




# Relatório Sprint 2

Diogo Araújo – 1200967  
João Batista – 1211396  
David Dias – 1211415  
Ezequiel Estima – 1211417  
Marco Andrade - 1211469



## Conteúdo

Índice de Imagens .....	1
Class Diagram .....	1
US 307 .....	2
Análise da Complexidade .....	3
US 308 .....	3
Análise da Complexidade .....	6
US309 .....	7
Análise da Complexidade .....	10
US310 .....	11
Análise de complexidade .....	11
US311 .....	12

## Índice de Imagens

Figura 2 – Método loadBaskets 1ª parte.....	2
Figura 1 - Método loadBaskets 2ª parte .....	2
Figura 3 - Método generateShipmentList público .....	3
Figura 4 - Método generateShipmentList privado 1ªparte.....	4
Figura 5 - Método generateShipmentList privado 2ªparte.....	4
Figura 6 - Método addLeftOver 1ªparte .....	5
Figura 7 - Método addLeftOver 2ªparte .....	5
Figura 8 - Método findNearestHub .....	6
Figura 9 -Método generateShipmentList público .....	7
Figura 10 -Método generateShipmentList privado 1ªparte.....	8
Figura 11 - Método generateShipmentList privado 2ªparte.....	9
Figura 12 - Método findNearestProducers .....	10
Figura 13 - Método generateDeliveryRoutesWithShipmentList.....	11
Figura 14 – Método basketAnalyticsPerShipmentList .....	12
Figura 15 - Método clientAnalyticsPerShipmentList.....	13
Figura 16 - Método addProducer .....	13
Figura 17 - Método producerAnalyticsPerShipmentList .....	14
Figura 18 - Método numberOfDepletedProducts .....	15
Figura 19 - Método hubAnalyticsPerShipmentList.....	16
Figura 20 - Método generateAnalysis .....	16

## Class Diagram

Em anexo com a submissão deste ficheiro encontra-se o Diagrama de Classes com o nome CD.svg

## US 307

Para esta US foi pedido para importar uma lista de cabazes. Para tal foi criado um método chamado *loadBaskets*. Este método fica encarregue de carregar os cabazes em cada vértice no seu map chamado *orderByDay*. O map *orderByDay* dos produtores é organizado da seguinte forma *Map<Dia, Map<Id do Produto, ArrayList<Produto>>*. Isto porque, para cada dia um produtor pode ter vários produtos com o mesmo id mas, com validades diferentes, já para os compradores (empresas, clientes) o seu *orderByDay* o *Map* é organizado assim *Map<Dia, ArrayList<Produto>>*, porque para cada dia um cliente é indiferente a data de validade.

```
public int loadBaskets(List<String> basketsInfo) {
    producersWithOrders = new HashMap<>();
    buyersWithOrders = new HashMap<>();
    int successfullines = 0;
    ArrayList<Product> listProduct;
    for (String s : basketsInfo) {
        String[] values = s.split(" ");
        if (s.charAt(2) != 'L' & s.charAt(1) != 'L') {
            if (values[0].charAt(0) == 'P') {
                values[0] = values[0].replace(target: "\\", replacement: "");
            }
            int day = Integer.parseInt(values[1]);
            try {
                int vKey = keyMemberId.get(values[0]); // obter a key do vértice através do Loc id
                listProduct = new ArrayList<>();
                for (int i = 2; i < values.length; i++) {
                    if (Double.parseDouble(values[i]) != 0) {
                        //inserção de cabazes se forem produtores
                        if (values[0].charAt(0) == 'P') {
                            // inserção no producersWithOrders-----
                            //check se já existe o dia no map
                            if (!producersWithOrders.containsKey(day)) {
                                producersWithOrders.put(day, new HashMap<>());
                            }
                            //check se já existe o tipo de produto nesse dia
                            if (!producersWithOrders.get(day).containsKey(i - 1)) {
                                producersWithOrders.get(day).put(i - 1, new ArrayList<>());
                            }
                            producersWithOrders.get(day).get(i - 1).add((Producer) network.vertex(vKey));
                            Producer producer = (Producer) network.vertex(vKey);
                            // inserção no getOrderByDay do produtor-----
                            //check se já existe o dia no map
                            if (!producer.getOrderByDay().containsKey(day)) {
                                producer.getOrderByDay().put(day, new HashMap<>());
                            }
                            //adicionar o produto ao getOrderByDay
                            producer.getOrderByDay().get(day).put(i - 1, new ArrayList<>());
                            producer.getOrderByDay().get(day).get(i - 1).add(new Product(i - 1, Double.parseDouble(values[i])));
                        }
                    }
                }
            } catch (Exception ignored) {}
        }
    }
    return successfullines;
}
```

