# RELATÓRIO PROJETO
# REDES DE DISTRIBUIÇÃO (PI)

## ESTRUTURAS DE INFORMAÇÃO

INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO


André Barros _ *1211299*

Carlos Lopes _ *1211277*

Ricardo Moreira _ *1211285*

Tomás Russo _ *1211288*

Tomás Lopes _ *1211289*

# Índice

# O Problema

O objetivo deste projeto é criar uma solução informática que apoie a administração de uma empresa responsável pela gestão de uma instalação agrícola em modo de produção biológico (MPB). Neste projeto são explorados vários aspetos relevantes na administração de uma exploração agrícola em MPB, nomeadamente: a gestão de campos, culturas, regas, comercialização de produtos agrícolas, gestão de informação recolhida a partir de sensores meteorológicos e gestão de armazéns agrícolas.

No presente relatório, encontra-se a explicação e análise de toda a solução necessária para a componente proposta pela unidade curricular de Estruturas de Informação que aborda principalmente a gestão das redes de distribuição.

Desta forma, as redes devem ser geridas de uma forma eficiente usando as estruturas mais adequadas para o propósito.

# A Solução

## Diagrama de classes

Toda a estrutura de dados utilizada foi abstraída por outras classes que permitem manipular os dados através de uma interface específica que não depende da estrutura utilizada internamente.

Desta forma, ganhamos flexibilidade e permite que cada módulo seja testado independentemente dos outros com todas as suas especificidades e condições.

Depois de analisado o problema apresentado, a solução encontrada foi a seguinte:

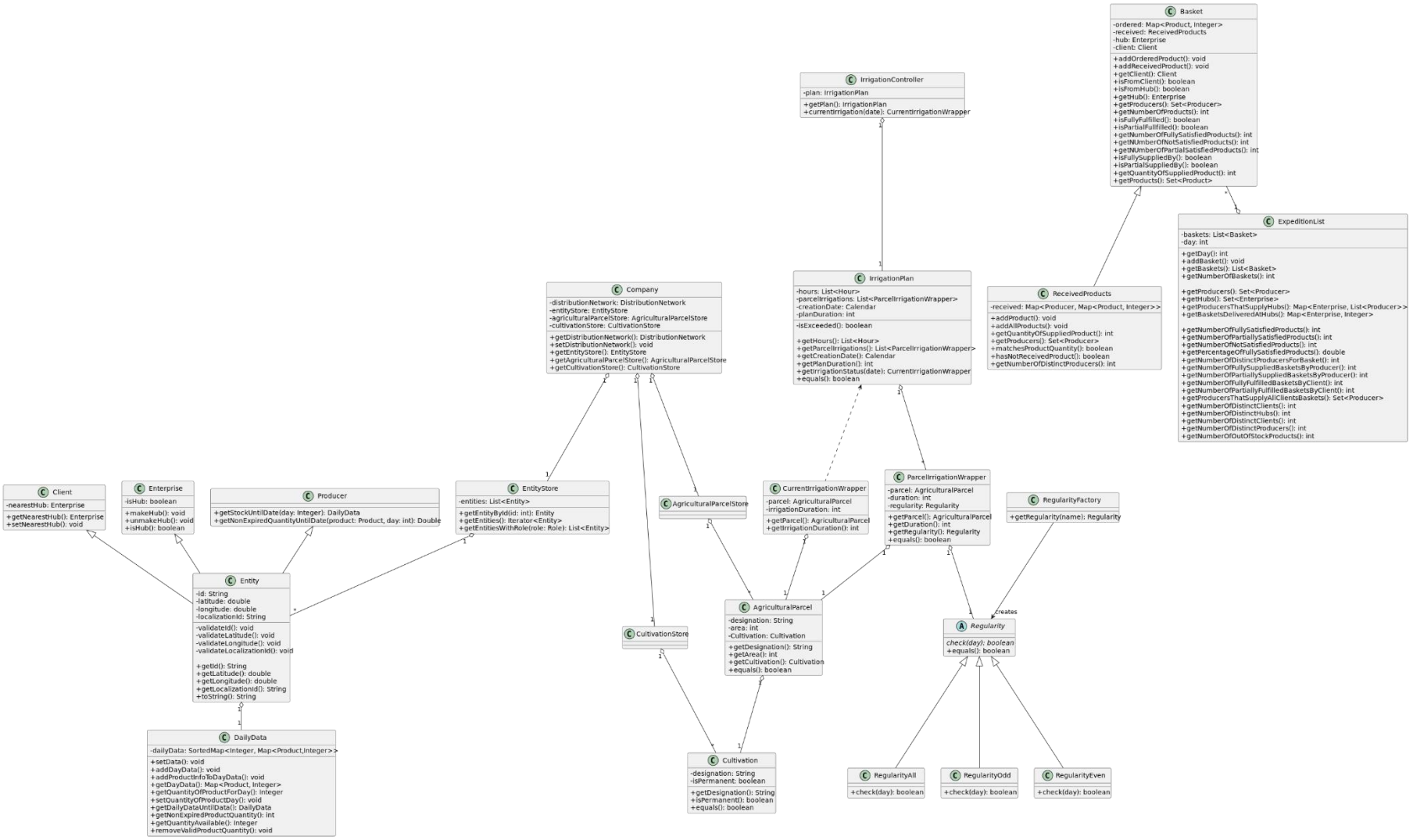


*Figura 1 - Diagrama de classes*

A classe *DailyData*, contém um mapa onde a *key* é um inteiro que representa um dia e o *value* é outro mapa onde a *key* é um produto e a chave é um inteiro que representa a quantidade desse mesmo produto.

As classes *Client* e *Producer*, estendem a classe *Entity*, desta forma, o atributo do tipo *DailyData* representa na classe *Client* as suas encomendas e na classe *Producer* as suas produções.

A classe *ExpeditionLists* representa uma lista de expedições de um determinado dia, assim, esta possui uma chave do tipo *Integer* que representa o dia a que a lista de expedição se refere e uma lista de cabazes (*Baskets)*.

Por sua vez, a classe *Basket* representa os cabazes expedidos e contém os seguintes atributos:

- *Client* – cliente que efetuou a encomenda do cabaz;
- *Hub* – empresa onde o cliente levanta o cabaz;
- *Ordered* – mapa onde a *key* é o produto encomendado e o *value* é um *Integer* que representa a quantidade encomendada desse produto;
- *Received* – mapa onde a *key* é o produtor que forneceu o produto e o *value* é outro mapa onde a *key* é o produto fornecido e o *value* é um *Integer* que representa a quantidade fornecida desse produto.

# Estruturas auxiliares

## Grafo

A estrutura base para resolver o problema apresentado é o grafo. Esta estrutura representa a informação como um conjunto de vértices e arestas conectados entre si de forma obter a representação adequada das conexões físicas entre as diferentes entidades.

Para a implementação do grafo foi usado o mapa de adjacências devido às características do projeto. Por exemplo: quanto à inserção de vértices, esta implementação apresenta vantagens a nível assintótico em relação, por exemplo, à matriz de adjacências, apresentando, respetivamente, ordens de complexidade O(1) e $O(V^2)$.

