Projeto Integrador – Sprint 2

Gabriel Gonçalves - 1191296

Tiago Leite - 1191369

João Durães - 1211314

Francisco Bogalho - 1211304

António Bernardo - 1210805

Índice

Introdução

Este projeto tem como objetivo desenvolver um software com um conjunto de funcionalidades que permita gerir uma rede de distribuição de cabazes entre agricultores e clientes.

As funcionalidades requisitadas para a realização deste projeto são:

Em relação ao *sprint* 1:

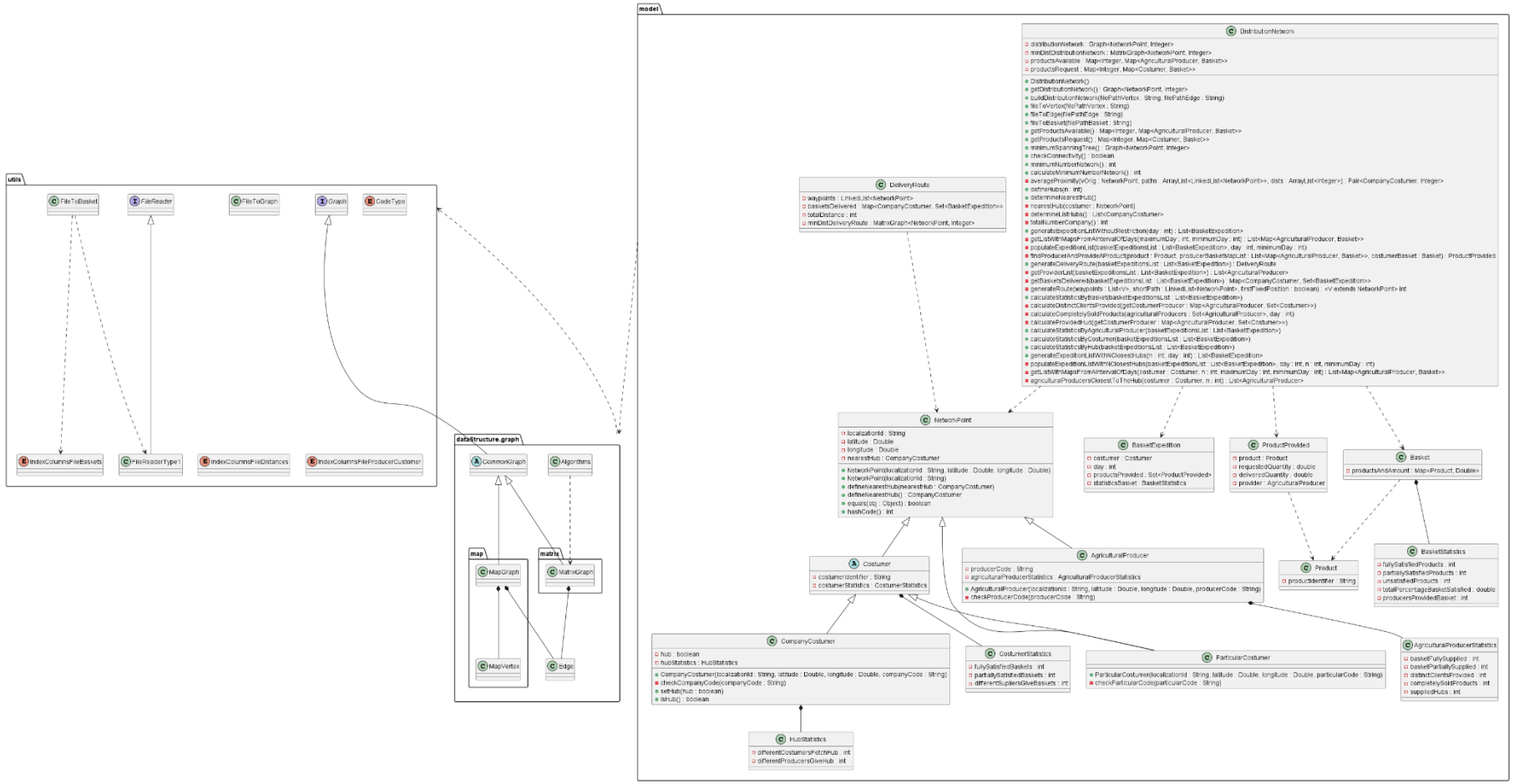
- Construir a rede de distribuição de cabazes a partir da informação fornecida nos ficheiros;
- Verificar se o grafo carregado é conexo e devolver o número mínimo de ligações necessário para nesta rede qualquer cliente/produtor conseguir contactar um qualquer outro;
- Definir os hubs da rede de distribuição, ou seja, encontrar as N empresas mais próximas de todos os pontos da rede (clientes e produtores agrícolas);
- Para cada cliente (particular ou empresa) determinar o hub mais próximo;
- Determinar a rede que conecte todos os clientes e produtores agrícolas com uma distância total mínima;

Em relação ao *sprint* 2:

- Importar a lista de cabazes;
- Gerar uma lista de expedição para um determinado dia que forneça os cabazes sem qualquer restrição quanto aos produtores;
- Gerar uma lista de expedição para um determinado dia que forneça apenas com os N produtores agrícolas mais próximos do hub de entrega do cliente;
- Para uma lista de expedição diária gerar o percurso de entrega que minimiza a distância total percorrida;
- Para uma lista de expedição calcular estatísticas:
    - por cabaz: nº de produtos totalmente satisfeitos, nº de produtos parcialmente satisfeitos, nº de produtos não satisfeitos, percentagem total do cabaz satisfeito, nº de produtores que forneceram o cabaz;
    - por cliente: nº de cabazes totalmente satisfeitos, nº de cabazes parcialmente satisfeitos, nº de fornecedores distintos que forneceram todos os seus cabazes;
    - por produtor: nº de cabazes fornecidos totalmente, nº de cabazes fornecidos parcialmente, nº de clientes distintos fornecidos, nº de produtos totalmente esgotados, nº de hubs fornecidos;
    - por hub: nº de clientes distintos que recolhem cabazes em cada hub, nº de produtores distintos que fornecem cabazes para o hub.

Estas funcionalidades devem ser implementadas da forma mais eficiente possível através do uso de um grafo que deve ser implementado usando a representação mais adequada e garantindo a manipulação indistinta dos clientes/empresas e produtores agrícolas.

Diagrama de classes



Para a estruturação das classes optou-se pela criação de 4 classes designadas por *NetworkPoint*, *CompanyCostumer*, *ParticularCostumer* e *AgriculturalProducer* no package *models*. Estas 3 últimas extendem da classe *NetworkPoint* que representa um ponto da rede.

A classe *CompanyCostumer* representa um cliente do tipo empresarial e a classe ParticularCostumer um cliente do tipo particular. Ambas as classes extendem de uma classe designada *Costumer*. Existe também a classe *AgriculturalProducer* que representa um produtor agrícola.

A estrutura de dados principal adotada é um grafo onde os vértices são do tipo *NetworkPoint* e as arestas do tipo *Integer*, esta estrutura encontra-se na classe *DistributionNetwork*.

Neste sprint criou-se algumas classes entre elas o *Basket* para os cabazes, o *BasketExpedition* para os cabazes de expedição, o *DeliveryRoute* para o percurso de entrega e algumas classes importantes para o cálculo das estatísticas por cabaz, por cliente, por produtor e por hub.

Todos os métodos necessários para a resolução dos exercícios foram desenvolvidos na classe *DistributionNetwork*.

Algoritmos - Explicação e Análise de Complexidade

US307

Classes e estruturas criadas para suportar os novos requisitos

Tivemos a necessidade de criar 2 classes para o desenvolvimento da importação e utilização dos cabazes. A class ***Basket*** e a ***Product***. Na class ***Basket*** criamos um *Map<Product, Double>*, um método para adicionar produto ***addProduct.*** Criamos ainda os *sets* e os *gets* necessários assim como o tradicional método *equalsl.*

```
private Map<Product, Double> productsAndAmount;

/**
 * Construtor para inicicalizar uma instância do objeto Basket.
 */
27 usages
public Basket() { this.productsAndAmount = new HashMap<>(); }

/**
 * Método para adicionar um novo produto ao cabaz. Se o produto
 * existir, a sua quantidade é aumentada.
 *
 * @param product O produto.
 * @param amount  A quantidade do produto a adicionar.
 */
53 usages
public void addProduct(Product product, Double amount) {
    productsAndAmount.merge(product, amount, Double::sum);
}

public Set<Product> getProductsSet() {
    if (productsAndAmount.isEmpty())
        return null;

    return new HashSet<>(productsAndAmount.keySet());
}

/**
 * Método para devolver o tamanho do set de produtos e quantidades
 *
 * @return O tamanho do set de produtos e quantidades
 */
1 usage
public int getProductsSetSize() { return this.productsAndAmount.size(); }

/**
 * Método para devolver a quantidade de um dado produto deste cabaz.
 *
 * @param product O produto do qual queremos a quantidade.
 * @return A quantidade do produto, ou null se o produto não existir.
 */
69 usages
public double getProductQuantity(Product product) {
    double quantity = 0.0;
    Double quantityCheck = productsAndAmount.get(product);

    // Se o produto existir no cabaz, atribuir a sua quantidade existente.
    if (quantityCheck != null) {
        quantity = quantityCheck;
    }
```

Na class ***Product*** esta class representa um produto que tem um identificador e para isso criamos um atributo ***productIdentifier***.

```
/**
 * Identificador do produto.
 */
5 usages
private String productIdentifier;

/**
 * Construtor para inicicalizar uma instância do objeto Product com os atributos productIdentifier e amount.
 *
 * @param productIdentifier Quantidade do produto
 */
public Product(String productIdentifier) { this.productIdentifier = productIdentifier; }
```

Importação da lista de cabazes

Na realização da importação de cabazes, em que utilizamos a informação proveniente de ficheiros e construímos os objetos necessários em memória, criamos a classe FileToBasket para realizar o processamento necessário para isto acontecer. Aqui podemos contar com 6 métodos, em que apenas 1 deles é publico, sendo esse o que é chamado para a realização da importação da informação.

O método público tem o nome de readFileToBasket, e os restantes 5 métodos privados têm o nome:

- convertDayStringInInt;
- convertAmountStringInDouble;
- findNetworkPoint;
- createProductsByHeader;
- createBasket;

O **método convertDayStringInInt**, é responsável por converter um dia representado por uma String, num dia representado pelo tipo de dados int. Desta forma, podemos converter os dias que vêm no ficheiro como String, no tipo de dados adequado.