Figura 2 – Método loadBaskets 1ª parte

```

    //inserção de cabazes se forem compradores
    if (values[0].charAt(0) != 'P' && !listProduct.isEmpty()) {
        // inserção em buyersWithOrders-----
        if (!buyersWithOrders.containsKey(day)) {
            buyersWithOrders.put(day, new ArrayList<>());
        }
        buyersWithOrders.get(day).add((Buyer) network.vertex(vKey));
        // inserção no getOrderByDay do comprador-----
        Buyer buyer = (Buyer) network.vertex(vKey);
        buyer.getOrderByDay().put(day, listProduct);
    }
    successfullines++;
} catch (NullPointerException ignored) {
}
}
return successfullines;
}
```

Figura 1 - Método loadBaskets 2ª parte

## Análise da Complexidade

Este algoritmo que foi criado para a importação de cabazes é determinístico e tem uma complexidade temporal  $O(C \times P)$ , sendo C o número de cabazes para serem inseridos e P o número de diferentes tipos de produtos. Esta análise de complexidade é justificada, porque para cada cabaz são inseridos os diferentes tipos de produtos.

## US 308

Nesta US é pedido para gerar uma lista de expedição sem restrições quanto ao Produtor que forneça o(s) produto(s) para o pedido de um cliente.

Para o efeito, foram desenvolvidos dois métodos *generateShipmentList* um será, portanto, a interface pública que faz a chamada do auxiliar com as cópias das estruturas que possuem os compradores e os produtores da rede, para não realizar *Object Mutation* que iriam afetar a chamada posterior da funcionalidade da US 308 ou da US 309 alterando diretamente nos produtores os seus produtos originando dados conflituosos que não estariam a representar a realidade.

```
public List<Shipment> generateShipmentList(int day) {  
  
    if(!producersWithOrders.containsKey(day)) return null;  
  
    Map<Integer, Map<Integer, ArrayList<Producer>>> producersWithOrders = new HashMap<>(this.producersWithOrders);  
    Map<Integer, ArrayList<Buyer>> buyersWithOrders = new HashMap<>(this.buyersWithOrders);  
  
    for (int i = 1; i < day; i++) {  
        generateShipmentList(i, buyersWithOrders, producersWithOrders);  
        if(addLeftOver(i, producersWithOrders) != null) producersWithOrders = addLeftOver(i, producersWithOrders);  
    }  
    return generateShipmentList(day, buyersWithOrders, producersWithOrders);  
}
```

Figura 3 - Método *generateShipmentList* público

```

private List<Shipment> generateShipmentList(int day, Map<Integer, ArrayList<Buyer>> buyersWithOrders,
                                           Map<Integer, Map<Integer, ArrayList<Producer>>> producersWithOrders) {

    List<Shipment> shipmentList = new ArrayList<>();
    Company closestHub;
    for (Buyer client : buyersWithOrders.get(day)) {
        closestHub = findNearestHub(client);
        Shipment s = new Shipment(client, closestHub);
        shipmentList.add(s);
        ArrayList<Product> clientOrder = client.getOrderByDay().get(day);
        boolean found = false;
        for (Product clientProduct : clientOrder) {
            if (found) {
                found = false;
            }
            ArrayList<Producer> producers = producersWithOrders.get(day).get(clientProduct.getId());
            if (producers != null) {
                for (Producer producer : producersWithOrders.get(day).get(clientProduct.getId())) {
                    if (found) continue;
                    ArrayList<Product> producerProduction = new ArrayList<>();
                    try {
                        producerProduction = producer.getOrderByDay().get(day).get(clientProduct.getId());
                    } catch (Exception e) {}
                    if (producerProduction != null) {
                        for (int j = producerProduction.size() - 1; j >= 0; j--) {
                            if (found) continue;
                            Product producerProduct = producerProduction.get(j);
                            if (producerProduct != null) {
                                if (producerProduct.getQuantity() >= clientProduct.getQuantity()) {
                                    s.addProducer(producer);
                                    s.addQuantity(clientProduct.getQuantity());
                                    s.addProduct(clientProduct);
                                    found = true;
                                    producer.subtractFromOrderByDay(producerProduct, clientProduct.getQuantity(), day, j);
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}

