject<>(participant, null, null));` | O(1) |
| `        visited.put(participant, false); }` | O(1) |
| `    Algorithms.shortestPathDijkstra(distributionNetwork, company, visited, dijkstraObjectMap);` | O(VE) |
| `    for (DijkstraObject<Participant, Double> dijkstraObject : dijkstraObjectMap.values()) {` | O(V) |
| `        if (dijkstraObject.getDistance() != null)` | O(1) |
| `            totalDistance += dijkstraObject.getDistance();  }` | O(1) |
| `    return totalDistance / (distributionNetwork.numVertices() - 1);` | O(1) |
| `}` | O(VE) |

# US304 - Hub mais próximo de cada Cliente

## Funcionalidades

### Objetivo

A US304 é responsável por determinar o HUB mais próximo de cada cliente, empresa ou particular, para tal o nosso grafo deve conter no mínimo 1 Hub.

### Abordagem

É importante destacar que se uma empresa for simultaneamente um HUB, não deve ser considerada como o HUB mais próximo de si mesma.

O princípio do algoritmo que utilizamos é simples, recorrer ao algoritmo de Dijkstra para determinar a distância mínima de todos os vértices de origem para todos os vértices a partir de si alcançáveis.

Com isto, podemos determinar qual o vértice destino que é simultaneamente um HUB e cuja distância é mínima, para cada um dos clientes da nossa rede de distribuição.

Esta funcionalidade apresenta enorme relevância no contexto em que a realização deste projeto se enquadra, uma vez que, é fundamental para cada cliente saber qual o centro de distribuição/recolha mais próximo de si.

Passemos agora à demonstração do algoritmo em si e da respetiva complexidade.

## Análise de Complexidade

| | # Line Complexity |
|---|---|
| `Map<String, String> getNearestHub()` | |
| `{ List<Participant> listOfClients = getListOfClients();` | O(1) |
| `return networkData.getDistributionNetwork().findNearest` | O(V + E) |
| `}` | O(V + E) |

| | # Line Complexity |
|---|---|
| `String findNearestHUBDistance(Participant client)` | |
| `  double minDistance = Double.MAX_VALUE;` | O(1) |
| `  List<String> hubIds = new ArrayList<>();` | O(1) |
| `  DijkstraObject<Participant, Double> first = new DijkstraObject<>(client, 0.0, null);` | O(1) |
| `  dijkstraObjectMap.put(client, first);` | O(1) |
| `  for (Participant participant : distributionNetwork.vertices()) {` | O(V) |
| `    if (!participant.equals(client)) dijkstraObjectMap.put(participant, (participant, null, null));` | O(1) |
| `    visited.put(participant, false); }` | O(1) |
| `  Algorithms.shortestPathDijkstra(distributionNetwork, client, visited, dijkstraObjectMap);` | O() |
| `  for (DijkstraObject<Participant, Double> dijkstraObject : dijkstraObjectMap.values()) {` | O(E) |
| `    if (dijkstraObject.getKey().getParticipantId().charAt(0) == 'E' && dijkstraObject.getDistance() != null) {` | O(1) |
| `      Company p = (Company) dijkstraObject.getKey();` | O(1) |
| `      if (p.isHub() && !client.equals(dijkstraObject.getKey())) {` | O(1) |
| `        minDistance = dijkstraObject.getDistance();` | O(1) |
| `        hubIds.add(dijkstraObject.getKey().getParticipantId());` | O(1) |
| `  return hubIds.get(hubIds.size() - 1);` | O(1) |
| `}` | O(V + E) |

# US305 - Árvore Geradora Mínima

## Funcionalidades

### Objetivo

De acordo com a US305, devemos ser capazes de determinar a rede que conecte todos os clientes e produtores agrícolas com uma distância total miníma, ou seja, determinar a árvore geradora miníma.

### Abordagem

Para tal recorremos ao algoritmo de Kruskal, já referido anteriormente, que é capaz de determinar esta rede quando lhe é apresentado um grado poderado e de forma perentória, um grafo conexo.

Assim sendo, o primeiro passo a ser tomado é garantir que o grafo a ser analisado é conexo, caso contrário, não será possível determinar a sua árvore geradora. De seguida, devemos ordenar as arestas no formato crescente de peso e inicializar conjuntos para cada vértice existente.

Sabendo isto, resta apenas desenvolver um algoritmo que a partir do de Kruskal seja capaz de responder ao que nos é requirido.

| | # Line Complexity |
|---|---|
| void runKruskalAlgorithm() | |
| {if (isConnected) { | O(1) |
| List<KruskalObject<Participant, Double>> kruskalObjectList = new ArrayList<>(); | O(1) |
| fillListWithKruskalObject(kruskalObjectList); | O(E) |
| Algorithms.kruskalAlgorithm(minimumSpanningTree, kruskalObjectList, participantConnectionMap, participantParentMap, listToPrint); | O(E log(E)) |
| } | O(E log (E)) |

Apresentamos agora um exemplo, que retrata o retorno deste algoritmo quando utilizamos o ficheiro de dados “small”:

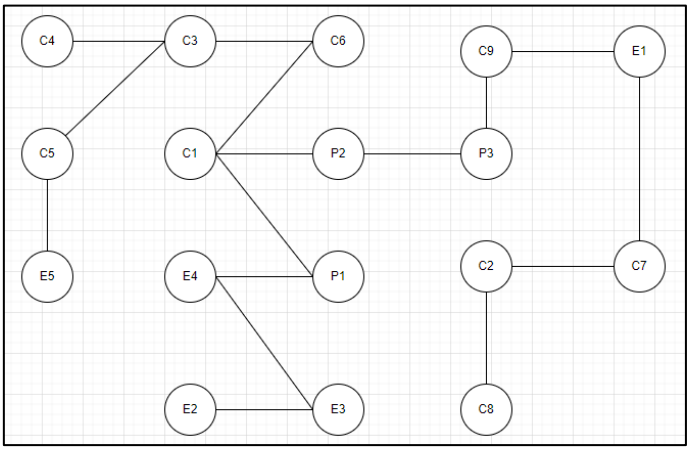


*Figura 1 – Selecionar Lista de Expedição*

# US308/US309 – Gerar uma lista de expedição

Dada a similaridade destas duas User Stories à reutilização de métodos de uma, noutra, a realização da análise destas vai ser feita em conjunto, referindo claro as suas diferenças.

## Funcionalidades

- Gerar uma lista de expedição simples, para um determinado dia "N", escolhido pelo utilizador, sem qualquer restrição quanto aos produtores.
- Gerar uma lista de expedição, para um determinado dia "N" que forneça apenas com os N produtores agrícolas mais próximos do hub de entrega do cliente. Critério de Aceitação: A lista de expedição deve indicar para cada cliente/cabaz, os produtos, quantidade encomendada/expedida e o produtor que forneceu o produto.