Este método recebe como parâmetro o valor em String que queremos converter em inteiro, e a linha do ficheiro de onde este valor provem.

```
private static int convertDayStringInInt(String day, int line) {
    int dayValue = 0;

    try {
        dayValue = Integer.parseInt(day);
    } catch (NumberFormatException ex) {
        throw new NumberFormatException("Verifique o valor do dia na linha " + line + " do ficheiro, pois " +
                "encontra-se inválido.");
    }

    return dayValue;
}
```

Este método é não determinístico. Isto acontece porque, o **método parseInt** é não determinístico e tem uma complexidade temporal no pior caso de O(n), logo o método **convertDayStringInInt** também não poderia ser não determinístico, nem ter uma complexidade temporal de pior caso inferior a O(n). O método parseInt percorre cada caracter da String e converte-o para inteiro. Isto torna-se não determinístico devido a que se for encontrado um caracter que não seja realmente um número, é lançada uma exceção. Assim, se tivermos uma letra em vez de um número logo na primeira posição analisada, iremos ter uma complexidade de O(1), devido a ser logo lançada a exceção.

Desta forma, para o atual método (**convertDayStringInInt**), temos também uma complexidade temporal no melhor caso de O(1) (complexidade constante) e no pior caso O(n) (complexidade linear), pois este método

tirando a chamada do método **parseInt** tem uma complexidade O(1), pois ele ou retorna o valor convertido, ou lança uma exceção.

O **método convertAmountStringInDouble**, tem a função de converter um valor em String para um valor do tipo double. Desta forma, torna-se possível converter as quantidades que estão presentes no ficheiro e que vêm como String, no tipo double.

Este método recebe como parâmetro o valor em String a converter, e a linha do ficheiro de onde provém esse valor.

```
private static double convertAmountStringInDouble(String columnValue, int line) {
    double value = 0.0;

    try {
        value = Double.parseDouble(columnValue);
    } catch (NumberFormatException ex) {
        throw new NumberFormatException("Foi encontrado um erro na linha " + line + " do ficheiro. Verifique as " +
                "quantidades existentes para os produtos nessa mesma linha.");
    }

    return value;
}
```

Este método conta com a utilização do método parseDouble, sendo que este método é não determinístico. Isto acontece porque, o método vai a cada caracter de uma String e encontra o seu valor correspondente em decimal. A complexidade temporal neste método (parseDouble) é de O(1), que acontece quando o primeiro caracter a analisar é um valor diferente de um número, lançando assim uma exceção. A complexidade do pior caso será O(n) em que todos os caracteres da String são percorridos. Dado isto, o método **convertAmountStringInDouble** será também não determinístico, tendo também a complexidade de O(1) no melhor caso, e no pior caso uma complexidade de O(n). Este método apenas contém instruções com complexidade O(1), sem contar com a instrução da chamada do outro método.

O **método findNetworkPoint** é responsável por encontrar e devolver o NetWorkPoint de um dado código identificador de um produtor agrícola ou de um cliente.

Este método recebe como parâmetro o grafo da rede de distribuição, que será onde iremos verificar se existe um dado agricultor ou algum cliente com um determinado código identificador. Recebe também como parâmetro o código identificador de um dado cliente ou produtor agrícola, e ainda a linha do ficheiro onde este código identificador se encontra. Ao passar esta última informação, podemos lançar uma exceção quando no ficheiro existe um dado código identificador de um produtor ou cliente que não existe na rede de distribuição.

```
private static NetworkPoint findNetworkPoint(Graph<NetworkPoint, Integer> distributionNetwork,
                                             String codeIdentifier, int line) {
    // Todos os network points da rede
    List<NetworkPoint> networkPointArrayList = distributionNetwork.vertices();

    for (NetworkPoint networkPoint : networkPointArrayList) {
        if (networkPoint instanceof AgriculturalProducer agriculturalProducer) {
            if (agriculturalProducer.getProducerCode().equals(codeIdentifier))
                return agriculturalProducer;
        } else if (networkPoint instanceof Costumer costumer) {
            if (costumer.getCostumerIdentifier().equals(codeIdentifier))
                return costumer;
        }
    }

    throw new NetworkPointNotFoundException("O cliente ou produtor especificado na linha " + line + " do ficheiro, " +
            "não foi encontrado na rede de distribuição. Por favor, verifique se está correto.");
}
```

Assumindo que o clone do arraylist feita pela instrução distributionNetwork.vertices() é O(n). Este método é não determinístico, devido a que dada uma lista enorme de pontos de rede, podemos encontrar o ponto desejado logo no início dessa lista ou apenas no fim, visto que quando esse ponto é encontrado, ele é retornado e o método acaba. Ou seja, se o ponto de rede desejado estiver no final da lista, iremos ter de percorrer todos esses pontos, tendo então a pior complexidade temporal de O(n + n). No melhor caso de complexidade temporal temos O(n).

O **método createProductsByHeader** permite criar todos os produtos que venham declarados no cabeçalho do ficheiro. Este método cria os objetos do tipo *Product* e coloca-os como valor num mapa em que a chave indica o seu índice no *array*, que contém os dados de uma dada linha do ficheiro. Desta forma, torna-se possível criar produtos em função do que seja necessário e não apenas os 12 produtos que estão presentes nos ficheiros de importação fornecidos, fazendo assim com que a leitura do ficheiro seja muito mais flexível.

Este método recebe como parâmetro um *array* com os dados presentes no cabeçalho do ficheiro.

```
private static Map<Integer, Product> createProductsByHeader(String[] header) {
    Map<Integer, Product> productMap = new HashMap<>();

    for (int pos = IndexColumnsFileBaskets.START_PRODUCT_INDEX.getValue(); pos < header.length; pos++) {
        productMap.put(pos, new Product(header[pos]));
    }

    return productMap;
}
```

Este algoritmo é determinístico, pois ele realiza sempre n ciclos para criar os n produtos que estão presentes no *array* passado por parâmetro. Sendo então este o único ciclo no método e todas as outras instruções tem complexidade constante, este algoritmo apresenta uma complexidade temporal de O(n), complexidade linear.

Se formos analisar o caso deste projeto em que apenas iremos ter sempre 12 produtos, podemos dizer que a complexidade deste método é O(1), pois ela irá realizar sempre o ciclo apenas 12 vezes.

O **método createBasket** é responsável por criar um cabaz com os dados provenientes de uma dada linha do ficheiro. O cabaz conta com a informação dos produtos e respetivas quantidades existentes. Apenas a informação sobre produtos que contenham quantidade maior de 0 é armazenada.

Este método recebe por parâmetro os dados da linha do ficheiro dentro de um *array*, um mapa que contem os objetos dos produtos criados no método falada anteriormente, e recebe ainda a linha de onde os dados provêm.

```
private static Basket createBasket(String[] data, Map<Integer, Product> productMap, int line) {
    Basket basket = new Basket();

    for (int pos = IndexColumnsFileBaskets.START_PRODUCT_INDEX.getValue(); pos < data.length; pos++) {
        double productAmount = convertAmountStringInDouble(data[pos], line);
        if (productAmount > 0.0)
            basket.addProduct(productMap.get(pos), productAmount);
    }

    return basket;
}
```

Este método conta com um ciclo inicial que contem uma complexidade temporal de O(n), pois ele vai percorrer a quantidade de cada produto existente no ficheiro, na linha a tratar. De seguida temos a chamada ao método **convertAmountStringInDouble**, em que vimos anteriormente que este método era não determinístico, fazendo então com que este atual método também não o seja. Vimos também que este método tinha uma complexidade no pior caso de O(n), fazendo então com que o atual método passe a ter uma complexidade de O(n*l) (l porque o tamanho da String não tem necessariamente n tamanho), pois este método irá ser chamado n vezes. Após isto, temos uma condição e de seguida uma chamada do método *addProduct* da classe *Basket*. Esse método apresenta uma complexidade temporal de O(1), pois apenas irá guardar os valores num *HashMap*. Como os produtos estão guardados num *HashMap*, a complexidade temporal da instrução productMap.get(pos) será de O(1).

Desta forma, podemos então concluir que este algoritmo é não determinístico e contém uma complexidade temporal de O(n*l).

O **método readFileToBasket** é o método principal responsável pela conversão do ficheiro em cabazes, em que ele mapeia os cabazes criados para os respetivos mapas correspondentes, o mapa dos cabazes dos produtores ou o mapa dos cabazes dos clientes. Ele começa por criar uma lista com todos as linhas do ficheiro, em que posteriormente percorre cada linha armazenada na lista e faz o tratamento dos dados lá existentes.

Este método recebe como parâmetro a rede de distribuição, onde estão contidos todos os pontos da rede, recebe o mapa que armazena os cabazes com os produtos disponíveis para serem distribuídos pelos clientes,

recebe também o mapa que contém os cabazes requisitados pelos clientes, e por fim, recebe o caminho para o ficheiro que contem toda a informação acerca dos cabazes.

```
public static void readFileToBasket(Graph<NetworkPoint, Integer> distributionNetwork,
                                     Map<Integer, Map<AgriculturalProducer, Basket>> productsAvailable,
                                     Map<Integer, Map<Costumer, Basket>> productsRequest,
                                     String filePath
) {

    if (distributionNetwork == null || distributionNetwork.numVertices() == 0)
        return;

    try {

        List<String[]> contentFile = new FileReaderType1().readFile(filePath, ignoreHeader: false);
        String[] header = contentFile.get(0);
        contentFile.remove( index: 0);

        // Cria os objetos Produto com os nomes dos produtos fornecidos no cabeçalho
        Map<Integer, Product> productMap = createProductsByHeader(header);

        int line = 2;
        for (String[] columns : contentFile) {
            // Se as linhas que sucedem o cabeçalho não tiver o mesmo número de colunas, existe erro.
            if (columns.length != header.length) {
                throw new InvalidFileLineException(line);
            }

            // Identificador
            String clientOrProductor = columns[IndexColumnsFileBaskets.COSTUMERS_PRODUCERS.getValue()];
            int day = convertDayStringInInt(columns[IndexColumnsFileBaskets.DAY.getValue()], line);
            Basket basket = createBasket(columns, productMap, line);
            NetworkPoint networkPoint = findNetworkPoint(distributionNetwork, clientOrProductor, line);

            // Apenas adiciona o cliente ou o produtor se o cabaz para este dia tiver produtos.
            if (basket.getProductsSetSize() != 0) {
                // Verifica que é produtor ou cliente
                if (clientOrProductor.toUpperCase().charAt(0) == CodeType.PRODUCER.getCharacter()) {
                    if (productsAvailable.get(day) == null)
                        productsAvailable.put(day, new HashMap<>());
                    productsAvailable.get(day).put((AgriculturalProducer) networkPoint, basket);
                } else {
                    if (productsRequest.get(day) == null)
                        productsRequest.put(day, new HashMap<>());
                    productsRequest.get(day).put((Costumer) networkPoint, basket);
                }
            }

            line++;
        }

    } catch (FileNotFoundException ex) {
        throw new IllegalArgumentException("Não existe nenhum ficheiro no caminho indicado.");
    }

}
```

Este algoritmo é logo à partida não determinístico, pois ele vai fazer o uso dos algoritmos analisados anteriormente e existem algoritmos que não são determinísticos. Vamos então agora analisar qual será o pior caso e o melhor caso na complexidade temporal.