```

Figura 4 - Método generateShipmentList privado 1ª parte

```

    }
}

shipmentListOrderByDayWithoutRestrictions.put(day, shipmentList);

return shipmentList;
}

```

Figura 5 - Método generateShipmentList privado 2ª parte

A interface privada computa as n chamadas necessárias para que os produtos transitem de um dia para o outro caso sobrem, para o efeito é realizada a chamada do método *addLeftOver* após ser gerada a lista de expedição para um n dia, porque só após os produtos serem expedidos é que podemos realizar a verificação das sobras desse mesmo dia.

```

public Map<Integer, Map<Integer, ArrayList<Producer>>> addLeftOver(int day, Map<Integer, Map<Integer, ArrayList<Producer>>> producersWithOrders) {

    //create a map that has the products of the next day and the left over products of the current day
    Map<Integer, ArrayList<Producer>> leftOver = new HashMap<>();
    Map<Integer, ArrayList<Producer>> producersWithOrdersOfNextDay = producersWithOrders.get(day+1);
    Map<Integer, ArrayList<Producer>> producersWithOrdersOfCurrentDay = producersWithOrders.get(day);

    if (producersWithOrdersOfNextDay!=null) {
        for (Map.Entry<Integer, ArrayList<Producer>> entry : producersWithOrdersOfCurrentDay.entrySet()) {
            ArrayList<Producer> producers = entry.getValue();
            ArrayList<Producer> producersOfCurrentDay = new ArrayList<>();

            for (Map.Entry<Integer, ArrayList<Producer>> entry2 : producersWithOrdersOfNextDay.entrySet()) {
                if (entry.getKey().equals(entry2.getKey())) {
                    ArrayList<Producer> producers1 = entry2.getValue();
                    producersOfCurrentDay.addAll(producers1);
                    break;
                }
            }

            for (Producer producer : producers) {
                ArrayList<Product> products = producer.getOrderByDay().get(day).get(entry.getKey());
                for (Product product : products) {
                    product.decreaseDaysLeft();
                    if (product.getQuantity() > 0 && product.getDaysLeft() > 0) {

                        try {
                            if (producer.getOrderByDay().get(day + 1).get(product.getId()) != null) {
                                producer.getOrderByDay().get(day + 1).get(product.getId()).add(product);
                            } else {
                                ArrayList<Product> produtos = new ArrayList<>();
                                produtos.add(product);
                                producer.getOrderByDay().get(day + 1).put(product.getId(), produtos);
                            }

                            if (!producersOfCurrentDay.contains(producer)) {
                                producersOfCurrentDay.add(producer);
                            }
                        } catch (Exception e) {}
                    }
                }
            }

            leftOver.put(entry.getKey(), producersOfCurrentDay);
        }
    }

    producersWithOrders.replace(day+1, leftOver);
    return producersWithOrders;
}

```

Figura 6 - Método addLeftOver 1ªparte

```

    }
}

leftOver.put(entry.getKey(), producersOfCurrentDay);
}

producersWithOrders.replace(day+1, leftOver);
return producersWithOrders;
}

```

Figura 7 - Método addLeftOver 2ªparte

É necessário, por sua vez, para cada um dos clientes encontrar o hub mais próximo de cada cliente, sendo este obtido com o *findNearestHub* que utiliza o *shortestPaths* aliado a uma filtragem simples por hub.

```

public Company findNearestHub(Buyer client){

    ArrayList<LinkedList<DistributionNetworkMember>> paths = new ArrayList<>();
    ArrayList<Track> edges = new ArrayList<>();
    Double minDist = Double.MAX_VALUE;
    Company closestHub = null;

    if (Algorithms.shortestPaths(network, client, trackComparator, binaryOperator, zero, paths, edges)){
        for (DistributionNetworkMember hub : hubs){
            int key = network.key(hub);
            if(edges.get(key).getMiles() < minDist){
                minDist = edges.get(key).getMiles();
                closestHub = (Company) hub;
            }
        }
    }

    return closestHub;
}

```

Figura 8 - Método *findNearestHub*

### Análise da Complexidade

O algoritmo *generateShipmentList* é determinístico, pois não existe divergência entre a complexidade do pior e do melhor caso. Assumindo que D é o número de dias, C é o número de clientes, PC o número de produtos de cada cliente, P o número de produtores e PP o número de produtos do cliente para um n dia que sejam do mesmo tipo que o produto do cliente a complexidade deste algoritmo é  $O(DxCxPCxPxPP)$ .