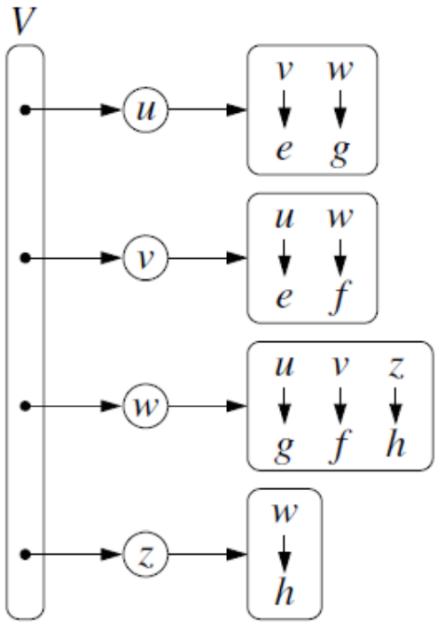


*Figura 2 - Representação gráfica de um mapa de adjacências*

O mapa de adjacências associa, para cada vértice do grafo, um mapa cujas chaves são todos os vértices adjacentes do vértice inicial, e sendo os valores os repetitivos pesos da aresta que une esses dois vértices. No exemplo acima, o vértice $u$ tem como adjacentes os vértices $v$ e $w$, onde a aresta que une $u$ e $v$ tem peso $e$, e a aresta que une $u$ e $w$ tem peso $g$.

# Algoritmos usados

## Sprint 1

### *US301 – Construir a rede de distribuição de cabazes _ Tomás Lopes*

Esta funcionalidade consiste em carregar a informação presente nos ficheiros apresentados para um grafo que representa a rede de distribuição de cabazes.

```
public LoadDistributionNetworkController(EntityStore entityStore, List<Map<String, String>> distancesData) {
  this.entityStore = entityStore;
  this.distancesData = distancesData;
}
```

*Figura 3 - Construtor LoadDistributionNetworkController*

```
public DistributionNetwork loadDistributionNetwork() {
  DistributionNetwork distributionNetwork = new DistributionNetwork();

  ...
  for (Map<String, String> distance : distancesData) {
    String id1 = distance.get(Constants.DISTANCES_LOC1_ID_FIELD_NAME);
    String id2 = distance.get(Constants.DISTANCES_LOC2_ID_FIELD_NAME);
    int distanceValue = Integer.parseInt(distance.get(Constants.DISTANCES_DISTANCE_FIELD_NAME));

    Entity entity1 = this.entityStore.getEntityByLocalizationId(id1);
    Entity entity2 = this.entityStore.getEntityByLocalizationId(id2);

    if (entity1 != null && entity2 != null)
      distributionNetwork.addRelation(entity1, entity2, distanceValue);
  }

  return distributionNetwork;
}
```

*Figura 4 - Load Distribution Network*

Desta forma, mapeamos as entidades para as suas respetivas localizações para serem usadas posteriormente, de forma a carregar a distância entre entidades para o grafo que representa a rede de distribuição.

Esta operação é de ordem O(*E*) onde *E* representa o número de relações entre entidades e consequentemente o número de arestas no grafo de suporte. Depois de percorridas todas as relações é necessário inseri-la no grafo de suporte que, como referido anteriormente, dada a implementação usada é uma operação de ordem O(1). Desta forma, a inserção de entidades na rede de distribuição é de ordem O(E), isto considerando que aas entidades já estão inseridas no repositório *EntityStore*.

A inserção na classe *EntityStore* é de ordem O(V) em que V representa o número de entidades e consequente número de vértices no grafo de suporte.

A classe *DistributionNetwork* abstrai a manipulação direta do grafo de forma a ganhar flexibilidade no uso da rede por outras funcionalidades.

### *US302 – Conexividade e número mín. ligações para qualquer cliente contactar qualquer outro _ Ricardo Moreira*

Esta funcionalidade permite ao utilizador saber se a rede de distribuição carregada é conexa, bem como descobrir qual o menor número de ligações necessário para que quaisquer dois clientes consigam-se conectar (ou seja, o diâmetro do grafo).

```
public GetMinNumberOfConnectionsController(DistributionNetwork distributionNetwork) {
  this.distributionNetwork = distributionNetwork;
}

public int getMinimumNumOfConnections() {
  if (distributionNetwork == null)
    throw new IllegalArgumentException("Invalid distribution network");

  if (!distributionNetwork.isConnected())
    throw new IllegalArgumentException("Distribution network is not connected");

  int[][] shortestPaths = distributionNetwork.shortestPathsBetweenAllNodes();

  int max = shortestPaths[0][0];

  for (int i = 0; i < shortestPaths.length; i++) {
    for (int j = 0; j < shortestPaths[i].length; j++) {
      if (shortestPaths[i][j] > max)
        max = shortestPaths[i][j];
    }
  }

  return max;
}

public boolean isConnected() {
  return distributionNetwork.isConnected();
}
```

*Figura 5 - Controlador*

Um grafo é conexo caso exista um caminho o qual passa por todos os vértices do mesmo. O método utilizado no nosso projeto para descobrir a conectividade foi efetuar uma pesquisa em largura e, caso o número de vértices percorridos for diferente do número total de vértices do grafo, podemos confirmar que o grafo que suporta a rede de distribuição não é conexo.

Sabendo que o método usa a pesquisa em largura, e que esta pesquisa visita todos os vértices e ligações pelo menos uma vez, podemos concluir que a complexidade deste algoritmo é linear, $O(V \cdot E)$.

```java
public static <V, E> boolean isConnected(Graph<V, E> g) {
  LinkedList<V> bfs = BreadthFirstSearch(g, g.vertices().iterator().next());
  return isConnected(g, bfs);
}

private static <V, E> boolean isConnected(Graph<V, E> g, LinkedList<V> bfs) {
  return bfs != null && bfs.size() == g.numVertices();
}
```

*Figura 6 - Método conectividade*

Para calcular o menor número de ligações entre os vértices mais distantes, foi usada uma implementação semelhante ao algoritmo *Floyd-Warshall*. Este método calcula o número de ligações dos caminhos mais curtos entre todos os pares de vértices. Ao obter esta matriz, para encontrar o número mínimo de ligações para que qualquer vértice consiga conectar qualquer outro, é só encontrar a maior entrada da matriz.

```java
public int[][] shortestPathsBetweenAllNodes() {
  return GraphAlgorithms.minEdges(network, Integer::compare, Integer::sum, 0);
}

public static <V, E> int[][] minEdges(Graph<V, E> g, Comparator<E> ce, BinaryOperator<E> sum, E zero) {
  int numVerts = g.numVertices();
  int[][] minEdges = new int[numVerts][numVerts];

  for (int i = 0; i < numVerts; i++)
    for (int j = 0; j < numVerts; j++)
      minEdges[i][j] = -1;

  for (int i = 0; i < numVerts; i++)
    minEdges[i][i] = 0;

  for (Edge<V, E> e : g.edges()) {
    int vOrig = g.key(e.getVOrig());
    int vDest = g.key(e.getVDest());
    minEdges[vOrig][vDest] = 1;
  }

  for (int k = 0; k < numVerts; k++)
    for (int i = 0; i < numVerts; i++)
      for (int j = 0; j < numVerts; j++)
        if (minEdges[i][k] != -1 && minEdges[k][j] != -1)
          if (minEdges[i][j] == -1 || minEdges[i][j] > minEdges[i][k] + minEdges[k][j])
            minEdges[i][j] = minEdges[i][k] + minEdges[k][j];

  return minEdges;
}
```

*Figura 7 - Método Floyd-Warshall modificado*

Este algoritmo contém três *loops* que percorrem os vértices do grafo, o que faz com que este método tenha complexidade $O(V^3)$.