### Objetivo

O principal objetivo de ambas estas User Stories é de fornecer uma lista de expedição para um determinado dia "N" que o utilizador escolha. Para simplificar, vamos considerar que o primeiro tipo de lista se trata de uma lista de expedição simples e o segundo de uma lista dos produtores mais próximos do Hub do cliente.

Uma lista de expedição trata-se apenas de uma lista de encomendas que foram satisfeitas totalmente ou parcialmente. Para cada encomenda há a tentativa de satisfazer cada produto, tendo em conta que um uma encomenda pode ser satisfeita por vários produtores, no entanto, cada produto pode apenas ser satisfeito por um produtor. Uma lista de expedição tem sempre um dia associado a si, pelo que só pode conter encomendas referentes ao dia em questão, e cada encomenda pode apenas ser preenchida por produtos que estejam disponíveis na rede de distribuição.

Para gerar uma lista de expedição simples, basta apenas iterar sobre todas as encomendas do dia "N" escolhido e tentar preenchê-las, se estas forem preenchidas, são adicionadas à lista. Sempre que um produto é disponibilizado para satisfazer uma encomenda, o stock da rede é atualizado. Neste caso, uma vez que não existe nenhum tipo de restrição quanto aos produtores, é utilizado o critério FIFO (First In First Out) de forma a escoar os produtos que estão presentes há mais tempo na rede, antes que os mesmos percam a validade.

Para gerar uma lista de expedição dos produtores mais próximos do Hub do cliente, toda a lógica do primeiro tipo de listas se mantém, no entanto, há então uma restrição nos produtos que são utilizados para satisfazer a encomenda de um cliente. Neste tipo de listas apenas os "N" (definido pelo utilizador) produtores mais próximos do Hub do cliente podem disponibilizar produtos para satisfazer as encomendas desse mesmo cliente.

### Abordagem

A abordagem para tornar esta geração possível passou por criar um ***DispatchListGenerator***, nele existem os métodos responsáveis por gerar ambas as listas.

Tal como foi referido a única coisa que difere de uma para a outra, são os produtos que são utilizados, isto permitiu o desenvolvimento de um método comum para gerar cada lista e de um outro método comum para preencher cada produto.

Passamos a uma breve explicação do método gerador, ***generateDispatchList***:

Inicialmente, é criada a instância da lista que vai ser retornada, depois há uma filtragem, pelo dia escolhido, de todas as encomendas existentes na rede.
Tendo então a lista de encomendas, uma a uma, vai haver uma tentativa de preencher todos os seus produtos.
Começa-se pela verificação de que de facto existem produtos disponíveis na rede, depois há chamada do método ***setProductsToBeUsedInOrder***, este tem a responsabilidade de determinar que produtos devem ser utilizados para satisfazer a encomenda.
Tendo a lista de produtos que vão ser utilizados para satisfazer a encomenda, procede-se à iteração de produto a produto da mesma e à tentativa de preencher completamente cada um dos produtos encomendados, utilizando o método ***fillProductRecursively***. Se pelo menos um produto for considerado preenchido a encomenda é considerada como, pelo menos, parcialmente preenchida.

```
private List<BasketOrder> generateDispatchList(List<BasketOrder> orders, int day, boolean isSimpleDispatchList) {
    List<BasketOrder> dispatchList = new ArrayList<>();
    // Get the list of orders of that day
    List<BasketOrder> ordersOfThatDay = orders.stream()
            .filter(order -> order.getDay() == day)
            .toList();
    // For each order fill it with the products available
    for (BasketOrder order : ordersOfThatDay) {
        if (productsAvailableInTheNetwork.isEmpty()) {
            order.setIsFulfilled(false);
            break;
        }
        // Set the products to be used in order
        setProductsToBeUsedInOrder(order, isSimpleDispatchList);
        // For each product in the order, try to fill it
        boolean isOrderFulfilled = false;
        for (Product product : order.getProducts()) {
            // Fill the product with the products available
            boolean isProductFilled = fillProductRecursively(product, day, order);
            // If the filling is successful, set the order as fulfilled
            if (isProductFilled) {
                isOrderFulfilled = true;
            } else order.setIsFulfilled(false);
        }
        // If the order is fulfilled, add it to the dispatch list and remove it from the orders list
        if (isOrderFulfilled) {
            dispatchList.add(order);
            orders.remove(order);
        }
    }
    // If no orders were fulfilled, return null
    if (dispatchList.isEmpty()) {
        return null;
    }
    return dispatchList;
}
```

Após todos os produtos da encomenda terem tentado ser preenchidos, verifica-se se a encomenda foi pelo menos parcialmente preenchida, se sim adicionamos a encomenda à lista

de expedição e removemos a encomenda da lista de encomendas.
Para finalizar, é verificado se a lista de expedição tem pelo menos uma encomenda expedida.

Com isto, é pertinente analisar os métodos ***setProductsToBeUsedInOrder*** e ***fillProductRecursively***. Seguimos com a análise do primeiro:

```
private void setProductsToBeUsedInOrder(BasketOrder order, boolean isSimpleDispatchList) {
    if (isSimpleDispatchList) {
        productsToBeUsedInOrder = productsAvailableInTheNetwork;
    } else {
        productsToBeUsedInOrder = getProductsForDispatchListWithRestrictions(order.getClient(), numberOfProducersToConsiderPerClient);
    }
}
```

Trata-se de um método simples, que apenas seleciona qual a lista de produtos a ser utilizada, no caso de se querer gerar uma lista de expedição simples utilizamos todos os produtos da rede, já se quisermos o segundo tipo de lista, é necessário criar a lista de produtos que tenha em conta a condição de que só os "N" produtores mais próximos do cliente podem fornecer produtos para a satisfação desse cabaz. É importante relembrar que este método é chamado uma vez por cada encomenda. Para criar esta lista é chamado o método ***getProductsForDispatchListWithRestrictions***.

```
private List<Product> getProductsForDispatchListWithRestrictions(Client client, int n) {
    // Get the list of N producers that are closest to the client
    List<Producer> eligibleProducers = networkData.getDistributionNetwork().getNProducersClosestToClientHub(client, n);
    // Get the list of products available from the eligible producers
    List<Product> productsAvailable = new ArrayList<>();
    for (Producer producer : eligibleProducers) {
        // Filter the products available in the network by the producers that are eligible
        productsAvailable.addAll(productsAvailableInTheNetwork.stream()
                .filter(product -> product.getProducer().equals(producer))
                .toList());
    }
    return productsAvailable;
}
```