O método *addLeftOver* é determinístico, pois não existe divergência entre a complexidade do pior e do melhor caso. Assumindo que E representa o número de entries, P o número de produtores e PP o número de produtos do cliente para um n dia que sejam do mesmo tipo que o produto do cliente a complexidade deste algoritmo é  $O(ExPxPP)$ .

O método *findNearestHub* é determinístico, pois não existe divergência entre a complexidade do pior e do melhor caso. A complexidade deste método  $O(V^2)$ .

## US309

Nesta US é pedido para gerar uma lista de expedição para um determinado dia com os N produtores agrícolas mais próximos do hub de entrega do cliente e que contenha informação sobre os produtores que abastecem o cliente, os produtos que cada produtor irá entregar para esse dia, a quantidade de cada produto, o hub de entrega e o cliente.

Para o efeito, foram desenvolvidos dois métodos *generateShipmentList* um será, portanto, a interface pública para evitar a já mencionada *Object Mutation* que iriam afetar a chamada posterior da funcionalidade da US 308 gerando dados conflituosos que não estariam a representar a realidade.

```
1211415 +1
public List<Shipment> generateShipmentList(int days, int n) {

    if(!producersWithOrders.containsKey(days)) return null;

    Map<Integer, Map<Integer, ArrayList<Producer>>> producersWithOrders = new HashMap<>(this.producersWithOrders);
    Map<Integer, ArrayList<Buyer>> buyersWithOrders = new HashMap<>(this.buyersWithOrders);

    for (int i = 1; i < days; i++) {
        generateShipmentList(i, n, buyersWithOrders, producersWithOrders);
        producersWithOrders = addLeftOver(i, producersWithOrders);
    }
    return generateShipmentList(days, n, buyersWithOrders, producersWithOrders);
}
```

Figura 9 -Método *generateShipmentList* público



A interface privada computa as n chamadas necessárias para que os produtos transitem de um dia para o outro caso sobrem, para o efeito é realizada a chamada do método *addLeftOver* após ser gerada a lista de expedição para um n dia, porque só após os produtos serem expedidos é que podemos realizar a verificação das sobras desse mesmo dia.

É necessário, por sua vez, para cada um dos clientes encontrar o hub mais próximo de cada cliente, sendo este obtido com o *findNearestHub* que utiliza o *shortestPaths* aliado a uma filtragem simples por hub.

```
private List<Shipment> generateShipmentList(int day, int n, Map<Integer, ArrayList<Buyer>> buyersWithOrders,
                                           Map<Integer, Map<Integer, ArrayList<Producer>>> producersWithOrders) {

    /*Retornar null se o número de produtores for menor ou igual a 0*/

    if (n <= 0) {
        throw new IllegalArgumentException("0 número de produtores tem de ser maior que 0");
    }

    /* Retornar null se não existirem hubs definidos no sistema */

    if (hubs.size() == 0) {
        throw new IllegalArgumentException("Não existem hubs definidos no sistema");
    }

    boolean flag = false;
    /*Declarar e inicializar a lista de expedições*/

    List<Shipment> shippingList = new ArrayList<>();

    /*Declarar e inicializar as listas com os dados necessários para criar um Shipment(expedição)*/

    Map<String, Producer> produtor = new HashMap<>();
    List<Double> qntyExpedido = new ArrayList<>();
    List<Product> produto = new ArrayList<>();

    /*Declarar e inicializar a lista que armazena todos os clientes e empresas no grafo */

    ArrayList<Buyer> clientes = buyersWithOrders.get(day);

    /*Iterar a lista de clientes no grafo*/

    if (clientes != null) {
        for (Buyer member : clientes) {

            ArrayList<Product> products = member.getOrderByDay().get(day);
```

Figura 10 -Método *generateShipmentList* privado 1ªparte

```

Company hub = findNearestHub(member);

for (Product product : products) {
    flag = false;
    /* Obtemos os N produtores mais próximos para o hub obtido */

    List<Producer> closestProducers = findNearestProducers(hub, n, day, product, producersWithOrders);

    // Itera-se a lista dos N produtores mais próximos ao hub
    if (closestProducers != null) {
        for (Producer producer : closestProducers) {
            // Obtemos a lista de produtos do produtor para o dia em específico
            List<Product> productsForDayOfProducer = producer.getOrderByDay().get(day).get(product.getId());

            /* Iteramos a lista de produtos dos clientes por cada iteração da lista de produtos do produtor e verificamos
            se possuem o mesmo produto, caso sim, verifica-se então se o produtor pode suprir ao cliente, cujo caso é adicionado
            às listas os dados para criação de um Shipment */

            if (productsForDayOfProducer != null) {
                for (int i = productsForDayOfProducer.size() - 1; i >= 0; i--) {
                    Product prodProducer = productsForDayOfProducer.get(i);
                    if (product.getQuantity() <= prodProducer.getQuantity()) {
                        produto.add(product);
                        produtor.putIfAbsent(producer.getLocId(), producer);
                        qntyExpedido.add(product.getQuantity());
                        // Restamos a quantidade do produto vendido do produtor
                        producer.subtractFromOrderByDay(prodProducer, product.getQuantity(), day, i);
                        flag = true;
                        break;
                    }
                }
            }
            if (flag) {
                break;
            }
        }
    }
}

shippingList.add(new Shipment(member, hub, produtor, qntyExpedido, produto));