Este algoritmo terá no melhor caso uma complexidade temporal de O(1), devido a que se for encontrado ou um erro no caminho do ficheiro, ou até mesmo um erro logo na primeira linha do ficheiro devido a que essa informação passe pelos métodos estudados anteriormente, algo pode não estar de acordo com o que é suposto e serem lançados erros em função disso.

Agora olhando para o pior caso, vamos analisar todos os ciclos e chamadas de métodos que são feitas por parte deste método. Este método utiliza o método *readFile* da classe FileReaderType1, sendo que este possui uma complexidade temporal de O(n*s), porque ele irá correr todas as linhas do ficheiro e armazená-las numa lista n vezes, mas para essas n vezes, ele irá percorrer toda a *String* que é a linha do ficheiro e remover todas as aspas (s é o tamanho da *String*). Depois disso, ele irá percorrer toda a *String* novamente de forma a separar as palavras pelas vírgulas (novamente um ciclo que é feito s vezes, ficando s+s=2s).

Depois é feita uma operação de *get* num *Arraylist* que tem uma complexidade de O(1). Já o remove que é feito nessa lista de seguida, tem uma complexidade O(n) como foi estudado nas aulas de ESINF.

Após isto, é chamado o método **createProductsByHeader** já analisado anteriormente.

De seguida, é realizado um ciclo para analisar cada um dos dados provenientes do ficheiro, tenho este ciclo uma complexidade de O(n). Posteriormente a este ciclo, são chamados os métodos *convertDayStringInInt*, *createBasket* e *findNetworkPoint*, já analisados anteriormente.

Depois temos uma condição em que esta chama um método com complexidade O(1), que é saber o tamanho de um *HashMap*, e como estudado nas aulas, este método representa essa complexidade.

De seguida a única operação realizada em que não é O(1) dentro desta condição, é a chamada *clientOrProductor.toUpperCase().charAt(0)*. Este método (*toUpperCase()*) representa uma complexidade O(n) em que ele percorre toda a *String* colocando cada caracter em maiúscula. (Os *put* e *get* em HashMap têm complexidade O(1), como estudado nas aulas)

Desta forma, podemos analisar a complexidade temporal no pior caso de todo o método, que é O(n*s+n*p), em que s são os tamanhos das *Strings* a converter e p o número de pontos na rede.

US308

Nesta *User Story* era pedido que fosse possível gerar uma lista de expedição que fornecesse os cabazes, sem qualquer restrição quanto aos produtores, para um determinado dia. Para realizar esta parte, foram criadas mais duas classes, a classe *BasketExpedition* e a classe *ProductProvided*.

A classe *BasketExpedition* representa a expedição de um dado cabaz, em que nesta classe constam os dados de um cliente, o dia em que o cabaz é expedido e os produtos que o cabaz contém. Esta classe possui ainda um método para sabermos onde este cabaz irá ser entregue, sendo o nome dele *getDeliveryLocation*.

```java
public class BasketExpedition {

    /**
     * O cliente para quem se destina o cabaz.
     */
    5 usages
    private Costumer costumer;

    /**
     * O dia em que o cabaz foi expedido.
     */
    4 usages
    private int day;

    /**
     * Lista dos produtos fornecidos para este cabaz.
     */
    9 usages
    private Set<ProductProvided> productsProvided;


public CompanyCostumer getDeliveryLocation() {
    return costumer.getNearestHub();
}
```

A classe *ProductProvided* representa um produto fornecido, em que aqui encontramos os atributos para guardar o produto a fornecido, a quantidade pedida pelo cliente, a quantidade entregue pelo produtor, e quem foi o produtor que forneceu o produto.

```java
public class ProductProvided {

    /**
     * O produto requerido.
     */
    4 usages
    private Product product;

    /**
     * Quantidade requerida do produto.
     */
    5 usages
    private double requestedQuantity;

    /**
     * Quantidade entregue do produto.
     */
    7 usages
    private double deliveredQuantity;

    /**
     * O fornecedor do produto.
     */
    8 usages
    private AgriculturalProducer provider;
```

De forma a alcançar o objetivo pedido, para além das classes, foram criados os seguintes métodos:

- *generateExpeditionListWithoutRestriction*;
- *getListWithMapsFromAIntervalOfDays*;
- *populateExpeditionList*;
- *findProducerAndProvideAProduct*.

Estes métodos foram implementados na classe DistributionNetwork, sendo esta a classe que contém a informação acerca dos pontos de rede e toda a informação que gire à volta da rede de distribuição.

Iremos então agora analisar estes 4 métodos referidos acima, dizendo a sua finalidade e a complexidade temporal correspondente, começando pelo método *findProducerAndProvideAProduct*.

O **método *findProducerAndProvideAProduct***, é responsável por procurar um produtor para fornecer um dado produto. Este produto ao ser fornecido, atualiza a informação dos mapas que contem a informação dos cabazes pedidos e dos cabazes fornecidos. Este método recebe por parâmetro um produto, que é o produto pedido pelo cliente, recebe também uma lista com os mapas de cabazes fornecidos em cada dia por produtores, e por fim, recebe o cabaz total requerido pelo cliente. No final, o método devolve o produto fornecido.

```
private ProductProvided findProducerAndProvideAProduct(
        Product product,
        List<Map<AgriculturalProducer, Basket>> producerBasketMapList,
        Basket costumerBasket) {

    double costumerRequestedQuantity = costumerBasket.getProductQuantity(product);
    ProductProvided productProvided = new ProductProvided(product, costumerRequestedQuantity, deliveredQuantity: 0,
            provider: null);
    boolean productFound = false;

    // Para cada mapa, verificar se existe um produtor que contém o produto desejado.
    Iterator<Map<AgriculturalProducer, Basket>> producerBasketMapsIterator = producerBasketMapList.iterator();
    while (producerBasketMapsIterator.hasNext() && !productFound) {
        Map<AgriculturalProducer, Basket> producerBasketMap = producerBasketMapsIterator.next();

        // Para cada produtor, verificar se este contém o produto requerido pelo cliente em cabaz e scom stock.
        Iterator<AgriculturalProducer> providersIterator = producerBasketMap.keySet().iterator();
        while (providersIterator.hasNext() && !productFound) {
            AgriculturalProducer provider = providersIterator.next();

            Basket producerBasket = producerBasketMap.get(provider);
            double producerAvailableQuantity = producerBasket.getProductQuantity(product);

            // Se o produto tiver o produto em quantidades acima de 0
            if (producerBasket.hasProduct(product) && producerAvailableQuantity > 0.0) {
                // Diferença de quantidade que o produtor tem com a quantidade requerida pelo cliente.
                double quantityDifference = producerAvailableQuantity - costumerRequestedQuantity;
                // A quantidade do produto a ser entregue.
                double deliveredQuantity = quantityDifference < 0 ? producerAvailableQuantity : costumerRequestedQuantity;
                // Subtrai a quantidade deste produto agora a receber ao cabaz pedido pelo cliente.
                costumerBasket.subtractProductQuantity(product, deliveredQuantity);
                // Subtrai a quantidade fornecida pelo produtor ao cabaz do produtor.
                producerBasket.subtractProductQuantity(product, deliveredQuantity);
                productProvided = new ProductProvided(product, costumerRequestedQuantity, deliveredQuantity, provider);
                productFound = true;
            }
        }
    }

    return productProvided;

}
```

Analisando a complexidade temporal deste algoritmo, temos que no início antes do ciclo, apenas são executadas instruções com complexidade O(1), devido ao *getProductQuantity* ser um *get* a um *HashMap*.

Este algoritmo é não determinístico, como podemos ver pelos dois ciclos existentes, quando um dado produto for encontrado na lista de produtos disponibilizados pelos fornecedores, o método termina. Desta forma, temos que o produto pode ser logo encontrado, ou pode até nem ser encontrado.

Este primeiro ciclo tem uma particularidade que é, ele é executado no pior cenário 3 vezes, tomando em conta o enunciado, pois ele contém os mapas dos três dias em que é possível expedir produtos, pois os produtos colocados na rede pelos produtores, ficam no máximo 3 dias disponíveis, o dia em que são colocados e os 2

dias seguintes. Desta forma teremos então uma complexidade temporal de O(1), pois no problema analisado do ponto de vista do enunciado, apenas será dessa forma, caso contrário, seria uma complexidade de O(n) (no pior caso).

O segundo ciclo, já depende do número de produtores que disponibilizaram produtos naquele determinado dia, tendo assim uma complexidade de no pior caso O(n), no melhor seria o produto ser encontrado logo no primeiro produtor (as instruções dentro do ciclo seriam executadas apenas 1 vez).

Agora, antes da condição, são executados *gets* a *HashMaps*, sendo então uma complexidade O(1).

Dentro da condição, temos também apenas instruções com complexidade O(1), pois apenas são feitas operações aritméticas e utilização de *gets* a *HashMaps*.

Analisando agora o algoritmo de um ponto de vista geral, temos então que ele não é determinístico. O seu melhor caso é O(1). Analisando o pior caso do ponto de vista do enunciado em que os produtos apenas ficam 3 dias disponíveis, temos uma complexidade temporal de O(3*n), que fica O(n). Do ponto de vista do algoritmo sem conhecimento do enunciado, temos uma complexidade de O(n*m), em que 'm' é o número de mapas de 'm' dias, e 'n' é o número de produtores que disponibilizaram cabazes num dado dia 'm'.