## *US303 – Definir os hubs da rede de distribuição _ Carlos Lopes*

Esta funcionalidade permite definir um certo número (*N*) de *Hubs* na rede de distribuição. Um *Hub* é obrigatoriamente uma empresa e carateriza-se pela facilidade de acesso por parte de todas as outras entidades representadas na rede de distribuição. Assim, definir os *N hubs* numa rede de distribuição passa por encontrar as empresas com melhor acessibilidade (melhor média de caminhos mínimos entre esta e todas as outras entidades) e torná-las *hubs* garantindo que as restantes não o sejam.

No construtor *controller* desta *user story* é definida a rede de distribuição sobre a qual se vai trabalhar (*network*) e no método *defineHubs* é passado como parâmetro o número de *hubs* a definir (*numberOfHubs*).

```java
public class DefineHubsController {
  private DistributionNetwork network;

  public DefineHubsController(DistributionNetwork network) {
    this.network = network;
  }

  public List<Enterprise> defineHubs(int numberOfHubs) throws InvalidNumberOfHubsException {
    return network.defineHubs(numberOfHubs);
  }
}
```

*Figura 8 - DefineHubs Controller Class*

Este método chama um método da *DistributionNetwork* passando por parâmetro o número de *hubs* a definir. Nesse método, após a validação desse número e verificada a conexidade do grafo, é necessário obter todas as empresas presentes no grafo, para depois as iterar e verificar a média de caminhos mínimos para todos os pontos do grafo. À medida que estas são iteradas, casos estejam previamente definidas como *hub*, deixam de estar.

As empresas são guardadas numa lista, juntamente com a média de caminhos mínimos calculada. Essa lista recebe uma *Entry* onde a *key* é a empresa e o *value* é a média.

```java
public List<Enterprise> defineHubs(int numberOfHubs) throws InvalidNumberOfHubsException {
    if (numberOfHubs <= 0)
      throw new InvalidNumberOfHubsException();


      if (!this.isConnected())
      return null;

    List<Map.Entry<Enterprise, Integer>> list = new ArrayList<>();
    List<Enterprise> enterprises = this.getEnterprises();

    for (int i = 0; i < enterprises.size(); i++) {
      Enterprise e1 = enterprises.get(i);

      // if e1 was a Hub before "unMakes" it
      e1.unMakeHub();

      int average = this.getAveragePathDistanceBetweenGroupOfEntities(e1);

      list.add(new AbstractMap.SimpleEntry<Enterprise, Integer>(e1, average));
    }
  (...)
```

*Figura 9 - Método defineHubs da DistributionNetwork*

O método *getEnterprises* utiliza um método que, uma classe, itera os vértices todos do grafo e retorna numa lista, todos esses vértices, tendo assim complexidade linear O(V).

```java
public List<Enterprise> getEnterprises() {
    return network.getEntitiesWithClass(Enterprise.class);
  }
```

*Figura 10 - Método getEnterprises*

Por sua vez, o método *getAveragePathDistanceBetweenGroupOfEntities*, utiliza um método *shortestPaths* da classe *GraphAlgoritms*, onde, passando por parâmetro o grafo, o vértice, um comparador, um operador binário, um inteiro representativo de zero, uma lista para guardar os caminhos e uma lista para guardas as distâncias dos caminhos, calcula todos os caminhos mínimos entre o vértice passado por parâmetro e todos os outros vértices do grafo. Por fim, retorna a média das distâncias desses caminhos.

```java
public int getAveragePathDistanceBetweenGroupOfEntities(Entity e1) {

    ArrayList<Integer> distances = new ArrayList<>();
    GraphAlgorithms.shortestPaths(network, e1, Integer::compareTo, Integer::sum, 0, new ArrayList<>(), distances);

    int sum = 0;
    int count = distances.size();
    for (int i = 0; i < count; i++) {
      sum += distances.get(i);
    }

    return sum / count;
}
```

*Figura 11 - getAveragePathBetweenGroupOfEntities*

Este método apresenta uma complexidade $O(V^2)$, pois é necessário, iterar todos os vértices e para cada um calcular o menor caminho possível.

Tendo todas as médias dos caminhos mínimos, basta apenas ordenar a lista pelas suas médias de forma crescente e tornar hubs as N empresas com menores médias, retornando as numa lista. O método de ordenação utilizado foi o *Merge Sort* que apresenta uma complexidade O(n.log(n)).

```java
  (...)
    // order list
    list = new MergeSort<Map.Entry<Enterprise, Integer>>().sort(list, cmp);

    // make N enterprises Hubs and return them in a list
    List<Enterprise> result = new ArrayList<>();
    for (int i = 0; i < numberOfHubs; i++) {
      try {
        Enterprise hub = list.get(i).getKey();
        hub.makeHub();
        result.add(hub);
      } catch (Exception E) {
        System.out.printf("There are only %d number of Enterprises\n", i);
        break;
      }
    }

    return result;
  }
```

*Figura 12 - continuação método defineHubs*

Assim, percorrendo as N empresas e calculando a média de distancia mínima entre estas e todos os outros vértices (complexidade $O(V^2)$ explicada anteriormente), obtém-se uma complexidade total de $O(n^3)$.

## *US304 – Para cada cliente determinar o hub mais próximo _ Tomás Russo*

O objetivo da US304 consiste em determinar, para cada cliente da rede de distribuição (seja este um cliente particular ou empresa), o *hub* de entrega mais próximo. Para executar esta US deverá primeiramente ser criado um objeto da classe *FindNearestHubController*, devendo ser passado por parâmetro um objeto da classe *DistributionNetwork*, que contém o grafo da rede de distribuição.

```
public FindNearestHubController(DistributionNetwork distributionNetwork) {
    this.distributionNetwork = distributionNetwork;
}
```

*Figura 13 - Construtor da classe FindNearestHubController*

Depois de ter o objeto criado, podemos invocar o método *findNearestHub*, que retornará um mapa que associa a cada cliente ou empresa (ambos objetos da superclasse abstrata *Entity*), o seu *hub* (objeto da classe *Enterprise*) mais próximo.

```
public Map<Entity, Enterprise> findNearestHub() {
    Map<Entity, Enterprise> result = new HashMap<>();
    ...
}
```

*Figura 14 - Declaração do mapa de retorno*

Para obter o *hub* mais próximo para cada cliente ou empresa, é necessário iterar por todas as entidades desses tipos presentes no grafo, determinando, para cada uma, o respetivo *hub* mais próximo, recorrendo ao algoritmo de *Dijkstra*.

```
public Map<Entity, Enterprise> findNearestHub() {
    ...
    List<Entity> clientsAndEnterprises =
      distributionNetwork.getclientsAndEnterprises();

    for (Entity entity : clientsAndEnterprises) {
      result.put(entity, distributionNetwork.getNearestHub(entity));
    }

    return result;
}
```