```

Figura 11 - Método generateShipmentList privado 2ªparte

Depois é iterado os produtos do dia de um cliente e é procurado os N produtores mais próximos ao hub obtido por cliente e que fornecem para o produto que esta a ser iterado nesse momento. Para isso utilizamos o *findNearestProducers* que utiliza o *shortestPaths* aliado a uma filtragem de produtores que fornecem para esse produto.

```
/**
 * Algoritmo de Dijkstra que devolve uma lista com os N produtores mais próximos de um dado hub para um determinado dia
 *
 * @param hub Information of the Vertex that represents the hub
 * @param n Number of producers to find
 * @param day Day to find the producers
 * @param product Product to find the producers
 * @return List of producers
 */

//usage: 1211415
private List<Producer> findNearestProducers(DistributionNetworkMember hub, int n, int day, Product product,
                                           Map<Integer, Map<Integer, ArrayList<Producer>>> producersWithOrders ) {

    List<Producer> producers = new ArrayList<>();

    //ALL: Implementar o algoritmo de Dijkstra para encontrar os N produtores mais próximos de um dado hub para um determinado dia
    if (product!=null) {
        if (producersWithOrders.get(day)!=null&&producersWithOrders.get(day).get(product.getId())!=null){
            ArrayList<Producer> vOrigins = producersWithOrders.get(day).get(product.getId());
            Map<Double,Producer> mapMilesByProducer = new HashMap<>();
            ArrayList<Double> miles = new ArrayList<>();

            ArrayList<LinkedList<DistributionNetworkMember>> paths = new ArrayList<>();
            ArrayList<Track> dists = new ArrayList<>();

            if (Algorithms.shortestPaths(network, hub,trackComparator,binaryOperator,zero, paths, dists)) {
                for (Producer vOrig: vOrigins){
                    int key = network.key(vOrig);
                    mapMilesByProducer.put(dists.get(key).getMiles(),vOrig);
                    mapMilesByProducer.put(dists.get(key).getMiles(),vOrig);
                    miles.add(dists.get(key).getMiles());
                }
            }
            Collections.sort(miles);

            if (n>miles.size()) n=miles.size();

            for (int i = 0; i < n; i++) {
```

Figura 12 - Método *findNearestProducers*

## Análise da Complexidade

O algoritmo *generateShipmentList* é determinístico, pois não existe divergência entre a complexidade do pior e do melhor caso. Assumindo que D é o número de dias, C é o número de clientes, NPC o número de produtos de cada cliente, N o número de produtores e PP o número de produtos do cliente para um n dia que sejam do mesmo tipo que o produto do cliente a complexidade deste algoritmo é  $O(DxCxNPCxN \times PP)$ .

O método *addLeftOver* é determinístico, pois não existe divergência entre a complexidade do pior e do melhor caso. Assumindo que E representa o número de entries, P o número de produtores e PP o número de produtos do cliente para um n dia que sejam do mesmo tipo que o produto do cliente a complexidade deste algoritmo é  $O(E \times P \times PP)$ .

O método *findNearestHub* é determinístico, pois não existe divergência entre a complexidade do pior e do melhor caso. A complexidade deste método  $O(V^2)$ .

O método *findNearestProducers* é determinístico, pois não existe divergência entre a complexidade do pior e do melhor caso. A complexidade deste método  $O(V^2 + N)$ .