O **método *getListWithMapsFromAIntervalOfDays*** é utilizado para construir uma lista com os mapas de cabazes disponibilizados à rede de distribuição pelos produtores, em que esses mapas pertençam a um dado intervalo de dias. Este é o método que cria a lista falada anteriormente, em que na resolução de apenas o enunciado, terá 3 mapas contidos. Este método recebe como parâmetro o dia máximo e o dia mínimo que será o intervalo de dias em que iremos buscar os cabazes fornecidos pelos produtores. No final ele devolve a lista que contém os mapas dos cabazes fornecidos pelos produtores de cada um desses dias do intervalo.

```
private List<Map<AgriculturalProducer, Basket>> getListWithMapsFromAIntervalOfDays(int maximumDay, int minimumDay) {
    if (maximumDay < minimumDay)
        throw new IllegalArgumentException("O dia mais alto não pode ser menor do que o dia mais baixo.");

    if (minimumDay < 1)
        throw new IllegalArgumentException("O dia mais baixo não pode ser menor do que 1.");

    List<Map<AgriculturalProducer, Basket>> producerBasketMapList = new ArrayList<>();
    // Para cada dia do intervalo.
    for (int actualDay = maximumDay; actualDay >= minimumDay; actualDay--) {
        producerBasketMapList.add(this.productsAvailable.get(actualDay));
    }

    return producerBasketMapList;
}
```

Este método é determinístico, pois o ciclo irá sempre acontecer as 'n' vezes que será o intervalo entre o dia mais baixo e o dia mais alto. Dentro do ciclo, a adição de um mapa na lista e o *get* ao *HashMap* serão de complexidade O(1), como estudado nas aulas. Desta forma, podemos concluir que o algoritmo apresenta uma complexidade temporal de O(n), pois as instruções dentro do ciclo irão acontecer sempre n vezes.

Já o **método *populateExpeditionList*** é responsável por popular a lista de expedição de cabazes para um dado dia. Este método recebe como parâmetro a lista de expedição de cabazes a popular, o dia para o qual esta lista de expedição é criada, e o último dia ao qual ainda podemos ir buscar produtos fornecidos.

```
private void populateExpeditionList(List<BasketExpedition> basketExpeditionsList, int day, int minimumDay) {
    // Mapa com os cabazes requeridos para o dia day.
    Map<Costumer, Basket> costumerBasketMap = this.productsRequest.get(day);
    // Lista com os mapas de cabazes fornecidos em cada dia por produtores, no intervalo de dias dado.
    List<Map<AgriculturalProducer, Basket>> producerBasketMapList = getListWithMapsFromAIntervalOfDays(day,
            minimumDay);

    Set<Costumer> costumerSet = costumerBasketMap.keySet();

    // Será necessário procurar cada produto que cada cliente deseja.
    for (Costumer costumer : costumerSet) {
        Basket costumerBasket = costumerBasketMap.get(costumer);
        BasketExpedition basketExpedition = new BasketExpedition(costumer, day);

        // Para cada produto que se encontra requerido no cabaz do cliente.
        for (Product product : costumerBasket.getProductsSet()) {
            ProductProvided productProvided = findProducerAndProvideAProduct(product, producerBasketMapList,
                    costumerBasket);

            if (productProvided != null)
                basketExpedition.addProductProvided(productProvided);

        }

        basketExpeditionsList.add(basketExpedition);

    }

}
```

Este algoritmo podemos logo à partida dizer que ele não é determinístico, pois ele utiliza o método *findProducerAndProvideAProduct* abordado anteriormente, e como esse é não determinístico, este também não será.

Este método, na primeira linha contém uma instrução O(1) e logo de seguida utiliza o método abordado anteriormente a este, em que foi analisado que tinha uma complexidade O(n).

Depois temos dois ciclos um dentro do outro, em que o primeiro será O(c) e o segundo O(p). Entre estes dois ciclos temos instruções O(1), em que uma é um *get* a um *HashMap* e outra a instanciação de um objeto. Dentro do segundo ciclo, temos uma chamada ao método *findProducerAndProvideAProduct*, já analisado anteriormente. Depois disso temos uma condição que contem uma instrução dentro que é O(1), visto que o método lá utilizado apenas realiza uma adição do objeto no *Set*. Após este ciclo, é feita a adição do cabaz expedido na lista de expedição de cabazes, operação com complexidade temporal O(1).

Desta forma, já podemos então analisar a complexidade temporal do algoritmo no melhor e pior caso. Assumindo que estamos a olhar para a complexidade de acordo com o enunciado, para o método *findProducerAndProvideAProduct*, temos que para o melhor caso é O(n + c * p), sendo 'c' o número de clientes, 'p' o número de produtos. Já para o pior caso, este algoritmo terá O(n + c * p * a), onde 'a' o número de produtores agrícolas. Se para o pior caso se utilizássemos o método *findProducerAndProvideAProduct* sem ter o enunciado em conta, o pior caso de complexidade temporal seria O(n + c * p * a * m), em que 'm' é o número de mapas de 'm' dias.

Por fim, o **método *generateExpeditionListWithoutRestriction*** tem a função de gerar uma lista de expedição de cabazes para um determinado dia, sem restrições enquanto os produtores. Este método recebe

como parâmetro o dia para o qual a lista de expedição sem restrições será gerada. No final, o método devolve uma lista preenchida com cabazes de expedição.

```
public List<BasketExpedition> generateExpeditionListWithoutRestriction(int day) {

    if (productsRequest.get(day) == null || productsAvailable.isEmpty())
        return null;

    // Determina os hubs mais próximos dos clientes
    determineNearestHub();

    List<BasketExpedition> basketExpeditionsList = new ArrayList<>();

    int minimumDay = Math.max(day - MAX_DAYS_BEFORE_BASKET_EXPIRE, 1);

    populateExpeditionList(basketExpeditionsList, day, minimumDay);

    return basketExpeditionsList;
}
```

Neste método, temos que ele é não determinístico, devido a ele utilizar outros métodos também não determinísticos. Aqui existem instruções de O(1), e todas as instruções de chamada a métodos que não são O(1), já foram analisados anteriormente. O único método na referido neste relatório é o *determineNearestHub*, em que ele já foi analisado no *sprint* anterior e determinado que o melhor caso era O(V^2*w^2), e o pior caso era O((V (V + ((V+E) log(V)) + V) + V+V+V) + V log(V) + V + (V((V+E) log(V)))).

Desta forma, podemos analisar este método e chegar à conclusão que a complexidade temporal deste método no melhor caso é de O((V (V + ((V+E) log(V)) + V) + V+V+V) + V log(V) + V + (V((V+E) log(V))) + n + c * p)), e no pior caso O(v^2*w^2 + n + c * p * a * m).

US309

Nesta *User Story* era pedido que fosse possível gerar uma lista de expedição para um determinado dia que forneça apenas com os N produtores agrícolas mais próximos do hub de entrega do cliente. Desta forma, para alcançar o objetivo pedido, foram criados os seguintes métodos:

- *agriculturalProducersClosestToTheHub*
- *getListWithMapsFromAIntervalOfDays*
- *populateExpeditionListWithNClosestHubs*
- *generateExpeditionListWithNClosestHubs*

Estes métodos foram implementados na classe DistributionNetwork, sendo esta a classe que contém a informação acerca dos pontos de rede e toda a informação que gire à volta da rede de distribuição.

Iremos então agora analisar estes 4 métodos referidos acima, dizendo a sua finalidade e a complexidade temporal correspondente, começando pelo método *agriculturalProducersClosestToTheHub*.

O método agriculturalProducersClosestToTheHub é responsável por encontrar os n Produtores Agrícolas que estão mais próximos do hub mais próximo de um determinado cliente, com base nas distâncias entre o hub e os outros vértices de uma distributionNetwork representada por um Grafo. Ele faz isso encontrando o caminho mais curto entre o hub e todos os outros vértices da rede usando o método shortestPaths da classe Algorithms. Em seguida, ele classifica os AgriculturalProducers com base em sua distância até o hub e retorna os primeiros n AgriculturalProducers nessa lista ordenada.

```
private List<AgriculturalProducer> agriculturalProducersClosestToTheHub(Costumer costumer, int n) {
    CompanyCostumer hub = costumer.getNearestHub();
    List<NetworkPoint> vertices = distributionNetwork.vertices();

    // Cria uma lista para receber todos os caminhos entre dois vértices
    ArrayList<LinkedList<NetworkPoint>> paths = new ArrayList<>();
    // Cria uma lista para receber todos os valores ponderados dos caminhos
    ArrayList<Integer> dists = new ArrayList<>();

    // Calcula o caminho mais curto entre um vértice e todos os outros vértices
    Algorithms.shortestPaths(distributionNetwork, hub, Integer::compare, Integer::sum, zero: 0, paths, dists);

    List<Pair<AgriculturalProducer, Integer>> agriculturalProducerDistanceToHub = new ArrayList<>();

    for (NetworkPoint networkPoint : vertices) {
        // Verifica se o vértice inicial é igual ao vértice final e se o vértice final é um companyCostumer
        if ((networkPoint instanceof AgriculturalProducer)) {
            // Atribuir à variável a distância entre dois vértices
            int distance = dists.get(distributionNetwork.key(networkPoint));

            agriculturalProducerDistanceToHub.add(new Pair<>((AgriculturalProducer) networkPoint, distance));
        }
    }

    Collections.sort(agriculturalProducerDistanceToHub);

    List<AgriculturalProducer> agriculturalProducers = new ArrayList<>();

    for (int i = 0; i < agriculturalProducerDistanceToHub.size(); i++) {
        if (n > i) {
            agriculturalProducers.add(agriculturalProducerDistanceToHub.get(i).getKey());
        }
    }

    return agriculturalProducers;
}
```

A complexidade temporal deste método depende da complexidade de tempo do método shortestPaths e do método sort.

O método shortestPaths encontra os caminhos mais curtos entre o hub e todos os outros vértices no grafo. A complexidade de tempo do algoritmo de Dijkstra é O(|E| + |V| log |V|), onde |E| é o número de arestas e |V| é o número de vértices do grafo.

O método sort usa a interface Comparable para classificar os AgriculturalProducers com base na sua distância até o hub. Como a implementação Comparable tem uma complexidade de tempo de O(1), então a complexidade de tempo do método sort será O(n log n), onde n é o número de AgriculturalProducers.

Portanto, a complexidade de tempo geral do método é O(|E| + |V| log |V| + n log n).

Este método é determinístico, pois ele começa por encontrar os caminhos mais curtos entre o hub mais próximo de um determinado cliente e todos os outros vértices em uma rede de distribuição usando o algoritmo de Dijkstra, que é determinístico. Em seguida, ele classifica os AgriculturalProducers com base em sua distância até o hub usando o método de classificação, que também é determinístico. Finalmente, ele retorna os n primeiros AgriculturalProducers nesta lista ordenada.