*Figura 15 - Preenchimento do mapa de retorno*

O método *getNearestHub*, presente na classe *DistributionNetwork*, é o responsável por retornar o *hub* mais próximo para o cliente/empresa passado por parâmetro.

```
public Enterprise getNearestHub(Entity entity) {
    ...

    return nearestHub;
}
```

*Figura 16 - Esqueleto da função getNearestHub*

Existem casos em que não se justifica a aplicação do algoritmo de *Dijkstra*, por ser imediata a obtenção do *hub* mais próximo: quando a entidade passada por parâmetro é ela mesmo um *hub* (é retornada a própria entidade); e quando existem exatamente 0 ou 1 *hubs* na rede – nestas situações é retornado nulo ou o único *hub* da rede, respetivamente.

```
public Enterprise getNearestHub(Entity entity) {
    if (entity.isHub())
      return (Enterprise) entity;

    List<Enterprise> hubs = this.getHubs();

    if (hubs.size() == 0)
      return null;

    if (hubs.size() == 1)
      return hubs.get(0);

    ...
}
```

*Figura 17 – Verificações iniciais para melhorar a eficiência do algoritmo*

Estas verificações indicam desde já que este algoritmo é não determinístico, sendo o melhor caso aquele que ocorre quando é passado por parâmetro um *hub*. Já o método *getHubs* possui complexidade O(V), na medida em que itera por todos os vértices (V) do grafo.

Se o número de *hubs* da rede for maior ou igual a dois, então é necessário utilizar o algoritmo de *Dijkstra* para obter os caminhos mais curtos entre todos os vértices do grafo.

```
public Enterprise getNearestHub(Entity entity) {
    ...

    ArrayList<Integer> distancesToOtherVertices = this.shortestPathsDistances(entity);
    ...
}
```

*Figura 18 - Obtenção das distâncias para todos os outros vértices do grafo*

O método *shortestPathsDistances* utiliza o algoritmo de *Dijkstra* para determinar as distâncias entre o vértice de origem (que neste caso é a entidade passada por parâmetro), e todos os outros vértices do grafo.

O algoritmo de *Dijkstra* tem complexidade O((V+E) * log V), na medida em que itera sobre todos os vértices do grafo, o que possui uma complexidade temporal O(V), e por sua vez itera sobre todas as arestas incidentes nesse vértice, que apresenta uma complexidade temporal O(E). Dado que a complexidade total do algoritmo é diretamente proporcional tanto ao número de vértices quanto ao número de arestas, podemos dizer que a complexidade temporal total do algoritmo de Dijkstra é O((V+E) * log V).

Depois de obtidas todas as distâncias, basta encontrar o *hub* cuja distância ao vértice representado pela entidade passada por parâmetro é a menor. Fazemos isso iterando pelas distâncias contidas no *ArrayList* obtido no passo anterior que representam a distância entre a entidade e um *hub*, sendo assim possível encontrar a menor. Podemos agora retornar o *hub* mais próximo da entidade.

```
public Enterprise getNearestHub(Entity entity) {
    ...

    Enterprise nearestHub = hubs.get(0);
    int minDistance = distancesToOtherVertices.get(network.key(hubs.get(0)));

    for (int i = 1; i < hubs.size(); i++) {
        Enterprise hub = hubs.get(i);
        int distance = distancesToOtherVertices.get(network.key(hub));

        if (distance < minDistance) {
            minDistance = distance;
            nearestHub = hub;
        }
    }

    return nearestHub;
}
```

*Figura 19 - Obtenção da menor distância entre a entidade e um hub*

A complexidade total do algoritmo desta US, no caso médio, que também corresponde ao pior caso, terá complexidade O(V * ((V+E) * log V)), sendo que é executado o algoritmo de *Dijkstra* para cada vértice do grafo. No melhor caso, o algoritmo terá complexidade O(V),

sendo esse aquele que ocorre quando o grafo apenas possuir empresas, e essas empresas forem todas *hubs*.

### *US305 – Determinar a rede que conecte todos os clientes e produtores agrícolas com uma distância total mínima _ André Barros*

Esta funcionalidade consiste em determinar a rede que conecte todos os clientes e produtores agrícolas com o caminho mais curto entre eles.

```
public ConnectedNetworkShortestPathController(DistributionNetwork network) {
   if (network == null)
     throw new IllegalArgumentException("Network is null");
   this.network = network;
 }

 /*
  * Get shortest path between all entities in a network
  */
 public Graph<Entity, Integer> getConnectedNetworkShortestPath() throws NetworkNotConnectedException {
   return network.getMinimumShortestPathNetwork();
```

*Figura 20 - Excerto do controlador*

É primeiro verificado se a network é conexa, pois caso não seja não é possível devolver a rede que conecta todos os clientes e produtores. Caso contrário, será utilizado o método de *Kruskall* que, a partir de um grafo, obtém a distância total mínima entre todos os vértices.

Na classe *DistributionNetwork* o método *getMiniminumShorthestPathNetwork*() implementa o *comparator* de *Edges<Entity,Integer>* para comparar pesos de cada ramo do grafo, retornando no fim a rede mínima entre todos os vértices.

```
/**
   * Gets the shortest path between all entities in the network
   *
   * @param ce comparator to sort the entities by the distance
   * @return graph of the shortest path from the distribution network
   */
  public AdjacencyMapGraph<Entity, Integer> getMinimumShortestPathNetwork() {
    final Comparator<Edge<Entity, Integer>> ce = new Comparator<Edge<Entity, Integer>>() {
      @Override
      public int compare(Edge<Entity, Integer> arg0, Edge<Entity, Integer> arg1) {
        return arg0.getWeight() - arg1.getWeight();
      }
    };

    return GraphAlgorithms.kruskall(network, ce);
  }
```

*Figura 21 - Construtor da DistributionNetwork*

Inicialmente é criada a árvore MST (Minimum Spanning Tree), em que depois são adicionados todos os vértices da *network*.

A seguir, é criada uma lista dos ramos da *network*. Depois é usado o *Merge Sort* que recebe o *comparator* dos *Edges* para ordena-los pelo peso por ordem crescente.

Depois, por todos os ramos vamos fazer uma *DepthFirstSearch*. Com isto, se na lista não estiver contido o vértice de destino, irá ser adicionado o ramo que contém o vértice de origem e de destino. No final, é retornado a árvore do resultado, MST.

Este método é não deterministico pois se a *network* tiver os ramos já ordenados pelo peso, a complexidade do método será de O(E). Caso contrário, se os ramos da *network* tiver estiverem ordenados, a complexidade será de O(E log(E)).

```java
/**
   * Calculates the minimum distance graph using Kruskal algorithm
   *
   * @param <V> vertex type
   * @param <E> edge type
   * @param g   initial graph
   * @param ce  comparator between elements of type E
   * @return the minimum distance graph
   */
  public static <V, E> AdjacencyMapGraph<V, E> kruskall(Graph<V, E> g, Comparator<Edge<V, E>> ce) {
    AdjacencyMapGraph<V, E> mst = new AdjacencyMapGraph<>(false);

    List<Edge<V, E>> edges = new ArrayList<>();

    for (V v : g.vertices())
      mst.addVertex(v);

    for (Edge<V, E> e : g.edges())
      edges.add(e);

    edges = new MergeSort<Edge<V, E>>().sort(edges, ce);

    for (Edge<V, E> e : edges) {
      V vOrig = e.getVOrig();
      V vDest = e.getVDest();
      List<V> connectedVerts = DepthFirstSearch(mst, vOrig);
      if (!connectedVerts.contains(vDest))
        mst.addEdge(vOrig, vDest, e.getWeight());
    }

    return mst;
  }
}
```

*Figura 22 - Algoritmo Kruskal*

# Sprint 2

## *US307 – Importar a lista de cabazes _ Ricardo Moreira*

Nesta US, é pretendido que seja importada uma lista de cabazes a partir dos dados contidos num ficheiro. Os cabazes correspondem às encomendas de produtos agrícolas feitas por clientes - particulares ou empresas - à rede de distribuição diariamente, e estes produtos são disponibilizados pelos produtores, colocando as respetivas quantidades que têm para vender na rede.