Este começa por obter a lista dos "N" produtores mais próximos do Hub do cliente e depois cria a lista de produtos que deve ser considerada através da filtragem dos produtos disponíveis na rede, ficando apenas com aqueles que dizem respeito aos produtores elegíveis.
Para obter a lista dos produtores elegíveis é utilizado o método ***getNProducersClosestToClientHub***, vamos explorá-lo um pouco.

```java
public List<Producer> getNProducersClosestToClientHub(Client client, int n) {
    // Get Dijkstra's map from the hub (distanceToAllTheParticipants)
    Map<Participant, DijkstraObject<Participant, Double>> distanceToAllTheParticipants = runDijkstra(getParticipantByID(client.getNearestHubId()));
    // Get a map with the producers and their distance from the hub
    Map<Participant, DijkstraObject<Participant, Double>> distanceToAllTheProducers = new HashMap<>();
    for (Participant participant : distanceToAllTheParticipants.keySet()) {
        if (participant.getParticipantId().startsWith(Participant.PRODUCER)) {
            distanceToAllTheProducers.put(participant, distanceToAllTheParticipants.get(participant));
        }
    }
    // Sort the map by the distance
    List<Map.Entry<Participant, DijkstraObject<Participant, Double>>> sortedMap = new ArrayList<>(distanceToAllTheProducers.entrySet());
    sortedMap.sort(Comparator.comparingDouble(o -> o.getValue().getDistance()));
    // Get the n producers that are closest to the hub
    List<Producer> closestProducers = new ArrayList<>();
    for (int i = 0; i < n; i++) {
        closestProducers.add((Producer) sortedMap.get(i).getKey());
    }
    return closestProducers;
}
```

Este utiliza o Algoritmo de Dijkstra, analisado em TODOMUDARX, de forma a obter a distância do Hub a todos os participantes da rede. Dado o resultado da sua utilização é feita uma filtragem de forma a obter um Mapa apenas com a distância do Hub a todos os produtores da rede. Finalmente, este mapa é ordenado, por ordem crescente, sendo assim possível ir buscar os “N” primeiros elementos dele, sendo esses os produtores elegíveis.

Fica então a faltar a análise do método ***fillProductRecursively***. Uma vez que cada produto pode apenas ser produzido por um produtor, há situações nas quais o produtor, digamos “A” tem disponível o produto “Banana” em quantidade 3 no dia 1 e quantidade 6 no dia 2, se ao gerar uma lista de expedição, independentemente do tipo, se houver um pedido, por parte de um cliente, de “Banana” em quantidade 9 e a lista que está a ser gerada é para o dia 2, primeiro o produto vai ser preenchido em 3, no entanto, verificamos que ainda o mesmo produtor pode satisfazer totalmente esta parte do pedido se também fornecer as 6 bananas que disponibiliza no dia 2. É exatamente por causa destes casos que este método se torna útil, primeiro, porque permite que este fluxo seja possível e assegurado e, segundo, porque caso as regras de negócio mudassem e o número de dias que um produto fica disponível na rede fosse alterado, este método continuava a assegurar que após um produto ser parcialmente fornecido, haveria uma tentativa de que o mesmo produtor completasse este preenchimento com o mesmo produto, mas disponibilizado em dias diferentes.

```java
private boolean fillProductRecursively(Product product, int day, BasketOrder order) {
    boolean result = false;
    // Get the equivalent product from the available products
    Product equivalentProduct = getEquivalentProduct(product, day);
    if (equivalentProduct != null) {
        result = true;
        // Fill the product with the equivalent product
        boolean isProducedProductFullyConsumed = product.fillProduct(equivalentProduct);
        // Check if the produced product is fully consumed
        if (isProducedProductFullyConsumed) {
            // Remove the produced product from the available products if it is fully consumed
            networkData.removeAndUpdateProductsSoldOutByProducer(equivalentProduct.getProducer(), equivalentProduct);
            if (product.getQuantity() != 0) {
                // Check if the same producer still has the same product available
                Producer producer = equivalentProduct.getProducer();
                Product sameProductFromSameProducer = getProductXFromSameProducer(producer, product, day);
                if (sameProductFromSameProducer != null) {
                    // Recursively call the method to try filling the product with the same product from the same producer
                    return fillProductRecursively(product, day, order);
                } else order.setIsFulfilled(false);
            }
        }
    }
    return result;
}
```

Começa por utilizar o método ***getEquivalentProduct***, este tem a responsabilidade de procurar nos produtos que vão ser utilizados para satisfazer a encomenda por algum produto que seja igual ao que se está a tentar preencher, tendo sempre em conta a validade do mesmo.

```java
private Product getEquivalentProduct(Product product, int day) {
    // Gets the first product that is of the same type
    for (Product productAvailable : productsToBeUsedInOrder) {
        if (productAvailable.getName().equals(product.getName())) {
            // Check if the product has already expired
            if (productAvailable.isTimelyAvailable(day)) {
                // If not expired, return the product
                return productAvailable;
            }
        }
    }
    return null;
}
```

Caso seja encontrado um produto disponível para ajudar a satisfazer o pedido, dá-se o processo de o preencher e caso seja verificado que o produto fornecido pelo produtor foi completamente utilizado, para além do stock ser atualizado, se se verificar também que o produto pedido não foi completamente satisfeito utiliza-se o método ***getProductXFromSameProducer***, este tem um comportamento semelhante ao ***getEquivalentProduct***, no entanto, aceita apenas produtos que sejam iguais ao que recebe como parâmetro e que tenha o produtor em comum, de forma a cumprir o requisito de só haver um produtor a satisfazer um produto.

```
private Product getProductXFromSameProducer(Producer producer, Product productX, int day) {
    for (Product product : productsAvailableInTheNetwork) {
        if (product.getProducer().equals(producer) && product.equals(productX)) {
            if (product.isTimelyAvailable(day)) {
                return product;
            }
        }
    }
    return null;
}
```

Se for obtido algum resultado não nulo deste método, há uma nova chamada recursiva do método ***fillProductRecursively***.

## Análise de Complexidade

Dada a importância deste processo, o objetivo é que o mesmo esteja o mais otimizado possível, foi procurado que a solução implementada fosse o mais eficiente possível. É então, relevante analisar a complexidade temporal no pior caso dos métodos referidos de forma a concluir a complexidade temporal do método ***generateDispatchList***.