O método *getListWithMapsFromAIntervalOfDays* obtém uma lista de mapas que associam AgriculturalProducers com objetos Basket para um determinado intervalo de dias, calculado a partir de dois dias passado pelos parâmetros maximumDay e MinimumDay.

Para cada dia do intervalo, o método obtém um mapa que associa AgriculturalProducers com objetos Basket do campo productsAvailable e filtra esse mapa para incluir apenas os AgriculturalProducers no parâmetro AgricultureProducerList. Em seguida, ele adiciona esse mapa filtrado à lista ProducerBasketMapList. Finalmente, o método retorna a lista ProducerBasketMapList.

```
private List<Map<AgriculturalProducer, Basket>> getListWithMapsFromAIntervalOfDays(Costumer costumer, int n, int maximumDay, int minimumDay) {
    List<Map<AgriculturalProducer, Basket>> producerBasketMapList = new ArrayList<>();

    List<AgriculturalProducer> agriculturalProducerList = agriculturalProducersClosestToTheHub(costumer, n);

    for (int actualDay = maximumDay; actualDay >= minimumDay; actualDay--) {
        Map<AgriculturalProducer, Basket> agriculturalProducerBasketMap = this.productsAvailable.get(actualDay);
        Map<AgriculturalProducer, Basket> resultAgriculturalProducerBasketMap = new HashMap<>();

        for (AgriculturalProducer agriculturalProducer : agriculturalProducerList) {
            Basket basket = agriculturalProducerBasketMap.get(agriculturalProducer);

            if (basket != null) {
                resultAgriculturalProducerBasketMap.put(agriculturalProducer, basket);
            }
        }

        producerBasketMapList.add(resultAgriculturalProducerBasketMap);
    }

    return producerBasketMapList;
}
```

Este método tem uma complexidade temporal igual á do método *agriculturalProducersClosestToTheHub* vezes o número de dias e o número de *AgriculturalProducers*, porque chama o método *agriculturalProducersClosestToTheHub* para a combinação de todos os dias do intervalo com todos *AgriculturalProducers* na *agriculturalProducerList,* ou seja, O(n * d + |E| + |V| log |V| + n log n) onde n é o número de AgriculturalProducers em agriculturalProducerList e d é o número de dias.

O método *populateExpeditionListWithNClosestHubs* preenche uma lista de objetos BasketExpedition com informações sobre os produtos solicitados pelos clientes em um determinado dia, e os AgriculturalProducers que podem fornecer esses produtos. Para isso o método considera os n Produtores Agrícolas mais próximos do hub mais próximo de cada Cliente.

Este método utiliza o método *findProducerAndProvideAProduct* que já foi explicado anteriormente na *User Story* 308.

O método itera sobre os Costumers neste mapa e, para cada Costumer, chama o método getListWithMapsFromAIntervalOfDays para obter uma lista de mapas que associam AgriculturalProducers com objetos Basket para o intervalo de dias entre day e MinimumDay.

Para cada produto no cabaz do cliente atual, o método chama o método findProducerAndProvideAProduct para localizar um AgriculturalProducer que fornece o produto e adiciona esse objeto ProductProvided a um objeto BasketExpedition para o cliente atual. Finalmente, adiciona o objeto BasketExpedition à lista basketExpeditionList.

Este método tem complexidade temporal igual ao método *getListWithMapsFromAIntervalOfDays* vezes o número de produtos no cabaz do cliente vezes o número de clientes, ou seja, O(c * p * (n * d + |E| + |V| log |V| + n log n)) onde p é o número de produtos no cabaz do cliente e c é número de clientes.

```
private void populateExpeditionListWithNClosestHubs(List<BasketExpedition> basketExpeditionList, int day, int n, int minimumDay) {
    Map<Costumer, Basket> costumerBasketMap = this.productsRequest.get(day);

    Set<Costumer> costumerSet = costumerBasketMap.keySet();

    for (Costumer costumer : costumerSet) {

        List<Map<AgriculturalProducer, Basket>> producerBasketMapList = getListWithMapsFromAIntervalOfDays(costumer, n, day, minimumDay);

        Basket costumerBasket = costumerBasketMap.get(costumer);
        BasketExpedition basketExpedition = new BasketExpedition(costumer, day);

        // Para cada produto que se encontra requerido no cabaz do cliente.
        for (Product product : costumerBasket.getProductsSet()) {
            ProductProvided productProvided = findProducerAndProvideAProduct(product, producerBasketMapList, costumerBasket);

            if (productProvided != null)
                basketExpedition.addProductProvided(productProvided);

        }

        basketExpeditionList.add(basketExpedition);
    }

}
```

Por fim o método *generateExpeditionListWithNClosestHubs* gera uma lista de expedição para um determinado dia que forneça apenas com os N produtores agrícolas mais próximos do hub de entrega do cliente.

Este método começa por verificar se o dia existe, depois verifica se os produtos disponíveis estão vazios e também verifica se o número de hubs é positivo.

Depois chama o método *determineNearestHub* para determinar os hubs mais próximos dos clientes.

Seguidamente calcula o minimunDay e chama populateExpeditionListWithNClosestHubs.

Este método acaba por devolver uma lista de basketExpedition com informação sobre os produtos solicitados pelos clientes num determinado dia.

```
public List<BasketExpedition> generateExpeditionListWithNClosestHubs(int n, int day) {

    if (productsRequest.get(day) == null || productsAvailable.isEmpty() || n <= 0) {
        return null;
    }

    // Determina os hubs mais próximos dos clientes
    determineNearestHub();

    List<BasketExpedition> basketExpeditionList = new ArrayList<>();

    int minimumDay = Math.max(day - MAX_DAYS_BEFORE_BASKET_EXPIRE, 1);

    populateExpeditionListWithNClosestHubs(basketExpeditionList, day, n, minimumDay);

    return basketExpeditionList;
}
```

A complexidade de tempo deste método é igual á soma da do método *determineNearestHub* e da do método *populateExpeditionListWithNClosestHubs*.

US310

Na **US310** é pretendido gerar o percurso de entrega que minimiza a distância total percorrida para uma lista de expedição.

Desta forma, para alcançar o objetivo pedido, foi criado um *model* denominado ***DeliveryRoute***.

Este *model* é constituído pelos seguintes atributos:

- *waypoints* - Lista com os pontos de passagem do percurso (produtores e hubs);
- *basketsDelivered* - Mapa que contêm os cabazes entregues em cada hub. A chave do mapa é o hub e os valores uma lista do tipo *Set* de cabazes;
- *totalDistance* - Distância total entre todos os pontos do percurso;
- *minDistDeliveryRoute* - Grafo com a distância mínima dos pontos de passagem do percurso.

```
public class DeliveryRoute {

    /**
     * Lista com os pontos de passagem do percurso (produtores e hubs).
     */
    6 usages
    private LinkedList<NetworkPoint> waypoints;

    /**
     * Mapa que contêm os cabazes entregues em cada hub.
     */
    2 usages
    private Map<CompanyCostumer, Set<BasketExpedition>> basketsDelivered;

    /**
     * Distância total entre todos os pontos do percurso.
     */
    4 usages
    private int totalDistance;

    /**
     * Grafo com a distância mínima dos pontos de passagem do percurso.
     */
    2 usages
    private MatrixGraph<NetworkPoint, Integer> minDistDeliveryRoute;
```

Neste mesmo *model* existe ainda 2 funções bastante importantes.

A função *getBasketsDeliveredHub* retorna para um determinado *hub* passado por parâmetro a lista de cabazes a ser entregues.

A função *distanceBetweenTwoWaypoints* retorna a distância entre dois pontos de passagem do percurso. Caso algum dos dois pontos não pertençam ao percurso é lançado uma exceção do tipo *IllegalArgumentException*.

```
/**
 * Função para retornar uma lista com os cabazes a ser entregues num hub.
 *
 * @param hub Hub.
 * @return Lista com os cabazes a ser entregues num hub.
 */
1 usage
public Set<BasketExpedition> getBasketsDeliveredHub(CompanyCostumer hub) {
    return basketsDelivered.get(hub);
}

/**
 * Função que retorna a distância entre dois pontos de passagem do percurso.
 *
 * @param point1 Ponto 1 de passagem do percurso.
 * @param point2 Ponto 2 de passagem do percurso.
 * @return Distância entre dois pontos de passagem do percurso.
 */
2 usages
public int distanceBetweenTwoWaypoints(NetworkPoint point1, NetworkPoint point2) {
    if (waypoints.contains(point1) && waypoints.contains(point1)) {
        return minDistDeliveryRoute.edge(point1, point2).getWeight();
    } else {
        throw new IllegalArgumentException("O ponto indicado não pertence à passagem do percurso.");
    }
}
```

No model ***DistributionNetwork*** que vai ser o responsável por gerar o percurso de entrega, criou-se alguns métodos auxiliares para este efeito.

Criou-se a função ***getProviderList*** que percorre uma dada lista de expedição e verifica quais os diferentes produtores contidos na mesma.

Esta função é importante pelo facto de indicar quais os produtores que devem fazer parte dos pontos de passagem do percurso.

Este método tem complexidade de tempo **O(nm)**, onde n é o número de cabazes expedidos na lista e m é o número de produtores agrícolas em cada cabaz.

```
private List<AgriculturalProducer> getProviderList(List<BasketExpedition> basketExpeditionsList) {
    List<AgriculturalProducer> providerList = new LinkedList<>();

    if (basketExpeditionsList != null) {
        for (BasketExpedition basketExpedition : basketExpeditionsList) {

            for (AgriculturalProducer agriculturalProducer : basketExpedition.getProviderList()) {
                if (!providerList.contains(agriculturalProducer)) {
                    providerList.add(agriculturalProducer);
                }
            }
        }
    }

    return providerList;
}
```

Criou-se a função ***getBasketsDelivered*** que além de ser responsável por retornar a lista de *hubs* que devem fazer parte dos pontos de passagem do percurso indica também quais são os cabazes a entregar em cada *hub*. Isto é possível através da utilização de um mapa, onde a chave é o *hub* e os valores uma lista do tipo *Set* de cabazes.