Com isto, é necessário ler o ficheiro de cabazes. Este ficheiro tem de estar no formato *.csv* e contém uma lista linha por linha da entidade que disponibiliza (caso seja produtor) ou encomenda (caso seja cliente/empresa) um cabaz de produtos, o dia e a respetiva quantidade de cada produto.

| Clientes-Produtores | Dia | Prod1 | Prod2 | Prod3 | Prod4 | Prod5 | Prod6 | Prod7 | Prod8 | Prod9 |
|---|---|---|---|---|---|---|---|---|---|---|
| C1 | 1 | 0 | 0 | 0 | 0 | 5 | 2 | 0 | 0 | 0 |
| C2 | 1 | 0 | 5.5 | 4.5 | 0 | 4 | 0 | 0 | 0 | 1 |
| C3 | 1 | 10 | 0 | 0 | 0 | 9 | 2.5 | 0 | 0 | 4.5 |
| E1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| E2 | 1 | 9 | 6 | 9 | 0 | 6 | 8.5 | 7.5 | 0 | 2.5 |
| E3 | 1 | 0 | 0 | 0 | 0 | 0 | 10 | 0 | 0 | 5 |
| P1 | 1 | 0 | 7.5 | 9 | 2 | 6 | 0 | 8.5 | 3 | 3.5 |
| P2 | 1 | 7.5 | 6.5 | 1.5 | 7 | 4 | 2.5 | 4.5 | 3.5 | 1 |
| P3 | 1 | 2.5 | 2 | 0 | 0 | 1.5 | 8 | 9 | 6 | 3.5 |

*Figura 23 - Exemplo de um ficheiro de uma lista de cabazes*

Os dados deste ficheiro são lidos para uma *List* de *Maps*, o que permite posteriormente mapear cada item da lista para um *Map* de Produtos – Quantidades, que representa cada quantidade de produto encomendado/produzido por dia, e cada entidade terá a sua *DailyData*, uma classe que abstrai a lista de cabazes por cada dia. A estrutura acaba por ficar a seguinte (*Entity* pode ser um cliente, produtor ou empresa):

*Entity* -> *DailyData* (dia) -> *Map* <Produto, Quantidade>

Caso a *Entity* não exista (ainda não foi carregada a partir de outra *User Story*), será apresentado um erro e o cabaz não será inserido.

```java
public static int toPlan(List<Map<String, String>> data, EntityStore entityStore) {
  int count = 0;

  for (Map<String, String> map : data) {
    String entityId = map.get(Field.CLIENTPROD.name);
    int day = Integer.parseInt(map.get(Field.DAY.name));

    Entity foundEntity = entityStore.getEntityById(entityId);
    ...

    Map<Product, Double> thisDayData = new HashMap<>();
    int i = 1;

    while (map.get(Field.PRODUCT.name + i) != null) {
      String productName = Field.PRODUCT.name + i;
      Double quantity = Double.parseDouble(map.get(Field.PRODUCT.name + i));

      i++;
      ...

      try {
        Product product = new Product(productName);
        thisDayData.put(product, quantity);
      } ...
    }
    ...

    foundEntity.getDailyData().addDayData(day, thisDayData);
    count++;
  }

  return count;
}
```

*Figura 24 – Excerto do método que mapeia uma lista para cabazes de entidades*

O tempo de complexidade desde método que mapeia uma lista vinda da leitura de um ficheiro para cabazes de entidades é $O(n^2)$, onde $n$ é o número de itens na lista. O número de operações a serem efetuadas é diretamente proporcional ao tamanho da lista e, para cada operação, é necessário chamar o método *getEntityById()*, o qual efetua uma pesquisa em tempo linear pelas entidades e retorna a pretendida assim que a encontrar, daí a complexidade total ser quadrática.

### *US308 – Gerar uma lista de expedição para um determinado dia que forneça os cabazes sem qualquer restrição quanto aos produtores* _ *Carlos Lopes*

O objetivo desta *User Story* é gerar uma lista de expedições para um determinado dia, sem quaisquer restrições na escolha dos produtores. Uma lista de expedições deve apresentar todos os cabazes expedidos num determinado dia. Cada cabaz deve apresentar a qual cliente esta associado, qual o *Hub* de entrega, produtos e quantidades encomendadas, e produtos e quantidades recebidas, especificando qual produtor entregou qual produto.

Para tal, antes de gera a lista de expedições, é necessário calcular qual é o stock válido para suprimir as encomendas dos clientes. Assim, é necessário para todos os dias anteriores percorrer todas as encomendas e fazer os cálculos necessário para obter uma cópia do stock dos produtores alterada com todas essas encomendas.

Para cada dia anterior ao dia da lista de expedição é necessário percorrer todos os clientes, por sua vez percorrer todos os produtos encomendados, para, por fim, percorrer todos os produtos, para achar qual o melhor para suprimir essa encomenda.

```
public Map<Producer, DailyData> getActualStock(Integer day) {
    List<Client> clientsList = this.network.getClients();
    Map<Producer, DailyData> prodStocks = this.network.getProducersStockUntilDate(day);

    for (int i = 1; i < day; i++) { // iterate all days before
      for (int j = 0; j < clientsList.size(); j++) { // iterate all clients
        Map<Product, Double> ordered = clientsList.get(j).getDayData(i);

        if (ordered == null)
          continue;

        for (Product product : ordered.keySet()) { // iterates client product orders
          Producer bestProducer = null;
          Double bestQuant = 0.;

          Double quantOrdered = ordered.get(product);

          for (Producer producer : prodStocks.keySet()) { // iterates all producers
            Double quantAvailable = prodStocks.get(producer).getNonExpiredProductQuantity(product, i);

            if (quantAvailable >= quantOrdered) {
              bestProducer = producer;
              bestQuant = quantOrdered;
              break;
            } else if (bestQuant < quantAvailable) {
              bestProducer = producer;
              bestQuant = quantAvailable;
            }
          }

          if (bestProducer != null)
            prodStocks.get(bestProducer).removeValidProductQuantity(product, bestQuant, i);
        }
      }
    }
    return prodStocks;
}
```

*Figura 25 - Método getActualStock*

Este método apresenta uma complexidade de O(n^4) devidos aos quatro ciclos referidos anteriormente.