Vamos fazer uma análise forma a analisar os métodos que são utilizados por outros primeiro, para que aquando da análise de um método que chame outros, esses ditos "outros" já tenham a sua análise completa.

| | Line Complexity |
|---|---|
| public *List*<Producer> getNProducersClosestToClientHub(Client client, int n) { | |
| *Map*<Participant, DijkstraObject<Participant, Double>> distanceToAllTheParticipants = runDijkstra(getParticipantByID(client.getNearestHubId()); | O(VE) |
| *Map*<Participant, DijkstraObject<Participant, Double>> distanceToAllTheProducers = new HashMap<>(); | O(1) |
| for (Participant participant : distanceToAllTheParticipants.keySet()) { | O(V) |
| if (participant.getParticipantId().startsWith(Participant.*PRODUCER*)) { | O(1) |
| distanceToAllTheProducers.put(participant, distanceToAllTheParticipants.get(participant)); } } | O(1) |
| *List*<*Map.Entry*<Participant, DijkstraObject<Participant, Double>>> sortedMap = new ArrayList<>(distanceToAllTheProducers.entrySet()); | O(V) |
| sortedMap.sort(*Comparator.comparingDouble*(o -> o.getValue().getDistance())); | O(Vlog(V)) |
| *List*<Producer> closestProducers = new ArrayList<>(); | O(1) |
| for (int i = 0; i < n; i++) { | O(V) |
| closestProducers.add((Producer) sortedMap.get(i).getKey());} | O(1) |
| return closestProducers; } | O(1) |
| | O(VE) |

| | Line Complexity |
|---|---|
| private List<Product> getProductsForDispatchListWithRestrictions(Client client, int n) { | |
| List<Producer> eligibleProducers = networkData.getDistributionNetwork().getNProducersClosestToClientHub(client, n); | O(VE) |
| List<Product> productsAvailable = new ArrayList <>(); | O(1) |
| for (Producer : producer eligibleProducers) { | O(V) |
| productsAvailable.addAll(productsAvailableInTheNetwork.stream(); | O(V) |
| .filter(product -> product.getProducer().equals(producer)) | O(V) |
| .toList()); } | O(V) |
| return productsAvailable; } | O(1) |
| | O(VE) |

| | Line Complexity |
|---|---|
| private void setProductsToBeUsedInOrder(BasketOrder order, boolean isSimpleDispatchList) { | |
| if(isSimpleDispatchList){ | O(1) |
| productsToBeUsedInOrder = productsAvailableInTheNetwork; | O(1) |
| } else { | O(1) |
| productsToBeUsedInOrder = getProductsForDispatchListWithRestrictions(order.getClient(), numberOfProducersToConsiderPerClient); } } | O(VE) |
| | O(VE) |

| | Line Complexity |
|---|---|
| private Product getEquivalentProduct(Product product, int day) { | |
| for(Product productAvailable : productsToBeUsedInOrder) { | O(N) |
| if (productsAvailable.getName().equals(product.getName())) { | O(1) |
| if (productAvailable.isTimelyAvailable(day)) { | O(1) |
| return productAvailable; } } } | O(1) |
| return null; } | O(1) |
| | O(N) |

| | Line Complexity |
|---|---|
| private Product getProductXFromSameProducer(Producer producer, Product productX , int day) { | |
| for(Product product : productsAvailableInTheNetwork) { | O(1) |
| if (product.getProducer().equals(producer) && product.equals(productX)) { | O(N) |
| if(product.isTimelyAvailable(day)) { | O(1) |
| return product; } } } | O(1) |
| return null; { | O(1) |
| | O(N) |

| | Line Complexity |
|---|---|
| private boolean fillProductRecursively(Product product, int day, BasketOrder order) { | |
| boolean result = false; | O(1) |
| Product equivalentProduct = getEquivalentProduct(product, day); | O(N) |
| if(equivalentProduct != null) { | O(1) |
| result = true ; | O(1) |
| boolean isProducedProductFullyConsumed = product.fillProduct(equivalentProduct); | O(1) |
| if (isProducedProductFullyConsumed) { | O(1) |
| networkData.removeAndUpdateProductsSoldOutByProducer(equivalentProduct.getProducer(), equivalentProduct); | O(1) |
| if(product.getQuantity() != 0) { | O(1) |
| Producer producer = equivalentProduct.getProducer(); | O(1) |
| Product sameProductFromSameProducer = getProductXFromSameProducer(producer, product, day) ; | O(N) |
| if (sameProductFromSameProducer != null ) { | O(1) |
| return fillProductRecursively(product, day, order); | O(N) |
| } else order.setIsFulfilled(false); } } } | O(1) |
| return result; } | O(1) |
| | O(N) |

| | Line Complexity |
|---|---|
| private List<BasketOrder> generateDispatchList(List<BasketOrder> orders, int day, boolean isSimpleDispatchList) { | |
| List<BasketOrder> dispatchList = new ArrayList<>(); | O(1) |
| List<BasketOrder> ordersOfThatDay = orders.stream(); | O(1) |
| .filter(order -> order.getDay() == day); | O(N) |
| .toList(); | O(N) |
| for (BasketOrder order : ordersOfThatDay) { | O(N) |
| if (productsAvailableInTheNetwork.isEmpty()) { | O(1) |
| order.setIsFulfilled(false); | O(1) |
| break; } | O(1) |
| setProductsToBeUsedInOrder(order, isSimpleDispatchList); | O(VE) |
| boolean isOrderFulfilled = false; | O(1) |
| for (Product product : order.getProducts()) { | O(P) |
| boolean isProductFilled = fillProductRecursively(product, day, order); | O(T) |
| if (isProductFilled) { | O(1) |
| isOrderFulfilled = true; | O(1) |
| } else order.setIsFulfilled(false); } | O(1) |
| if (isOrderFulfilled) { | O(1) |
| dispatchList.add(order); | O(1) |
| orders.remove(order); } } | O(N) |
| if (dispatchList.isEmpty()) { | O(1) |
| return null; } | O(1) |
| return dispatchList; } | O(1) |
| | O(NPT) |

Concluímos então que o método “principal”, ***generateDispatchList***, tem uma complexidade temporal, no pior caso, de O(NPT), onde N se trata do número de encomendas, P do número de produtos e T do número de produtos que podem ser utilizados para satisfazer uma dada encomenda.

Podemos ainda dizer que este se trata de um método não determinístico, uma vez que o método ***fillProductRecursively*** é não determinístico também, pois é possível a sua complexidade ser O(1), quando a complexidade dos métodos ***getEquivalentProduct*** e ***getProductXFromSameProducer***, são O(1). Apesar de ser muito difícil de acontecer, esta é uma possibilidade real, dado que se o produto que pretendemos procurar for sempre o primeiro da lista e preencher sempre o produto que é pedido completamente à primeira a complexidade temporal do método é mesmo O(1), tornando a do método ***generateDispatchList*** O(NP).

# US310 – Percurso de entrega que minimiza a distância percorrida

## Funcionalidades

### Objetivo

Esta funcionalidade tem como objetivo primordial permitir a geração de um caminho que minimize a distância total percorrida, quando se está a proceder à entrega de diversas encomendas que constituem os cabazes da rede de distribuição.

Simultaneamente devem ser apresentados todos os pontos de passagem do percurso, o número de cabazes entregues em cada hub e finalmente as distâncias entre todos os pontos do percurso (culminam numa distância total).