Este método tem uma complexidade de tempo **O(n)**, onde n é o número de cabazes contidos na lista de expedição.

```
private Map<CompanyCostumer, Set<BasketExpedition>> getBasketsDelivered(List<BasketExpedition> basketExpeditionsList) {
    Map<CompanyCostumer, Set<BasketExpedition>> basketsDelivered = new HashMap<>();

    if (basketExpeditionsList != null) {
        for (BasketExpedition basketExpedition : basketExpeditionsList) {
            CompanyCostumer hub = basketExpedition.getCostumer().getNearestHub();

            Set<BasketExpedition> basketExpeditionSet = basketsDelivered.get(hub);
            if (basketExpeditionSet == null) {
                basketExpeditionSet = new HashSet<>();
                basketsDelivered.put(hub, basketExpeditionSet);
            }

            basketExpeditionSet.add(basketExpedition);
        }
    }

    return basketsDelivered;
}
```

A função ***generateRoute*** é responsável por gerar o melhor percurso entre um conjunto de pontos. Esta recebe como parâmetros uma lista de pontos que devem ser incluídos no percurso, além disso é recebido uma *LinkedList* para guardar o melhor percurso e um booleano que indica se na verificação de todas as possibilidades do percurso o ponto inicial deve ser fixo ou não.

Esta função utiliza um grafo que contém a distância mínima dos pontos da rede de distribuição, este grafo já foi devidamente preenchido com a utilização do algoritmo *Floyd-Warshall* que possui uma complexidade **O(n3)** e é determinístico. Este método tem uma complexidade de tempo de **O(n^2)**, onde n é o número de pontos de passagem, sendo determinístico.

```
private <V extends NetworkPoint> int generateRoute(List<V> waypoints, LinkedList<NetworkPoint> shortPath, boolean firstFixed
    int betterDistance = Integer.MAX_VALUE;
    LinkedList<NetworkPoint> betterShortPath = new LinkedList<>();

    int waypointsSize = waypoints.size();
    for (int i = 0; i < waypointsSize; i++) {
        int currentDistance = 0;

        for (int j = 0; j < waypointsSize - 1; j++) {
            currentDistance += minDistDistributionNetwork.edge(waypoints.get(j), waypoints.get(j + 1)).getWeight();
        }

        if (betterDistance > currentDistance) {
            betterShortPath.clear();
            betterShortPath.addAll(waypoints);
            betterDistance = currentDistance;
        }

        Collections.rotate(waypoints, distance: 1);

        if (firstFixedPosition) {
            V firstPosition = waypoints.get(1);
            waypoints.remove(firstPosition);
            waypoints.add( index: 0, firstPosition);
        }
    }

    shortPath.addAll(betterShortPath);

    return betterDistance;
}
```

Por fim, existe a função ***generateDeliveryRoute*** que é o principal responsável por gerar o percurso de entrega que minimiza a distância total percorrida para uma lista de expedição diária utilizando as funções auxiliares anteriores explicadas.

Inicialmente esta função chama a função *getProviderList* para receber a lista de produtores que devem ser incluídos no percurso e a função *getBasketsDelivered* para receber os hubs e respetivos cabazes a serem entregues.

Passando a parte da geração do percurso, inicialmente é procurado o melhor percurso entre os produtores e seguidamente entre o último produtor e todos os hubs.

Um exemplo prático de como funciona esta função:

Os produtores a considerar no percurso são: CT10, CT17, CT6

Os *hubs* a considerar no percurso são: CT10, CT11, CT14, CT4, CT5, CT9

Inicialmente verifica-se o melhor percurso entre os produtores que é CT17, CT6, CT10 com um custo de 141412.

Seguidamente junta-se à lista de *hubs* o último produtor do percurso (CT10) e verifica-se o melhor percurso que é CT10, CT4, CT5, CT9, CT11, CT14 com um custo de 1038242.

Ficando assim o percurso de entrega que minimiza a distância total percorrida: CT17, CT6, CT10, CT4, CT5, CT9, CT11, CT14 com um custo final de 1179654

Esta classe retorna no final uma instância do objeto DeliveryRoute contendo os pontos de passagem do percurso (produtores e hubs), os cabazes entregues em cada hub, a distância entre todos os pontos do percurso e a distância total.

```
public DeliveryRoute generateDeliveryRoute(List<BasketExpedition> basketExpeditionsList) {
    // Cabazes entregues em cada hub
    Map<CompanyCostumer, Set<BasketExpedition>> basketsDelivered = getBasketsDelivered(basketExpeditionsList);

    // Lista de produtores
    List<AgriculturalProducer> providerList = getProviderList(basketExpeditionsList);
    Collections.sort(providerList);
    // Lista de hubs
    List<NetworkPoint> hubList = new LinkedList<>(basketsDelivered.keySet());
    Collections.sort(hubList);

    // Cria uma lista para receber o caminho mais curto
    LinkedList<NetworkPoint> waypoints = new LinkedList<>();

    int totalDistance = 0;
    //Verifica o caminho mais curto entre todos os produtores
    totalDistance += generateRoute(providerList, waypoints, firstFixedPosition: false);

    // Para haver uma conexão entre os produtores e os hub, é colocado o último produtor
    int positionLastProvider = providerList.size() - 1;
    hubList.add( index: 0, waypoints.get(positionLastProvider));
    waypoints.remove(positionLastProvider);

    // Verifica o caminho mais curto entre o último produtor e todos os hub
    totalDistance += generateRoute(hubList, waypoints, firstFixedPosition: true);

    // Retorna uma rota de entrega
    return new DeliveryRoute(waypoints, basketsDelivered, totalDistance, minDistDistributionNetwork);
}
```

US311

Nesta *User Story* é necessário calcular estatísticas para uma determinada lista de expedição.

Por cabaz:

Em relação às estatísticas por cabaz é necessário calcular o nº de produtos totalmente satisfeitos, o nº de produtos parcialmente satisfeitos, o nº de produtos não satisfeitos, a percentagem total do cabaz satisfeito e nº de produtores que forneceram o cabaz.

Para atender às necessidades optou-se pela criação de uma classe denominada ***BasketStatistics*** que contêm como atributos os vários pontos a calcular.

```
public class BasketStatistics {
    /**
     * O nº de produtos totalmente satisfeitos.
     */
    6 usages
    private int fullySatisfiedProducts;

    /**
     * O nº de produtos parcialmente satisfeitos.
     */
    6 usages
    private int partiallySatisfiedProducts;

    /**
     * O nº de produtos não satisfeitos.
     */
    6 usages
    private int unsatisfiedProducts;

    /**
     * Percentagem total do cabaz satisfeito.
     */
    3 usages
    private double totalPercentageBasketSatisfied;

    /**
     * O nº de produtores que forneceram o cabaz.
     */
    3 usages
    private int producersProvidedBasket;
```

Esta classe contém ainda algumas funções como o *calculatePercentageBasketSatisfied* que é responsável por calcular a percentagem total do cabaz satisfeito, o *fullySatisfiedBasket* que indica se o cabaz é totalmente satisfeito e o *partiallySatisfiedBasket* que indica se o cabaz é parcialmente satisfeito.

A classe possui também a habitual função *equals*.

```
/**
 * Função para calcular a percentagem total do cabaz satisfeito.
 *
 * @return Percentagem total do cabaz satisfeito.
 */
1 usage
private double calculatePercentageBasketSatisfied() {
    int totalProducts = fullySatisfiedProducts + partiallySatisfiedProducts + unsatisfiedProducts;

    return (double) fullySatisfiedProducts / totalProducts * 100;
}

/**
 * Função para verificar se um cabaz é totalmente satisfeito.
 *
 * @return Retorna true se o cabaz é totalmente satisfeito, caso contrário devolve false.
 */
3 usages
public boolean fullySatisfiedBasket() {
    return fullySatisfiedProducts > 0 && partiallySatisfiedProducts == 0 && unsatisfiedProducts == 0;
}

/**
 * Função para verificar se um cabaz é parcialmente satisfeito.
 *
 * @return Retorna true se o cabaz é parcialmente satisfeito, caso contrário devolve false.
 */
3 usages
public boolean partiallySatisfiedBasket() {
    return partiallySatisfiedProducts > 0 || unsatisfiedProducts > 0;
}
```

Para realizar o cálculo das estatísticas por cabaz colocou-se um atributo na classe ***BasketExpedition***, com uma relação de composição, referente a essas mesmas estatísticas.

```
/**
 * Estatísticas do cabaz.
 */
2 usages
private BasketStatistics statisticsBasket;
```

Nesta mesma classe criou-se um método que é responsável por calcular as estatísticas do cabaz.

Para o cálculo do nº de produtos totalmente satisfeitos, nº de produtos parcialmente satisfeitos e o nº de produtos não satisfeitos, optou-se por realizar ao percorrer a lista de produtos fornecidos para o cabaz.

Ao percorrer esta lista em cada produto verifica-se em que estado encontra-se e incrementa-se a variável desse mesmo estado.

Já para o nº de produtores que forneceram o cabaz, optou-se por adicionar em cada produto o seu produtor a uma lista do tipo *Set*. No final para saber quantos produtores forneceram o capaz é apenas necessário verificar o tamanho desta mesma lista.

No fim deste método é criada uma instância das estatísticas por cabaz com os atributos calculados.

Este método tem uma complexidade de tempo de **O(n)**, onde n é o número de produtos na lista de produtos fornecidos, sendo determinístico

```java
/**
 * Método para calcular as estatísticas do cabaz.
 */
1 usage
public void calculateBasketStatistics() {
    int fullySatisfiedProducts = 0;
    int partiallySatisfiedProducts = 0;
    int unsatisfiedProducts = 0;
    Set<AgriculturalProducer> providerList = new HashSet<>();

    for (ProductProvided productProvided : productsProvided) {

        if (productProvided.unsatisfiedProduct()) {
            unsatisfiedProducts++;
        } else {
            if (productProvided.fullySatisfiedProduct()) {
                fullySatisfiedProducts++;
            } else if (productProvided.partiallySatisfiedProduct()) {
                partiallySatisfiedProducts++;
            }

            providerList.add(productProvided.getProvider());
        }
    }

    this.statisticsBasket = new BasketStatistics(fullySatisfiedProducts, partiallySatisfiedProducts, unsatisfiedProducts,
            providerList.size());
}
```

Por fim, para calcular para uma lista de expedição as estatísticas por cabaz, criou-se um método na classe ***DistributionNetwork*** que verifica se a lista de expedição recebida não é nula e caso não seja é percorrida e em cada cabaz contido nela é calculado as suas estatísticas através do método *calculateBasketStatistics* explicado anteriormente.

```
/**
 * Método para calcular para uma lista de expedição as estatísticas por cabaz.
 *
 * @param basketExpeditionsList A lista de expedição de cabazes a ser populada.
 */
5 usages
public void calculateStatisticsByBasket(List<BasketExpedition> basketExpeditionsList) {
    if (basketExpeditionsList != null) {
        for (BasketExpedition basketExpedition : basketExpeditionsList) {
            basketExpedition.calculateBasketStatistics();
        }
    }
}
```

Por cliente:

Em relação às estatísticas por cliente é necessário calcular o nº de cabazes totalmente satisfeitos, nº de cabazes parcialmente satisfeitos, nº de fornecedores distintos que forneceram todos os seus cabazes.