Após ter o stock válido, é necessário repetir o processo apenas para o último dia, fazendo o registo na lista de expedições (complexidade de O(n^3)).

```
public ExpeditionList getExpeditionList(Integer day)
      throws InvalidOrderException, InvalidHubException, UndefinedHubsException {
    ExpeditionList expeditionList = new ExpeditionList(day);

    List<Client> clientsList = this.network.getClients();

    if (this.getNearestHub(clientsList.get(0)) == null)
      throw new UndefinedHubsException();
    Map<Producer, DailyData> prodStocks = this.getActualStock(day);

    for (int j = 0; j < clientsList.size(); j++) { // iterate all clients
      Client client = clientsList.get(j);
      Map<Product, Double> ordered = client.getDayData(day);
      ReceivedProducts received = new ReceivedProducts();

      Enterprise hub = this.getNearestHub(client);

      if (ordered == null)
        continue;

      for (Product product : ordered.keySet()) { // iterates client product orders
        Producer bestProducer = null;
        Double bestQuant = 0.;

        Double quantOrdered = ordered.get(product);

        for (Producer producer : prodStocks.keySet()) { // iterates all producers
          Double quantAvailable = prodStocks.get(producer).getNonExpiredProductQuantity(product, day);

          if (quantAvailable >= quantOrdered) {
            bestProducer = producer;
            bestQuant = quantOrdered;
            break;
          } else if (bestQuant < quantAvailable) {
            bestProducer = producer;
            bestQuant = quantAvailable;
          }

        }

        if (bestProducer == null)
          continue;

        // remove stock
        prodStocks.get(bestProducer).removeValidProductQuantity(product, bestQuant, day);

        // register for expeditionsList
        received.setProduct(bestProducer, product, bestQuant);

      }

      Basket basket = new Basket(ordered, received, hub, client);

      expeditionList.addBasket(basket);

    }

    return expeditionList;
  }
```

*Figura 26 - getExpeditionList method*

### *US309 – 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 _ André Barros*

O objetivo desta US passa por gerar uma lista de expedição para um determinado dia que forneça com N produtores agrícolas mais próximos do hub de entrega do cliente.

Uma lista de expedições deve apresentar todos os cabazes expedidos num determinado dia. Cada cabaz deve apresentar a qual cliente esta associado, qual o *Hub* de entrega, produtos e quantidades encomendadas, e produtos e quantidades recebidas, especificando qual produtor entregou qual produto.

Desta forma, iremos inicialmente buscar a lista de clientes pela network com a classe *Client*. Caso na *Network* não esteja nenhum *Hub* definido é lançada uma *Exception, UndefinedHubsException()*. Após esta verificação, iremos então buscar por o *Stock* de cada produtor dentro do alcance do N produtores mais próximos.

```
public ExpeditionList getExpeditionListForNNearestProducers(Integer day, Integer nProducers)
      throws InvalidOrderException, InvalidHubException, UndefinedHubsException {
    ExpeditionList expeditionList = new ExpeditionList(day);

    List<Client> clientsList = this.network.getEntitiesWithClass(Client.class);

    if (this.getNearestHub(clientsList.get(0)) == null)
      throw new UndefinedHubsException();
    Map<Enterprise, Map<Producer, DailyData>> prodStocks = this.getActualStockForNNearestProducers(day, nProducers);

    ( ... )
  }
```

*Figura 27 - Método getExpeditionListForNNearestProducers()*

O método *getActualStockForNNearestProducers()* inicialmente irá buscar  irá percorrer pelos dias *i-2, i-1 e i* e por cada cliente o que cada cliente encomendou, guardando essa informação num *Map<Product, Double>*. A seguir, por cada produto da encomenda, irá percorrer todos os produtores e verificar se estes conseguem suprimir a quantidade pedida. Caso consiga, o produtor irá ser escolhido para entregar produtos ao cliente em questão. Este método irá ter de complexidade (...).

```java
public Map<Enterprise, Map<Producer, DailyData>> getActualStockForNNearestProducers(Integer day, Integer
nProducers) {
    List<Client> clientsList = this.network.getEntitiesWithClass(Client.class);
    Map<Client, Enterprise> clientHub = new HashMap<>();

    for (int i = 0; i < clientsList.size(); i++) {
      clientHub.put(clientsList.get(i), this.getNearestHub(clientsList.get(i)));
    }

    Map<Enterprise, Map<Producer, DailyData>> prodStocks = this.getNNearestProducersStock(nProducers, day);

    for (int i = 1; i < day; i++) { // iterate all days before
      for (Client client : clientHub.keySet()) { // iterate all clients
        Map<Product, Double> ordered = client.getDayData(i);
        Enterprise hub = clientHub.get(client);

        if (ordered == null)
          continue;

        for (Product product : ordered.keySet()) { // iterates client product orders
          Producer bestProducer = null;
          Double bestQuant = 0.;

          Double quantOrdered = ordered.get(product);

          for (Producer producer : prodStocks.get(hub).keySet()) { // iterates all producers
            Double quantAvailable = prodStocks.get(hub).get(producer).getNonExpiredProductQuantity(product, i);

            if (quantAvailable ≥ quantOrdered) {
              bestProducer = producer;
              bestQuant = quantOrdered;
              break;
            } else if (bestQuant < quantAvailable) {
              bestProducer = producer;
              bestQuant = quantAvailable;
            }
          }

          if (bestProducer ≠ null)
            prodStocks.get(hub).get(bestProducer).removeValidProductQuantity(product, bestQuant, i);
        }
      }
    }
    return prodStocks;
  }
```

*Figura 28 - Método getActualStockForNNearestProducers()*

O método *getNNearestProducersStock()* vai buscar para os N produtores mais próximos do *Hub* do cliente e para o dia da expedição o stock de cada um deles. Desta forma, iremos ter de percorrer por todos os hubs, encontrar os N produtores mais próximos do *Hub* com o método *getNNearestProducersByHub()*. Depois para cada um dos produtores, iremos verificar se o stock dos produtores não se repete e colocar o stock que cada um tem até há data. Após isto é retornado um *Map* que faz a conexão para cada *Hub* os produtores próximos e o seu stock.

```java
public Map<Enterprise, Map<Producer, DailyData>> getNNearestProducersStock(Integer nProducers, Integer day) {
    List<Enterprise> hubs = this.getHubs();
    Map<Enterprise, Map<Producer, DailyData>> result = new HashMap<>();

    Map<Producer, DailyData> visitedStock = new HashMap<>();

    for (Enterprise hub : hubs) {
      List<Producer> producersList = this.getNNearestProducersByHub(hub, nProducers);

      Map<Producer, DailyData> stock = new LinkedHashMap<>();

      for (Producer producer : producersList) {
        DailyData dailyStock = visitedStock.get(producer);
        if (dailyStock == null) {
          dailyStock = producer.getStockUntilDate(day);
          visitedStock.put(producer, dailyStock);
        }

        stock.put(producer, dailyStock);
      }

      result.put(hub, stock);
    }

    return result;
  }
```