### Abordagem

Tendo em vista o nosso objetivo, o primeiro passo foi idealizar uma solução capaz de minimizar de forma visível o a distância total percorrida. Para isso, foram necessários ter em conta alguns aspetos, entre eles:

- Dia selecionado;
- Ponto de partida;
- Pontos de passagem;
- Priorizar proximidade;
- Controlar o número de cabazes entregues por hub.

Ponto de partida, um dos principais fatores a ter em conta é o ponto de partida do nosso caminho. Para uma maior flexibilidade, o ponto de começo deve ser sempre um Produtor presente na lista de expedição para o dia selecionado.

Este dia deve ter uma lista de expedição previamente associada, ou será impossível executar esta funcionalidade.

Outro aspeto crítico são os pontos de passagem do percurso, isto porque a rota gerada deve passar de forma mandatária em todos os pontos (hubs e produtores) associados à lista de expedição, da mesma maneira que o critério de priorização destes pontos, é a sua proximidade ao ponto prévio.

Para respeitar este requisito recorremos ao algoritmo de Dijkstra, primeiro para todos os produtores e de seguida, todos os hubs. Paralelamente, são também tidos em conta o número de cabazes que é entregue em cada um destes hubs.

Destacar também que todos os produtores e hubs possuem um estado, estado esse designado de "Visitado" com enorme utilidade, isto porque, certos pontos da nossa rede podem já ter sido visitados num dos percursos intermédios.

No que diz respeito à entrega dos cabazes, a única condição imposta, é que pelo menos um dos seus produtos tenha uma qualquer quantidade fornecida (pelo menos um dos produtos deve ser satisfeito, mesmo que não na sua quantidade total).

Com a abordagem definida, resta agora passar a ideia do papel para o ambiente de compilação. Como seria de esperar, todos os algoritmos e métodos implementados seguem os melhores padrões de desenvolvimento de software.

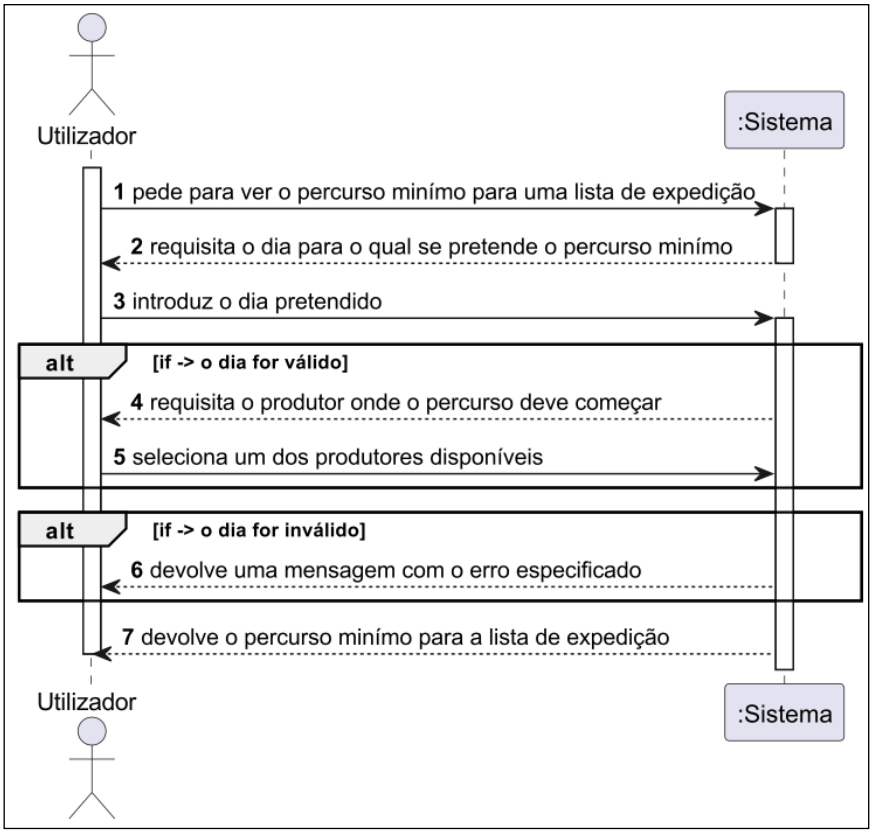


*Figura 1 - Diagrama de Sequência US310*

Finalmente destacar a ideia de que dependendo do ponto inicial, é esperado que as distâncias totais obtidas sejam diferentes, e até por vezes, menos eficientes do ponto de vista da distância percorrida.

## Implementação e Análise de Complexidade

Como foi referido anteriormente, a solução para o problema apresentado baseia-se no algoritmo de Dijkstra, e não passa de uma ligeira adaptação para utilizar todo o seu potencial.

| | # Line Complexity |
|---|---|
| void fillWithProducersAndHubs(int day, boolean restrictDispatchList) { | |
| int FIRST_BASKET = 1; | O(1) |
| List<BasketOrder> dispatchListForSelectedDay; | O(1) |
| if (restrictDispatchList) dispatchListForSelectedDay = getRestrictedDispatchList(day); | O(1) |
| else dispatchListForSelectedDay = getSimpleDispatchList(day); | O(1) |
| Set<Producer> producersPresentInDispatchSet = new HashSet<>(); | O(1) |
| Set<Company> hubsPresentInDispatchSet = new HashSet<>(); | O(1) |
| boolean basketIsEmpty = true; | O(1) |
| for (BasketOrder basketOrderForTheDay : dispatchListForSelectedDay) { | O(V) |
| for (Product product : basketOrderForTheDay.getProducts()) { | O(n) |
| if (product.getProducer() != null) { | O(1) |
| producersPresentInDispatchSet.add(product.getProducer()); | O(1) |
| basketIsEmpty = false; | O(1) |
| if (!basketIsEmpty) { | O(1) |
| hubsPresentInDispatchSet.add((Company) distributionNetwork.getParticipantByID(basketOrderForTheDay.getClient().getNearestHubId()) | O(1) |
| if (!basketsDroppedPerHub.containsKey(basketOrderForTheDay.getClient().getNearestHubId())) | O(1) |
| else basketsDroppedPerHub.put(basketOrderForTheDay.getClient().getNearestHubId()); | O(1) |
| } | O(n V) |

Este primeiro método, fillWithProducersAndHubs, é mais do que autoexplicativo, sendo a sua função preencher as estruturas de informação com os produtores e hubs que serão necessariamente visitados no caminho mínimo da lista de expedição.