Para atender às necessidades optou-se pela criação de uma classe denominada ***CostumerStatistics*** que contêm como atributos os vários pontos a calcular.

```
/**
 * O nº de cabazes totalmente satisfeitos.
 */
4 usages
private int fullySatisfiedBaskets;

/**
 * O nº de cabazes parcialmente satisfeitos.
 */
4 usages
private int partiallySatisfiedBaskets;

/**
 * O nº de fornecedores distintos que forneceram todos os seus cabazes.
 */
4 usages
private int differentSupliersGiveBaskets;
```

Esta classe contém ainda algumas funções de incrementação necessárias para os cálculos pedidos. Tal como as habituais funções de *set* e *equals*

```
/**
 * Metodo incrementador da variavel fullySatisfiedBaskets.
 */
1 usage   francisco_lascasas_bogalho
public void addFullySatisfiedBaskets() { this.fullySatisfiedBaskets++; }

/**
 * Metodo incrementador da variavel partiallySatisfiedBaskets.
 */
1 usage   francisco_lascasas_bogalho
public void addPartiallySatisfiedBaskets() { this.partiallySatisfiedBaskets++; }

/**
 * Metodo alterar o valor da variavel differentSupliersGiveBaskets.
 *
 * @param differentSupliersGiveBaskets
 */
1 usage   francisco_lascasas_bogalho
public void setDifferentSupliersGiveBaskets(int differentSupliersGiveBaskets) {
    this.differentSupliersGiveBaskets = differentSupliersGiveBaskets;
}
```

```java
/**
 * Método para verificar se este objeto atual é igual a outro objeto do mesmo tipo.
 *
 * @param obj O objeto a ser comparado com este atual.
 * @return true se o outro objeto representa as estatísticas do cliente igual a este, caso contrário devolve false.
 */
francisco_lascasas_bogalho
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }

    if (obj == null || (!(obj instanceof CostumerStatistics that))) {
        return false;
    }

    return (fullySatisfiedBaskets == that.fullySatisfiedBaskets &&
            partiallySatisfiedBaskets == that.partiallySatisfiedBaskets &&
            differentSupliersGiveBaskets == that.differentSupliersGiveBaskets);
}
```

Criou-se um método na classe ***DistributionNetwork*** o método principal responsável por todas as estatísticas pedidas, a este método com associados 3 métodos.

```java
public void calculateStatisticsByCostumer(List<BasketExpedition> basketExpeditionsList) {
    if (basketExpeditionsList != null) {
        calculateStatisticsByBasket(basketExpeditionsList);

        Map<Costumer, Set<AgriculturalProducer>> producerByCostumer = new HashMap<>();

        //Percorre a lista passada por parametro
        for (BasketExpedition basketExpedition : basketExpeditionsList) {
            BasketStatistics basketStatistics = basketExpedition.getStatisticsBasket();
            Costumer costumer = basketExpedition.getCostumer();

            //Verifica se o cabaz está totalmete satisfeitos ou parcialmente satisfeitos
            if (basketStatistics.fullySatisfiedBasket()) {
                costumer.getCostumerStatistics().addFullySatisfiedBaskets();
            } else if (basketStatistics.partiallySatisfiedBasket()) {
                costumer.getCostumerStatistics().addPartiallySatisfiedBaskets();
            }

            Set<AgriculturalProducer> producerSet = producerByCostumer.get(costumer);
            //Verifica se o producerSet tem conteudo se não, cria-o
            if (producerSet == null) {
                producerSet = new HashSet<>();
                producerByCostumer.put(costumer, producerSet);

            }

            producerSet.addAll(basketExpedition.getProviderList());
        }

        for (Costumer costumer : producerByCostumer.keySet()) {
            costumer.getCostumerStatistics().setDifferentSupliersGiveBaskets(producerByCostumer.get(costumer).size());
        }
    }
}
```

Por produtor:

Em relação às estatísticas por produtor é necessário calcular o nº de cabazes fornecidos totalmente, nº de cabazes fornecidos parcialmente, nº de clientes distintos fornecidos, nº de produtos totalmente esgotados, nº de hubs fornecidos.

Para atender às necessidades optou-se pela criação de uma classe denominada ***AgriculturalProducerStatistics***que contêm como atributos os vários pontos a calcular.

```
/**
 * O nº de cabazes fornecidos totalmente.
 */
4 usages
private int basketFullySupplied;

/**
 * O nº de cabazes fornecidos parcialmente.
 */
4 usages
private int basketPartiallySupplied;

/**
 * O nº de clientes distintos fornecidos.
 */
4 usages
private int distinctClientsProvided;

/**
 * O nº de produtos totalmente esgotados.
 */
4 usages
private int completelySoldProducts;

/**
 * O nº de hubs fornecidos.
 */
4 usages
private int suppliedHubs;
```

Esta classe contem ainda algumas funções de incrementação necessárias para os cálculos pedidos. Tal como as habituais funções de *set* e a *equals*

```
/**
 * Metódo incrementador da variavel basketFullySupplied.
 */
1 usage
public void addBasketFullySupplied() { this.basketFullySupplied++; }

/**
 * Metódo incrementador da variavel basketPartiallySupplied.
 */
1 usage
public void addBasketPartiallySupplied() { this.basketPartiallySupplied++; }

/**
 * Metódo alterar o valor da variavel distinctClientsProvided.
 */
1 usage
public void setDistinctClientsProvided(int distinctClientsProvided) {
    this.distinctClientsProvided = distinctClientsProvided;
}

/**
 * Metódo alterar o valor da variavel completelySoldProducts.
 */
1 usage
public void setCompletelySoldProducts(int completelySoldProducts) {
    this.completelySoldProducts = completelySoldProducts;
}

/**
 * Metódo alterar o valor da variavel suppliedHubs.
 */
1 usage
public void setSuppliedHubs(int suppliedHubs) { this.suppliedHubs = suppliedHubs; }
```

```
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }

    if (obj == null) {
        return false;
    }

    AgriculturalProducerStatistics agriculturalProducerStatistics = (AgriculturalProducerStatistics) obj;

    return (basketFullySupplied == agriculturalProducerStatistics.basketFullySupplied &&
            basketPartiallySupplied == agriculturalProducerStatistics.basketPartiallySupplied &&
            distinctClientsProvided == agriculturalProducerStatistics.distinctClientsProvided &&
            completelySoldProducts == agriculturalProducerStatistics.completelySoldProducts &&
            suppliedHubs == agriculturalProducerStatistics.suppliedHubs);
}
```

Criou-se um método na classe ***DistributionNetwork*** o método principal responsável por todas as estatísticas pedidas, a este método com associados 3 métodos.

```
public void calculateStatisticsByAgriculturalProducer(List<BasketExpedition> basketExpeditionsList) {

    Map<AgriculturalProducer, Set<Costumer>> getCostumerProducer = new HashMap<>();
    int basketExpeditionDay = 0;

    if (basketExpeditionsList != null) {
        for (BasketExpedition basketExpedition : basketExpeditionsList) {
            for (ProductProvided productProvided : basketExpedition.getProductsProvided()) {

                AgriculturalProducer agriculturalProducer = productProvided.getProvider();

                if (agriculturalProducer != null) {
                    // basketFullySupplied & basketPartiallySupplied
                    if (productProvided.fullySatisfiedProduct()) {
                        agriculturalProducer.getAgriculturalProducerStatistics().addBasketFullySupplied();
                    } else {
                        agriculturalProducer.getAgriculturalProducerStatistics().addBasketPartiallySupplied();
                    }

                    Set<Costumer> costumerSet = getCostumerProducer.get(agriculturalProducer);
                    if (costumerSet == null) {
                        costumerSet = new HashSet<>();
                        getCostumerProducer.put(agriculturalProducer, costumerSet);
                    }
                    costumerSet.add(basketExpedition.getCostumer());
                }

                basketExpeditionDay = basketExpedition.getDay();

            }
        }
        //distinctClientsProvided
        calculateDistinctClientsProvided(getCostumerProducer);

        //completelySoldProducts
        calculateCompletelySoldProducts(getCostumerProducer.keySet(), basketExpeditionDay);

        //suppliedHubs
        calculateProvidedHud(getCostumerProducer);

    }
```

O método ***calculateDistinctClientsProvided*** responsável por calcular para uma lista de expedição as características do nº de clientes distintos fornecidos. Este método percorre um *map* de produtores agrícolas e verifica para cada produtor o número de clientes. Este método tem uma complexidade de tempo de O(n).

```
private void calculateDistinctClientsProvided(Map<AgriculturalProducer, Set<Costumer>> getCostumerProducer) {

    for (AgriculturalProducer agriculturalProducer : getCostumerProducer.keySet()) {
        int numberDistinctClientsProvided = getCostumerProducer.get(agriculturalProducer).size();
        agriculturalProducer.getAgriculturalProducerStatistics().setDistinctClientsProvided(numberDistinctClientsProvided);
    }
}
```

O método ***calculateCompletelySoldProducts*** responsável por calcular para uma lista de expedicao as caracteristicas do nº de produtos totalmente esgotado. Este método vai verificar em todos os cabazes dos produtores todo quais os que estão a 0. Este método tem uma complexidade de tempo de **O(a.b)**, onde **a** é o número de produtores agrícolas e **b** é o número de produtos no basket.

```
private void calculateCompletelySoldProducts(Set<AgriculturalProducer> agriculturalProducers, int day) {

    Map<AgriculturalProducer, Basket> agriculturalProducerBasketMap = productsAvailable.get(day);

    if (agriculturalProducerBasketMap != null) {
        int counter = 0;
        Iterator<AgriculturalProducer> agriculturalProducerIterator = agriculturalProducers.iterator();

        while (agriculturalProducerIterator.hasNext()) {

            AgriculturalProducer agriculturalProducer = agriculturalProducerIterator.next();
            Basket basket = agriculturalProducerBasketMap.get(agriculturalProducer);

            Iterator<Product> productIterator = basket.getProductsSet().iterator();
            while (productIterator.hasNext()) {
                if (basket.getProductQuantity(productIterator.next()) == 0.0) {
                    counter++;
                }
            }
            agriculturalProducer.getAgriculturalProducerStatistics().setCompletelySoldProducts(counter);
        }
    }
}
```

O método ***calculateProvidedHud*** responsável por calcular para uma lista de expedicao o nº de hubs fornecidos. Este método vai verificar iterar os hubs todos contabilizando-os. Este método tem uma complexidade de tempo de O(a), onde a é o número de produtores agrícolas.

```
private void calculateProvidedHud(Map<AgriculturalProducer, Set<Costumer>> getCostumerProducer) {

    for (AgriculturalProducer agriculturalProducer : getCostumerProducer.keySet()) {
        Set<Costumer> costumerSet = getCostumerProducer.get(agriculturalProducer);
        Set<CompanyCostumer> hubSet = new HashSet<>();

        if (costumerSet != null) {

            Iterator<Costumer> costumerIterator = costumerSet.iterator();
            while (costumerIterator.hasNext()) {
                hubSet.add(costumerIterator.next().getNearestHub());
            }
            agriculturalProducer.getAgriculturalProducerStatistics().setSuppliedHubs(hubSet.size());

        }
    }
}
```

Por hub:

Em relação às estatísticas por hub, é preciso calcular número de clientes distintos que recolhem cabazes em cada hub e o número de produtores distintos que fornecem cabazes por hub.