## US310

Nesta US é pedido para gerar o percurso de entrega que minimiza a distância total percorrida para uma lista de expedição diária. Para isso foram criados dois métodos: *generateDeliveryRoutes*, que decide qual a lista de expedição que a utilizar baseado num dia e numa opção passada por parâmetro, e *generateDeliveryRoutesWithShipmentList* que, com base numa lista de expedição vai gerar um conjunto de rotas de entrega. Este método cria uma estrutura do tipo *Map<Shipment,ArrayList<Delivery>>* que associa uma lista de rotas (*Delivery*) por cada *Shipment* da lista de expedição. *Delivery* é a classe que guarda todos os vértices da rota e a distância total de cada rota e *Shipment* é a classe já criada na US308 e na US309.

O método *generateDeliveryRoutesWithShipmentList* começa por verificar se a lista de expedição não é nula (isto pode acontecer caso ainda não tenha sido gerada a lista de expedição para aquele dia). Depois itera por todos o *Shipments* da lista de expedição e utiliza o método *shortestPaths* para calcular as rotas de menor distância desde o *Hub* onde o *Shipment* vai ser entregue até os restantes vértices da network. Logo a seguir, por cada *Producer* associado a esse *Shipment*, é criada uma *Delivery* utilizando o caminho mais curto entre o *Producer* e o *Hub* e a respetiva distância. No final todos os dados são colocados na estrutura criada para este caso.

```
private Map<Shipment,ArrayList<Delivery>> generateDeliveryRoutesWithShipmentList(List<Shipment> shipmentList){
    if (shipmentList == null){
        return null;
    }

    Map<Shipment,ArrayList<Delivery>> finalRoutes = new HashMap<>();

    for (Shipment shipment : shipmentList) {
        ArrayList<Delivery> shipmentRoutes = new ArrayList<>();

        ArrayList<LinkedList<DistributionNetworkMember>> paths = new ArrayList<>();
        ArrayList<Track> dists = new ArrayList<>();
        Algorithms.shortestPaths(network,shipment.getHub(),trackComparator,binaryOperator,zero,paths,dists);

        for (Producer producer : shipment.getProdutor().values()){
            LinkedList<DistributionNetworkMember> path = paths.get(network.key(producer));
            Collections.reverse(path);
            shipmentRoutes.add(new Delivery(path,dists.get(network.key(producer))));
        }
        finalRoutes.put(shipment,shipmentRoutes);
    }

    return finalRoutes;
}
```

Figura 13 - Método *generateDeliveryRoutesWithShipmentList*

### Análise de complexidade

Este método é determinístico, porque vai sempre percorrer todos os *Shipments* que existem para a lista passada por parâmetro, calculando o *shortestPaths* em cada uma das iterações.

Para além disso, considerando *S* o número *Shipments* na lista expedição e *V* o número de vértices do grafo, podemos concluir que sua complexidade é  $O(S \times V^2)$ , já que, por cada *Shipment* existente, utilizamos o método *shortestPaths* com complexidade  $O(V^2)$  que é o bloco de código que mais influencia a complexidade do código dentro do primeiro ciclo *for*

## US311

Nesta US é pedido para gerar estatísticas para uma lista de expedição, por Cabaz, Cliente, Produtor e Hub, para isso criou-se uma classe chamada *ShipmentsAnalysisAlgoritms* que tem por sua vez 4 métodos principais:

```
public static ArrayList<BasketAnalysis> basketAnalyticsPerShipmentList(List<Shipment> shipments, int day){  
  
    int numberOfProductsTotallySatisfied;  
    int numberOfProductsNotSatisfied;  
    double satisfiedPercentage;  
    int numberOfProducers;  
  
    ArrayList<BasketAnalysis> listBasketAnalytics = new ArrayList<>();  
    for (Shipment ship: shipments) {  
        if (!ship.getProdutor().isEmpty()) {  
            numberOfProductsTotallySatisfied = ship.getProduto().size();  
            numberOfProducers = ship.getProdutor().entrySet().size();  
            satisfiedPercentage = ( ((double)(numberOfProductsTotallySatisfied)/  
                                   ship.getClient().getOrderByDay().get(day).size())*100);  
        }else{  
            numberOfProductsTotallySatisfied = 0;  
            numberOfProducers = 0;  
            satisfiedPercentage = 0;  
        }  
        numberOfProductsNotSatisfied = (ship.getClient().getOrderByDay().get(day).size() - ship.getProduto().size());  
        listBasketAnalytics.add(new BasketAnalysis(numberOfProductsTotallySatisfied,  
                                                    numberOfProductsNotSatisfied, satisfiedPercentage, numberOfProducers));  
    }  
    return listBasketAnalytics;  
}
```