| | # Line Complexity |
|---|---|
| Map<List<List<DijkstraObjectDTO<Participant, Double>>>, List<Double>> getMinimumDeliveryPathForExpeditionList(String startingPointId) { | |
| Participant startingPoint; | O(1) |
| if (!(startingPointId.startsWith("P"))) return null; | O(1) |
| else startingPoint = distributionNetwork.getParticipantByID(startingPointId); | O(V) |
| List<List<DijkstraObject<Participant, Double>>> pathPointsAndDistancesBetweenThem = new ArrayList<>(); | O(1) |
| List<Double> distancesBetweenPoints = new ArrayList<>(); | O(1) |
| participantsPresentInDispatchList.remove(startingPoint); | O(1) |
| Set<Producer> visitedProducers = new HashSet<>(); | O(1) |
| Set<Company> visitedHubs = new HashSet<>(); | O(1) |
| for (Participant inListProducerOrHub : participantsPresentInDispatchList) { | O(V) |
| if (inListProducerOrHub instanceof Producer) { | O(1) |
| if (!visitedProducers.contains(inListProducerOrHub)) { | O(1) |
| Map<Participant, DijkstraObject<Participant, Double>> distanceFromStartingPointToAllReachableVertices = distributionNetwork.runDijkstra(startingPoint); | O(V E) |
| List<DijkstraObject<Participant, Double>> shortestPathFromOriginToDestination = new ArrayList<>(); | O(1) |
| AuxiliaryAlgorithms.buildPath(shortestPathFromOriginToDestination, distanceFromStartingPointToAllReachableVertices, startingPoint, inListProducerOrHu | O(V) |
| startingPoint = inListProducerOrHub; | O(1) |
| for (DijkstraObject<Participant, Double> pathPoint : shortestPathFromOriginToDestination) { | O(V) |
| if (!(pathPoint.getKey() instanceof Company) && !(pathPoint.getKey() instanceof Particular)) { | O(1) |
| visitedProducers.add((Producer) pathPoint.getKey()); | O(1) |
| } | --- |
| pathPointsAndDistancesBetweenThem.add(shortestPathFromOriginToDestination); | O(1) |
| distancesBetweenPoints.add(shortestPathFromOriginToDestination.get(shortestPathFromOriginToDestination.size() - 1).getDistance()); | O(1) |
| } | --- |
| } | --- |
| else if (inListProducerOrHub instanceof Company) { | O(1) |
| if (!visitedHubs.contains(inListProducerOrHub)) { | O(1) |
| Map<Participant, DijkstraObject<Participant, Double>> distanceFromStartingPointToAllReachableVertices = distributionNetwork.runDijkstra(startingPoint); | O(V E) |
| List<DijkstraObject<Participant, Double>> shortestPathFromOriginToDestination = new ArrayList<>(); | O(1) |
| AuxiliaryAlgorithms.buildPath(shortestPathFromOriginToDestination, distanceFromStartingPointToAllReachableVertices, startingPoint, inListProducerOrHu | O(V) |
| startingPoint = inListProducerOrHub; | O(1) |
| for (DijkstraObject<Participant, Double> pathPoint : shortestPathFromOriginToDestination) { | O(V) |
| if (!(pathPoint.getKey() instanceof Producer) && !(pathPoint.getKey() instanceof Particular)) { | O(1) |
| visitedHubs.add((Company) pathPoint.getKey()); | O(1) |
| } | --- |
| } | --- |
| pathPointsAndDistancesBetweenThem.add(shortestPathFromOriginToDestination); | O(1) |
| distancesBetweenPoints.add(shortestPathFromOriginToDestination.get(shortestPathFromOriginToDestination.size() - 1).getDistance()); | O(1) |
| } | --- |
| } | --- |
| } | --- |
| | --- |
| return setToUI(pathPointsAndDistancesBetweenThem, distancesBetweenPoints); | O(V) |
| } | O(V V E) |

Este é o método principal, que deriva da implementação do algoritmo de Dijkstra.

O seu nome é getMinimumDeliveryPathExpeditionList e é responsável por determinar os caminhos intermédios que formam o nosso percurso, filtrar pontos já visitados e além disso comandar o número de cabazes entregue em cada Hub.

Este método recorre a dois ciclos 'for', que percorrem os participantes da lista de expedição (produtores e hubs) e os caminhos intermédios formados respetivamente.

Durante a execução do primeiro ciclo, são determinados os caminhos intermédios e também a renomeação do vértice de origem. O segundo ciclo é encarregado de marcar todos os pontos que já foram visitados em caminhos anteriores e cuja presença como vértice de origem já não é necessária.

Entende-se por vértice de origem, um vértice que se encontra presente na lista de expedição e que necessita de ser visitado de forma perentória.

## Possíveis Melhorias

A adoção do sistema produtores-hubs, pode nem sempre assegurar o caminho mais curto.

Com isto, não estamos a evidenciar um ponto negativo, mas sim uma realidade, que é a possibilidade de a entrega começar a ser feita em um qualquer ponto da rede.

Por outro lado, uma possível abordagem com um carater diferente, porém com as mesmas raízes poderia ter sido viável, neste caso referindo a possibilidade de recorrer ao algoritmo de Kruskal para utilizarmos apenas arestas de menor custo.

# US311 – Calcular estatísticas

## Funcionalidades

### Objetivo

A funcionalidade "Calcular Estatísticas" tem como objetivo permitir ao utilizador obter informações estatísticas sobre as listas de expedição geradas. Estas estatísticas são interessantes de manter pois permitem conhecer o nível de satisfação das encomendas dos clientes, o rendimento dos produtos dos produtores, a capacidade de fornecimento dos hubs e outros dados relevantes.

As estatísticas são calculadas para cada um dos seguintes componentes:

- Cesto de encomendas:
  - Número de produtos totalmente satisfeitos
  - Número de produtos parcialmente satisfeitos
  - Número de produtos insatisfeitos
  - Percentagem de satisfação do cesto
  - Quantidade de produtos que abastecem o cesto
- Cliente:
  - Número de cestas totalmente satisfeitas
  - Número de cestas parcialmente satisfeitas
  - Número de produtores diferentes que forneceram todas as suas cestas
- Produtor:
  - Número de cestas totalmente fornecidas
  - Número de cestas parcialmente fornecidas
  - Número de clientes diferentes fornecidos
  - Número de produtos completamente esgotados
  - Número de hubs fornecidos
- Local de recolha:
  - Número de clientes diferentes que recolhem cabazes em cada Hub
  - Número de produtores diferentes fornecendo cestas em cada Hub

## Abordagem

O primeiro passo para o desenvolvimento desta funcionalidade, para além de compreender o que era pretendido, foi fazer uma análise geral ao código e mais especificamente à forma de como as listas de expedição estavam a ser geradas. Isto deve-se a duas razões:

- Perceber se, durante este processo, os dados que precisávamos para calcular as estatísticas estavam a ser perdidos.
- Certificar se tínhamos acesso fácil e rápido a estes dados

Após esta análise, tudo estava pronto para a implementação e, depois de algum tempo a considerar a processo da funcionalidade, decidimos criar o seguinte fluxo, visível na figura 1 abaixo apresentada .