*Figura 29 - Método getNNearestProducersStock()*

O método *getNNearestProducersByHub()* vai buscar a network todos os produtores existentes. Com o algoritmo do *shorthestPath*, iremos encontrar o menor caminho do hub para todos os vértices do grafo. A seguir, com essas distâncias, irá encontrar os N produtores do mais perto ao mais longe retornando no fim a lista dos produtores. A complexidade é $O(n^2)$ pois o loop de fora é executado N vezes tal como o loop interior.

```java
public List<Producer> getNNearestProducersByHub(Enterprise hub, int n) {
    List<Producer> producers = network.getEntitiesWithClass(Producer.class);

    if (producers.size() < n)
      return null;

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

    ArrayList<Integer> distancesToOtherVertices = this.shortestPathsDistances(hub);

    for (int i = 0; i < n; i++) {
      Producer nearestProducer = producers.get(0);
      int minDistance = distancesToOtherVertices.get(network.key(producers.get(0)));

      for (int j = 1; j < producers.size(); j++) {
        Producer producer = producers.get(j);
        int distance = distancesToOtherVertices.get(network.key(producer));

        if (distance < minDistance) {
          minDistance = distance;
          nearestProducer = producer;
        }
      }

      result.add(nearestProducer);
      producers.remove(nearestProducer);
    }

    return result;
  }
```

*Figura 30 - Método getNNearestProducersByHub()*

Por fim, o método *getExpeditionListForNNearestProducers()* irá devolver a lista de expedição para um certo número de produtores N, e para um dia. A complexidade será de $O(n^3)$ pois teremos de percorrer a lista dos clientes, produtos e por fim produtores.

```
public ExpeditionList getExpeditionListForNNearestProducers(Integer day, Integer nProducers)
      throws InvalidOrderException, InvalidHubException, UndefinedHubsException {

    ( ... )

    for (int j = 0; j < clientsList.size(); j++) { // iterate all clients
      Client client = clientsList.get(j);
      Map<Product, Double> ordered = client.getDayData(day);
      ReceivedProducts received = new ReceivedProducts();

      Enterprise hub = this.getNearestHub(client);

      double sum = 0.;

      for (Double quant : ordered.values())
        sum += quant;

      if (ordered == null || sum == 0.)
        continue;

      for (Product product : ordered.keySet()) { // iterates client product orders
        Producer bestProducer = null;
        Double bestQuant = 0.;

        Double quantOrdered = ordered.get(product);

        for (Producer producer : prodStocks.get(hub).keySet()) { // iterates all producers
          Double quantAvailable = prodStocks.get(hub).get(producer).getNonExpiredProductQuantity(product, day);

          if (quantAvailable ≥ quantOrdered) {
            bestProducer = producer;
            bestQuant = quantOrdered;
            break;
          } else if (bestQuant < quantAvailable) {
            bestProducer = producer;
            bestQuant = quantAvailable;
          }

        }

        if (bestProducer == null)
          continue;

        // remove stock
        prodStocks.get(hub).get(bestProducer).removeValidProductQuantity(product, bestQuant, day);

        // register for expeditionsList
        received.setProduct(bestProducer, product, bestQuant);

      }

      Basket basket = new Basket(ordered, received, hub, client);

      expeditionList.addBasket(basket);

    }

    return expeditionList;
  }
```

*Figura 31 - Método getExpeditionListForNNearestProducers()*

## *US310 – Para uma lista de expedição diária gerar o percurso de entrega que minimiza a distância total percorrida* _ *Tomás Russo*

O objetivo desta US passa por determinar, dada uma lista de expedição diária, um percurso que minimize a distância total percorrida por todos os produtores e hubs dessa lista.

De maneira a encontrar o menor caminho possível entre todos os pontos da lista de expedição, seria necessário traçar todos os possíveis caminhos entre estes, encontrando depois o caminho com menor distância. Embora esta solução seja viável para um pequeno número de pontos, esta rapidamente torna se impraticável à medida que esse número cresce, na medida em que seria necessário computar $n!$ caminhos, sendo $n$ o número total de pontos. Considerando um cenário em que por cada segundo são processados 6 caminhos, e se existissem apenas 13 pontos na lista de expedição, esta solução demoraria aproximadamente 33 anos a ser executada.

| Number of Intermediate nodes | Number of Paths | 6 paths/sec. |
|---|---|---|
| A,B | X, A, B, Y<br>X, B, A, Y | |
| A,B,C | X,A,B,C,Y<br>X,A,C,B,Y<br>X,B,A,C,Y<br>X,B,C,A,Y<br>X,C,A,B,Y<br>X,C,B,A,Y | |
| A,B,C,D | 24 paths | 144 sec. |
| 13 intermediate nodes | **6 227 020 800 paths** | **33 years** |
| n intermediate nodes | n! paths | |

*Figura 32 - Tempo de execução da solução "bruteforce" (Fonte: apontamentos de ESINF, disponíveis no Moodle)*

Sendo esta solução inviável, é necessário implementar um algoritmo mais eficiente, mesmo que este não consiga obter o caminho ideal.

Posto isto, foi criado um algoritmo que, dado um ponto inicial, procura sucessivamente pelo seu vizinho mais próximo até que todos os pontos da lista tenham sido percorridos. Adaptado ao contexto desta US, o algoritmo escolhe arbitrariamente um produtor da lista, encontrando sucessivamente o produtor mais próximo, até que todos os produtores tenham sido percorridos. Depois, partindo do último produtor visitado, procura pelo hub mais próximo, e assim continuadamente até que todos os hubs tenham sido visitados.

Embora este algoritmo já se apresente eficiente, podemos acrescentar uma otimização importante: atualmente, o algoritmo primeiramente percorre todos os produtores, e só depois percorre os hubs, já que é obrigatório recolher toda a mercadoria antes de a mesma poder ser entregue. No entanto, se no caminho do produtor A para o produtor B se passar pelo hub A, e se o hub A apenas receber produtos do produtor A, a mercadoria já pode ser

entregue ao hub A, evitando assim uma segunda passagem desnecessária pelo mesmo na ronda pelos hubs.

Assim sendo, no sentido de implementar esta otimização, sempre que é obtido um caminho entre um produtor e o seu produtor mais próximo, e se esse caminho possuir algum hub, é verificado se já foram visitados todos os produtores que o(s) abastecem. Em caso afirmativo, a mercadoria é então entregue, sendo esse(s) hub(s) retirado(s) da lista dos hubs a visitar posteriormente.

Para obter o caminho de expedição, devemos criar um objeto da classe ExpeditionPath, passando por parâmetro a rede de distribuição (objeto da classe DistributionNetwork) e uma lista de expedição (objeto da classe ExpeditionList).

```java
public ExpeditionPath(DistributionNetwork distributionNetwork, ExpeditionList expeditionList) {
    this.distributionNetwork = distributionNetwork;
    this.expeditionList = expeditionList;
    this.path = new ArrayList<>();
    setDeliveredBaskets();
    setHubsSuppliers();
    buildPath();
}
```

*Figura 33 - Construtor da classe ExpeditionPath*

No construtor, após inicializar os atributos da classe, são evocados três métodos para calcular o caminho de expedição.

Em primeiro lugar, através do método setDeliveredBaskets, é inicializado um mapa, deliveredBaskets, que associa um hub (Enterprise) a um inteiro que representa o número de cabazes entregues nesse hub. O método getBasketsDeliveredAtHubs consulta a lista de expedição e percorre todos os cabazes, mapeando o hub de entrega de cada um.

```java
private Map<Enterprise, Integer> deliveredBaskets;

private void setDeliveredBaskets() {
    this.deliveredBaskets = this.expeditionList.getBasketsDeliveredAtHubs();
}
```

*Figura 34 - Inicialização do mapa deliveredBaskets*

Depois é invocado o método setHubsSuppliers, de maneira a ser inicializado um outro mapa, hubsSuppliers, que associa um hub (Enterprise) a uma lista de produtores (Producer). Essa lista representa todos os produtores que entregam produtos àquele hub.

```java
private Map<Enterprise, List<Producer>> hubsSuppliers;

private void setHubsSuppliers() {
    this.hubsSuppliers = this.expeditionList.getProducersThatSupplyHubs();
}
```

*Figura 35 - Inicialização do mapa hubsSuppliers*

Estes dois mapas são utilizados na otimização referida anteriormente (evitar uma segunda paragem desnecessária em hubs que já podem ser fornecidos).