Figura 14 – Método *basketAnalyticsPerShipmentList*

O *basketAnalyticsPerShipmentList* obtém por cabaz o 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. Sendo um método determinístico em a sua complexidade é  $O(S)$  sendo S o tamanho de de *shipments*

```

public static ArrayList<ClientAnalysis> clientAnalyticsPerShipmentList(List<Shipment> shipments, int day, List<Buyer> buyers) {

    int numberOfBasketsTotallySatisfied;
    int numberOfBasketsPartiallySatisfied;
    Map<String, Producer> mapProducers;

    ArrayList<ClientAnalysis> listClientAnalysis = new ArrayList<>();

    for (Buyer b : buyers) {
        numberOfBasketsTotallySatisfied = 0;
        numberOfBasketsPartiallySatisfied = 0;
        mapProducers = new HashMap<>();
        for (Shipment ship : shipments) {
            if (ship.getClient().equals(b)) {
                if (!ship.getProdutor().isEmpty()) {
                    if ((ship.getClient().getOrderByDay().get(day).size() - ship.getProdutor().size()) == 0) {
                        numberOfBasketsTotallySatisfied++;
                        addProducer(ship, mapProducers);
                    } else {
                        numberOfBasketsPartiallySatisfied++;
                        addProducer(ship, mapProducers);
                    }
                }
            }
        }
        listClientAnalysis.add(new ClientAnalysis(b, numberOfBasketsTotallySatisfied,
            numberOfBasketsPartiallySatisfied, mapProducers.entrySet().size()));
    }

    return listClientAnalysis;
}

```

Figura 15 - Método *clientAnalyticsPerShipmentList*

O *clientAnalyticsPerShipmentList* obtém por cliente o nº de cabazes totalmente satisfeitos, nº de cabazes parcialmente satisfeitos e nº de fornecedores distintos que forneceram todos os seus cabazes. Sendo que utiliza também o método *addProducer*, adiciona os produtores distintos por cada shipment a uma lista.

```

private static Map<String, Producer> addProducer(Shipment ship, Map<String, Producer> mapProducers){

    for(Producer p: ship.getProdutor().values()) {
        mapProducers.putIfAbsent(p.getLocId(), p);
    }

    return mapProducers;
}

```

Figura 16 - Método *addProducer*

O método *clientAnalyticsPerShipmentList* é não determinístico pois no melhor caso não existe nenhum cliente no shipment, tendo assim uma complexidade de  $O(B \times S)$  e no pior caso a complexidade é de  $O(B * S * P)$ , sendo B o número de buyers e S o número de shipments, e P o número de produtores em cada shipment.

```

public static ArrayList<ProducerAnalysis> producerAnalyticsPerShipmentList(List<Shipment> shipments,int day) {

    Map<String,Producer> mapProducer = new HashMap<>();
    Map<String,Buyer> mapBuyer;
    Map<String, Company> mapHub;
    int numberOfProductsDepleted;
    int numberOfBasketsFullySatisfied;
    int numberOfBasketsPartiallySatisfied;
    ArrayList<ProducerAnalysis> listProducerAnalysis = new ArrayList<>();

    for (Shipment ship : shipments) {
        addProducer(ship, mapProducer);
    }

    for(Producer prod: mapProducer.values()){
        mapHub = new HashMap<>();
        mapBuyer = new HashMap<>();
        numberOfProductsDepleted = numberOfDepletedProducts(prod,day);
        numberOfBasketsFullySatisfied = 0;
        numberOfBasketsPartiallySatisfied =0;
        for (Shipment ship : shipments){
            if (ship.getProdutor().get(prod.getLocId())!=null){
                if (ship.getProdutor().entrySet().size() == 1 && (ship.getClientes().getOrderByDay().get(day).size() - ship.getProduto().size()) == 0){
                    numberOfBasketsFullySatisfied++;
                }else{
                    numberOfBasketsPartiallySatisfied++;
                }
                mapBuyer.computeIfAbsent(ship.getClientes().getLocId(), k -> ship.getClientes());
                mapHub.computeIfAbsent(ship.getHub().getLocId(), k -> ship.getHub());
            }
        }
        listProducerAnalysis.add(new ProducerAnalysis(prod,numberOfProductsDepleted,
            numberOfBasketsFullySatisfied,numberOfBasketsPartiallySatisfied,mapBuyer.entrySet().size(),mapHub.entrySet().size()));
    }
    return listProducerAnalysis;
}