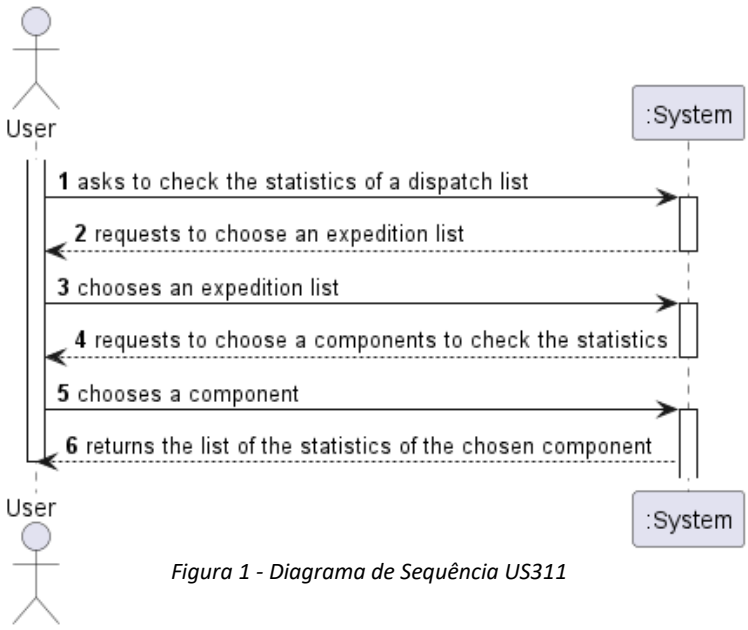


*Figura 1 - Diagrama de Sequência US311*

Passando para o cálculo em si, o utilizador deve primeiro selecionar a lista de expedição que deseja analisar. Um exemplo de como isto pode ser visualizado é apresentado na figura abaixo.

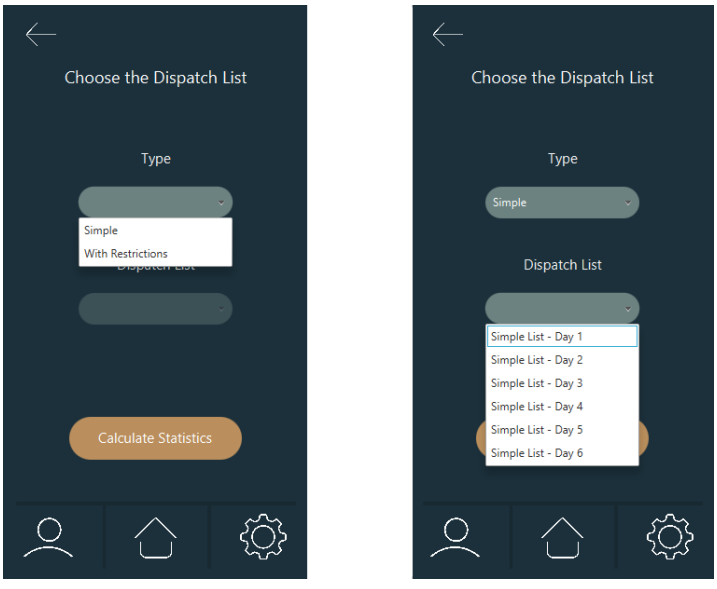


*Figura 2 – Selecionar Lista de Expedição*

De seguida, o processo de cálculo das estatísticas é iniciado através de uma iteração sobre os elementos da lista de expedição. Uma lista de expedição é composta por cestos de encomendas, cada encomenda é composta por uma lista de produtos. Para obter o pretendido, é então necessário iterar sobre todos os elementos da lista de expedição.

Tal como referido, existem quatro componentes. Para calcular estatísticas para cada um é preciso, de forma dependente de cada componente, iterar sobre a lista de expedição escolhida. Tendo isto em mente, decidimos iterar apenas uma vez e calcular tudo durante a iteração.

Para isto foram criadas estruturas de informação e classes próprias para cada componente.

```
private final List<BasketOrderStatistics> basketOrderStatistics;
private final Map<Client, ClientStatistics> mapClientStats;
private final Map<Producer, ProducerStatistics> mapProducerStats;
private final Map<String, HubStatistics> mapHubStats;
```

*Figura 3 – Estruturas de Informação e respetivas classes*

Os atributos destas classes são as métricas que pretendemos obter. Ao criar estas classes tornamos o processo de obtenção das estatísticas mais simples e intuitivo. Durante a iteração, são criadas ou reutilizadas instâncias dessas classes. Estas instâncias vão então sendo guardadas na estrutura de informação correspondente.

No final da iteração, todas as estatísticas estão calculadas e prontas para serem apresentadas ao utilizador. Para apresentar os números temos duas componentes visuais (figuras 3):

- Tabela – onde podemos ver todos os atributos por componente
- Gráfico de barras – onde podemos comparar atributo a atributo

| Totally Satisfied | Partially Satisfied | Not Satisfied | % Satisfaction | Nr Producers |
|---|---|---|---|---|
| 2 | 1 | 0 | 67.0 | 2 |
| 4 | 2 | 0 | 67.0 | 2 |
| 0 | 4 | 0 | 0.0 | 2 |
| 0 | 2 | 1 | 0.0 | 2 |
| 1 | 2 | 0 | 33.0 | 2 |
| 1 | 1 | 0 | 50.0 | 1 |
| 1 | 5 | 3 | 11.0 | 3 |
| 1 | 1 | 2 | 25.0 | 2 |