Posto isto, optamos por criar uma classe denominada ***HubtStatistics*** que contêm os pontos a calcular.

```java
/**
 * O nº de clientes distintos que recolhem cabazes em cada hub.
 */
private int differentCostumersFetchHub;

/**
 * O nº de produtores distintos que fornecem cabazes para o hub.
 */
private int differentProducersGiveHub;

/**
 * Construtor para inicializar uma nova instância desta classe com os atributos differentCostumersFetchHub
 * e differentProducersGiveHub.
 *
 * @param differentCostumersFetchHub O nº de clientes distintos que recolhem cabazes em cada hub.
 * @param differentProducersGiveHub  O nº de produtores distintos que fornecem cabazes para o hub.
 */
public HubStatistics(int differentCostumersFetchHub, int differentProducersGiveHub) {
    this.differentCostumersFetchHub = differentCostumersFetchHub;
    this.differentProducersGiveHub = differentProducersGiveHub;
}
```

Nesta classe podemos verificar que temos, uma função *equals*, sendo que esta no permite verificar se o objeto atual é igual a outro do mesmo tipo. Em que no caso do objeto ser igual irá retornar *true*, no caso de ser diferente irá retornar false.

```java
/**
 * Método para verificar se este objeto atual é igual a outro objeto do mesmo tipo.
 *
 * @param o O objeto a ser comparado com este atual.
 * @return true se o outro objeto representa as estatísticas do hub igual a este, caso contrário devolve false.
 */
@Override
public boolean equals(Object o) {
    if (this == o) {
        return true;
    }

    if (o == null) {
        return false;
    }

    HubStatistics that = (HubStatistics) o;
    return differentCostumersFetchHub == that.differentCostumersFetchHub && differentProducersGiveHub == that.differentProducersGiveHub;
}
```

Por último, para calcular para uma lista de expedição as estatísticas por hub, criou-se um método na classe ***DistributionNetwork*** que verifica se a lista de expedição não é nula e caso esta não seja, é percorrida e em cada hub será calculado as suas estatísticas através do método *calculateHubStatistics*.

Este método tem uma complexidade de tempo O(n), onde n é o número de expedições dos cabazes no ***basketExpeditionsList***. Isto acontece porque o método executa uma única passagem linear pela lista, realizando um número constante de operações em cada elemento.

```java
/**
 * Método para calcular para uma lista de expedição as estatísticas por hub
 *
 * @param basketExpeditionsList A lista de expedição de cabazes a ser populada.
 */
public void calculateStatisticsByHub(List<BasketExpedition> basketExpeditionsList) {
    if (basketExpeditionsList != null) {
        Map<CompanyCostumer, Set<Costumer>> costumersByHub = new HashMap<>();
        Map<CompanyCostumer, Set<AgriculturalProducer>> producerByHub = new HashMap<>();
        //Percorre a lista passada por parâmetro
        for (BasketExpedition basketExpedition : basketExpeditionsList) {
            CompanyCostumer hub = basketExpedition.getDeliveryLocation();

            Set<Costumer> costumerSet = costumersByHub.get(hub);
            //Verifica se o costumerSet tem conteúdo, se não, cria-o
            if (costumerSet == null) {
                costumerSet = new HashSet<>();
                costumersByHub.put(hub, costumerSet);
            }
            costumerSet.add(basketExpedition.getCostumer());

            Set<AgriculturalProducer> producerInBasket = basketExpedition.getProviderList();
            Set<AgriculturalProducer> agriculturalProducerSet = producerByHub.get(hub);
            //Verifica se agriculturalProducerSet existe, senão vai

            if (agriculturalProducerSet == null) {
                producerByHub.put(hub, producerInBasket);
            } else {
                agriculturalProducerSet.addAll(producerInBasket);
            }
        }

        for (CompanyCostumer hub : costumersByHub.keySet()) {
            int differentCostumersFetchHub = costumersByHub.get(hub).size();
            int differentProducersGiveHub = producerByHub.get(hub).size();
            hub.setHubStatistics(new HubStatistics(differentCostumersFetchHub, differentProducersGiveHub));
        }
```

Outros

Foram criadas algumas classes para lançar exceções personalizadas, de forma que seja mais fácil de avisar o utilizador e até o próprio programador do erro que foi lançado ao utilizar certo método/funcionalidade, em *RunTime*.

Estas classes são:

- ***InvalidFileLineException***

```
public class InvalidFileLineException extends RuntimeException {

    /**
     * Construtor para lançar uma mensagem por defeito.
     */
    no usages   Tiago Leite
    public InvalidFileLineException() { super("Foi encontrado um erro numa linha do ficheiro!"); }

    /**
     * Construtor para lançar a exceção com uma mesagem a indicar a linha do ficheiro onde erro foi encontrado.
     *
     * @param line A linha do ficheiro que contém erro.
     */
    3 usages   Tiago Leite
    public InvalidFileLineException(int line) {
        super("Verifique as colunas existentes na linha " + line + " do ficheiro! Por favor, corrija o erro.");
    }
```

Esta exceção é lançada quando alguma linha do ficheiro não se encontra válida. Aqui é possível utilizar mensagens personalizadas, indicar a linha do ficheiro onde o erro ocorreu, e até mesmo indicar também em que atributo o erro foi encontrado.

- ***InvalidProductQuantityException***

```
public class InvalidProductQuantityException extends RuntimeException {

    /**
     * Construtor para lançar uma mensagem por defeito.
     */
    no usages   Tiago Leite
    public InvalidProductQuantityException() {
        super("Quantidade de produto inválida.");
    }

    /**
     * Construtor para lançar a exceção com uma mensagem personalizada.
     *
     * @param message A mensagem personalizada.
     */
    1 usage   Tiago Leite
    public InvalidProductQuantityException(String message) {
        super(message);
    }
}
```

Esta exceção é lançada quando existe uma quantidade inválida de um certo produto no ficheiro que contém a informação acerca dos cabazes. Aqui existe a possibilidade de ser lançada uma mensagem por defeito, ou então uma mensagem personalizada.

- ***NetworkPointNotFoundException***

```java
public class NetworkPointNotFoundException extends RuntimeException {

    /**
     * Construtor para lançar uma mensagem por defeito.
     */
    no usages  Tiago Leite
    public NetworkPointNotFoundException() {
        super("O ponto de rede não é válido!");
    }

    /**
     * Construtor para lançar a exceção com uma mesagem a indicar a linha do ficheiro onde erro foi encontrado.
     *
     * @param line A linha do ficheiro que contém erro.
     */
    no usages  Tiago Leite
    public NetworkPointNotFoundException(int line) {
        super("O ponto de rede na linha " + line + " do ficheiro, não é valido!");
    }

    /**
     * Construtor para lançar a exceção com uma mensagem personalizada.
     *
     * @param message A mensagem personalizada.
     */
    1 usage  Tiago Leite
    public NetworkPointNotFoundException(String message) {
        super(message);
    }
}
```

Já esta exceção é lançada quando algum ponto de rede não é encontrado no sistema. Aqui é possível lançar a exceção com uma mensagem por defeito, uma mensagem em que também informa a linha do ficheiro onde foi encontrado um ponto de rede que não está registado no sistema, e é ainda possível lançar uma exceção com uma mensagem personalizada.

- ***ProductNotFoundException***

```java
public class ProductNotFoundException extends RuntimeException {
    /**
     * Construtor para lançar uma mensagem por defeito.
     */
    1 usage  Tiago Leite
    public ProductNotFoundException() {
        super("O produto do qual pretende adicionar quantidade não existe!");
    }

    /**
     * Construtor para lançar a exceção com uma mensagem personalizada.
     *
     * @param message A mensagem personalizada.
     */
    no usages  Tiago Leite
    public ProductNotFoundException(String message) {
        super(message);
    }
}
```

No caso desta exceção, ela é lançada quando um produto não se encontra registado no sistema, e é pedido por um cliente ou fornecido por um produtor. Aqui é possível lançar a exceção com uma mensagem por defeito ou então com uma mensagem personalizada.

Divisão de trabalho

Antes de elaborarmos o trabalho e, após uma análise em grupo, optamos por dividir as *User Stories* pelos diferentes membros da equipa da seguinte forma:

- US307 – Tiago Leite e João Durães;
- US308 – Tiago Leite;
- US309 – Francisco Bogalho;
- US310 – Gabriel Gonçalves;
- US311:
    - Por cabaz: Gabriel Gonçalves;
    - Por cliente: Francisco Bogalho;
    - Por produtor: João Durães;
    - Por *hub*: António Bernardo;

Melhoramentos possíveis

Em relação a melhoramentos, podemos indicar os seguintes pontos a serem melhorados neste projeto:

- Complexidade Temporal;

Conclusão

Em suma, neste projeto desenvolvemos as nossas capacidades ao usar as utilidades dos grafos. Podemos concluir ainda, que quanto mais estudado for o caso e melhor planeado for o trabalho mais facilmente se absorvem as alterações que acontecem ao longo do processo.

Posto isto, consideramos que, apesar de ser possível efetuar algumas melhorias neste trabalho, o projeto encontra-se de acordo com os requisitos pedidos.