```

Figura 17 - Método *producerAnalyticsPerShipmentList*

O método *producerAnalyticsPerShipmentList* obtém por produtor 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. Este também método também recorre ao *addProducers* mas neste caso visa obter todos os Produtores que forneceram produtos para uma determinada lista de expedição. Como também ao método *numberOfDepletedProducts*.

```

private static int numberOfDepletedProducts(Producer prod, int day){

    boolean depleted;
    int numberDepleted = 0;
    Collection<ArrayList<Product>> values = prod.getOrderByDay().get(day).values();
    for (ArrayList<Product> id : values){
        depleted= false;
        for (int i = id.size()-1; i>=0; i--) {
            if (id.get(i).getQuantity() == 0){
                depleted = true;
            }else{
                depleted =false;
            }
        }
        if (depleted){
            numberDepleted++;
        }
    }

    return numberDepleted;
}

```

Figura 18 - Método *numberOfDepletedProducts*

Este método verifica o número de produtos totalmente esgotados para um determinado *Producer*.

Analisando a então o método *producerAnalyticsPerShipmentList* temos que ele é determinístico tendo uma complexidade de  $O(P \times Prd \times Val \times S)$  sendo *P* o numero de Produtores que forneceram cabaz, *Prd* o numero de Produtos que cada Produtor tem, o numero de produtos com Validade diferente e *S* o numero de shipments.



```

public static ArrayList<HubAnalysis> hubAnalyticsPerShipmentList(List<Shipment> shipments) {

    Map<String,Company> mapHub = new HashMap<>();
    Map<String,Buyer> mapBuyer;
    Map<String,Producer> mapProducer;
    ArrayList<HubAnalysis> listHubAnalysis = new ArrayList<>();

    for (Shipment s: shipments) {
        mapHub.computeIfAbsent(s.getHub().getLocId(), k-> s.getHub());
    }

    for (Company hub: mapHub.values()){
        mapBuyer = new HashMap<>();
        mapProducer = new HashMap<>();
        for (Shipment ship: shipments) {
            if (ship.getHub().equals(hub)) {
                mapBuyer.computeIfAbsent(ship.getClient().getLocId(), k -> ship.getClient());
                addProducer(ship, mapProducer);
            }
        }
        listHubAnalysis.add(new HubAnalysis(hub,mapBuyer.entrySet().size(),mapProducer.entrySet().size()));
    }

    return listHubAnalysis;
}

```

Figura 19 - Método *hubAnalyticsPerShipmentList*

O *hubAnalyticsPerShipmentList* obtém por hub nº de clientes distintos que recolhem cabazes em cada hub, nº de produtores distintos que fornecem cabazes para o hub. Este método recorre também ao método *addProducer* para adicionar a uma lista os produtores distintos que fornecem a cada hub.

Este método é determinístico tendo uma complexidade de  $O(H \times S \times P)$ , sendo H o número de hubs, o S o número de Shipments e P o número de produtor por shipments.

Por fim existe o método dentro da class *DistributionNetwork* chamado *generateAnalysis*.

```

public ShipmentAnalysis generateAnalysis(int option, int day){

    List<Shipment> shipment;
    if (option == 1){
        shipment = shipmentListOrderByDayWithoutRestrictions.get(day);
    }else{
        shipment = shipmentListOrderByDayWithRestrictions.get(day);
    }

    if (shipment != null){
        ArrayList<BasketAnalysis> listBasketAnalytics = ShipmentsAnalysisAlgorithms.basketAnalyticsPerShipmentList(shipment, day);
        ArrayList<ClientAnalysis> listClientAnalytics = ShipmentsAnalysisAlgorithms.clientAnalyticsPerShipmentList(shipment, day, buyersWithOrders.get(day));
        ArrayList<ProducerAnalysis> listProducerAnalysis = ShipmentsAnalysisAlgorithms.producerAnalyticsPerShipmentList(shipment, day);
        ArrayList<HubAnalysis> listHubAnalysis = ShipmentsAnalysisAlgorithms.hubAnalyticsPerShipmentList(shipment);

        return new ShipmentAnalysis(listBasketAnalytics, listClientAnalytics, listProducerAnalysis, listHubAnalysis);
    }

    return null;
}

```

Figura 20 - Método *generateAnalysis*

Este método chama todos os métodos de geração de estatísticas e reúne toda a informação numa só class chamada *ShipmentAnalysis*. A complexidade deste método irá equivaler à pior complexidade dos métodos de análise, logo será de  $O(P \times Prd \times Val \times S)$  sendo P o número de produtores que forneceram cabaz, Prd o número de produtos que cada produtor tem, o número de produtos com validade diferente e S o número de shipments.