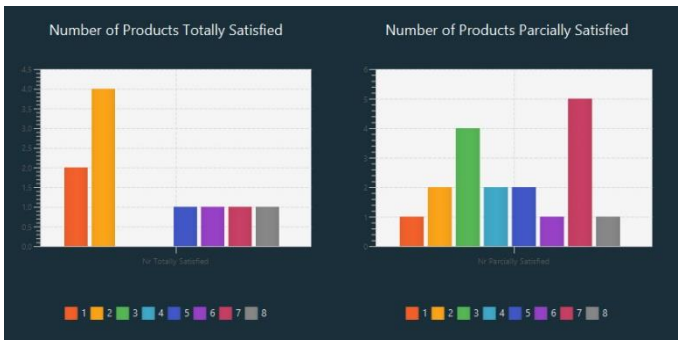


*Figura 4 – Apresentação das Estatísticas*

# Análise de Complexidade

## calculateStatistics

| | Time Complexity |
|---|---|
| `public void calculateStatistics() {` | |
| `for (BasketOrder order : dispatchedList) {` | O(n) |
| `boolean isTotallySatisfied = order.isFullySatisfied();` | O(m) |
| `BasketOrderStatistics basketOrderStats = new BasketOrderStatistics();` | O(1) |
| `ClientStatistics clientStats = mapClientStats.get(order.getClient());` | O(1) |
| `if (clientStats == null) {` | O(1) |
| `clientStats = new ClientStatistics(order.getClient());` | O(1) |
| `mapClientStats.put(order.getClient(), clientStats);` | O(1) |
| `ProducerStatistics producerStats;` | O(1) |
| `List<String> producersId = new ArrayList<>();` | O(1) |
| `HubStatistics hubStats = mapHubStats.get(order.getClient().getNearestHubId());` | O(1) |
| `if (hubStats == null) {` | O(1) |
| `hubStats = new HubStatistics(order.getClient().getNearestHubId());` | O(1) |
| `mapHubStats.put(order.getClient().getNearestHubId(), hubStats);` | O(1) |
| `for (Product product : order.getProducts()) {` | O(m) |
| `Producer currentProducer = product.getProducer();` | O(1) |
| `producerStats = mapProducerStats.get(currentProducer);` | O(1) |
| `if (producerStats == null) {` | O(1) |
| `producerStats = new ProducerStatistics(currentProducer);` | O(1) |
| `if (currentProducer != null) {` | O(1) |
| `producerStats.setNrProdsSoldOut(data.getNrProdsSoldOut(currentProducer));` | O(1) |
| `mapProducerStats.put(currentProducer, producerStats);` | O(1) |
| `if (!producerInBasketOrder(producersId,currentProducer) && currentProducer != null) {` | O(m) |
| `if (isTotallySatisfied)` | O(1) |
| `producerStats.setNrBasketOrdersTotallySatisfied(producerStats.getNrBasketOrdersTotallySatisfied() + 1);` | O(1) |
| `else` | O(1) |
| `producerStats.setNrBasketOrdersPartiallySatisfied(producerStats.getNrBasketOrdersPartiallySatisfied() + 1);` | O(1) |
| `producerStats.addClient(order.getClient());` | O(1) |
| `producerStats.addHub(hubStats.getHub());` | O(1) |
| `isTotallySatisfiedBasketOrder(basketOrderStats, product);` | O(1) |
| `if (currentProducer != null)` | O(1) |
| `hubStats.addProducer(currentProducer);` | O(1) |
| `basketOrderStats.setNrProducers(producersId.size());` | O(1) |
| `basketOrderStats.calculatePercentageOfSatisfaction();` | O(1) |
| `basketOrderStatistics.add(basketOrderStats);` | O(1) |
| `isTotallySatisfied(isTotallySatisfied, clientStats);` | O(1) |
| `clientStats.setNrProducers(producersId.size());` | O(1) |
| `hubStats.addClient(order.getClient());` | O(1) |
| | O(n × m) |

Este método apresenta uma complexidade de O(n × m) sendo n o número de encomendas e m o número de produtos dentro de cada encomenda

O método calculateStatisitcs é o método principal da funcionalidade, é através deste que todas as estatísticas são calculadas. Tal como referido, este executa dois loops, sendo o de fora o loop sobre as encomendas e o de dentro o loop sobre os produtos da encomenda.

Dentro do primeiro loop, o programa faz verificações e criar instâncias a ser usadas dentro do segundo loop. Verifica se a encomenda atual é nula e ignora essa iteração se assim for, verifica se a encomenda atual foi totalmente satisfeita e cria instâncias das classes referidos na figura 3.

Depois, no segundo loop, é aqui que todas as estatísticas são calculadas. Tal como o primeiro for loop, começa por verificar se o produto é nulo e ignora essa iteração se a condição for verdadeira. De seguida inicializa a instância do produtor, e atualiza as suas métricas. Faz o mesmo para o hub e a encomenda.

Após percorrer todos os produtos, o inner loop é finalizado e métricas em relação à encomenda, cliente e hub são atualizadas e guardadas.

## producerInBasketOrder

| | Time Complexity |
|---|---|
| `private boolean producerInBasketOrder(List<String> producersId, Producer producer) {` | |
| `    if (producer == null)` | O(1) |
| `        return false;` | O(1) |
| `    for (String id : producersId)` | O(1) |
| `        if (id.equals(producer.getProducerId()))` | O(1) |
| `            return true;` | O(1) |
| `    producersId.add(producer.getProducerId());` | O(1) |
| `    return false;` | O(1) |
| `}` | O(1) |

Este método tem como objetivo verificar se o Produtor passado por parâmetro já apareceu na encomenda atual. Esta verificação é importante pois, se tal não for verificado, iriemos adicionar quantidades de encomendas totalmente/parcialmente satisfeitas inexistentes.

Por exemplo, uma encomenda que apresenta três produtos, dois destes fornecidos pelo Produtor 'P1' e a outra pelo produto 'P2'. Digamos que esta encomenda foi totalmente satisfeita. Se tal condição não existir, iremos adicionar ao 'P2' uma encomenda totalmente satisfeita e ao 'P1' duas, o que é errado.

## isProductTotallySatisfied

| | Time Complexity |
|---|---|
| `private void isProductTotallySatisfied(BasketOrderStatistics basketOrderStats, Product product) {` | |
| `    if (product.getProducer() == null)` | O(1) |
| `        basketOrderStats.setNrProdsNotSatisfied(basketOrderStats.getNrProdsNotSatisfied() + 1);` | O(1) |
| `    else if (product.getQuantity() == 0)` | O(1) |
| `        basketOrderStats.setNrProdsTotallySatisfied(basketOrderStats.getNrProdsTotallySatisfied() + 1);` | O(1) |
| `    else if (product.getQuantity() > 0)` | O(1) |
| `        basketOrderStats.setNrProdsPartiallySatisfied(basketOrderStats.getNrProdsPartiallySatisfied() + 1);` | O(1) |
| `}` | O(1) |

O método isProductTotallySatisfied é um método bastante simples, apresentando uma complexidade de O(1). No entanto é um método com alguma importância no cálculo das estatísticas para a encomenda. O seu objetivo é verificar e atualizar o número de produtos totalmente/parcialmente/não satisfeitos.

Recebe por parâmetro a instância do objeto de estatística da encomenda e o produto atual. Verifica pelas quantidades o seu nível de satisfação e atualiza os valores.

## isTotallySatisfied

| | Time Complexity |
|---|---|
| `private void isTotallySatisfied(boolean isTotallySatisfied, ClientStatistics clientStats) {` | |
| `    if (isTotallySatisfied)` | O(1) |
| `        clientStats.setNrBasketOrdersTotallySatisfied(clientStats.getNrBasketOrdersTotallySatisfied() + 1);` | O(1) |
| `    else` | O(1) |
| `        clientStats.setNrBasketOrdersPartiallySatisfied(clientStats.getNrBasketOrdersPartiallySatisfied() + 1);` | O(1) |
| `}` | O(1) |

Finalmente, o método isTotallySatisfied, recebe um boolean que contém a informação relativa à satisfação de uma encomenda (totalmente ou parcialmente) e a instância do objeto de estatística de um cliente. Tendo em conta o resultado da variável isTottallySatisfied, o método é responsável por atualizar e guardar os valores.