Em último lugar é chamado o método buildPath, onde será finalmente calculado o caminho de expedição. Para tal, devemos, em primeiro lugar, obter todos os produtores que estão envolvidos na lista de expedição. Depois selecionamos um produtor aleatoriamente e calculamos, através do algoritmo A* (A Star), o caminho mais curto entre o produtor selecionado e todos os restantes.

Para fazer este cálculo, através do algoritmo A*, deve ser esboçada uma heurística que nos permite obter um valor aproximado da distância entre dois vértices. Nesta situação, dado que as entidades representam pontos geográficos, possuindo latitude e longitude, o valor da heurística ($h(v)$) é calculada com recurso à fórmula de Haversine, que dada as latitudes e longitudes de dois pontos (e o raio da Terra), calcula a distância aproximada entre eles.

$$d = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

*Figura 36 - Fórmula de Haversine*

O algoritmo A* utiliza essa heurística para estimar o custo do caminho mais curto entre dois pontos, sendo utilizada para guiar o algoritmo na direção do caminho mais promissor, o que ajuda a torná-lo mais eficiente comparativamente, por exemplo, ao algoritmo de Dijkstra. É por isso considerado um algoritmo de "pesquisa informada".

O algoritmo funciona da seguinte forma:

1. Inicialmente, coloca o ponto de partida na lista de vértices a explorar.
2. Enquanto houver vértices a explorar, o algoritmo seleciona o vértice com o menor custo total estimado, $f(v)$, isto é, o custo real até esse vértice, $g(v)$, somado do custo estimado até ao ponto de chegada, dado pelo valor da heurística, $h(v)$, e explora os seus vértices adjacentes. [$f(v) = g(v) + h(v)$, onde $v$ é o vértice selecionado][1].

[1] O algoritmo de Dijkstra funciona de forma semelhante, porém assume o valor de $h(v)$ como sendo 0 para todos os vértices.

3. Para cada um deles, o algoritmo atualiza o custo total estimado para chegar a esse vértice, tendo em conta o caminho mais curto até ao momento.
4. Se o algoritmo encontrar o ponto de chegada, este para e devolve o caminho encontrado. Caso contrário, repete os passos 2 e 3 até não haver mais vértices a explorar.

A complexidade temporal do A* depende da qualidade da heurística. Assumindo o pior caso, o algoritmo tem uma complexidade de $O(b^d)$, onde $b$ é o número médio de arestas a partir de cada vértice e $d$ é a profundidade do caminho mais curto. Isto significa que o tempo de execução aumenta exponencialmente com a profundidade do caminho mais curto. No entanto, como o algoritmo utiliza a heurística para o guiar na direção do caminho mais promissor, acaba por ser muito mais rápido que os seus concorrentes.

Assim sendo, para calcular o caminho mais curto entre um conjunto de produtores (dado um produtor inicial), podemos utilizar a heurística para estimar qual será o produtor mais próximo do inicial, e só depois calcular o caminho mais curto entre eles utilizando o A*.

```
private void buildPath() {
    List<Entity> entitiesPath = new ArrayList<>();

    List<Producer> producers = new ArrayList<>(this.expeditionList.getProducers());

    Entity start = producers.get(0);
    producers.remove(0);

    // Get the shortest path from the first producer to all other producers
    entitiesPath.addAll(distributionNetwork.getShortestPathUsingAStar(start, producers));

    ...
```

*Figura 37 - Obtenção do caminho mais curto entre todos os produtores no método buildPath*

Depois de obtido o caminho, implementamos a otimização já explicada anteriormente. Para tal, é utilizada uma classe interna (*inner class*) que representa as paragens do caminnho (Stop), indicando quantos cabazes foram entregues em cada uma delas.

```java
class Stop {
    public Entity entity;
    public Integer basketsDelivered;

    public Stop(Enterprise hub, int basketsDelivered) {
        this.entity = hub;
        this.basketsDelivered = basketsDelivered;
    }

    public Stop(Producer producer) {
        this.entity = producer;
        this.basketsDelivered = null;
    }

    public Stop(Client client) {
        this.entity = client;
        this.basketsDelivered = null;
    }
}
```

*Figura 38 - Classe interna Stop*

Depois, repete-se o mesmo processo para os hubs, onde se excluem os hubs já fornecidos. Finalizamos gerando as paragens para os hubs (generateStopsForHubs), onde se calcula os cabazes entregues em cada um, e definimos a distância total do caminho calculado.

```java
private void buildPath() {
    ...

    List<Enterprise> hubsSupplied = generateStopsForProducers(entitiesPath);

    Set<Enterprise> hubs = expeditionList.getHubs();
    // Remove the hubs that were already supplied
    hubs.removeAll(hubsSupplied);

    // Get the last stop in the path already calculated, and set it as the start
    start = entitiesPath.get(entitiesPath.size() - 1);
    entitiesPath.clear();

    entitiesPath.addAll(distributionNetwork.getShortestPathUsingAStar(start, hubs));

    generateStopsForHubs(entitiesPath, hubsSupplied);

    setTotalDistance();
}
```

*Figura 39 - Cálculo do caminho mais curto entre os hubs ainda não fornecidos*

Podemos finalmente apresentar o caminho mais curto, utilizando o método *printPath*.

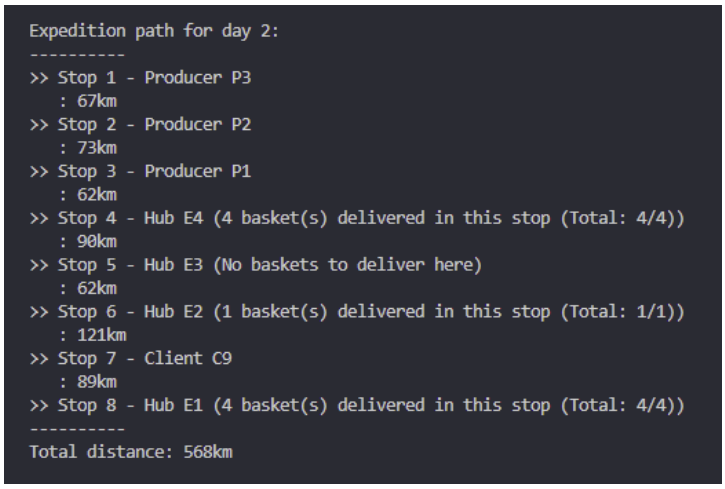


*Figura 40 - Impressão do caminho de expedição para o dia 2 (lista sem restrições), e para os ficheiros "small"*

O caminho calculado nesta US baseia-se no método do vizinho mais próximo (*nearest-neighbour)*, o que garante, na grande maioria das vezes, o caminho mais curto entre todos (ou então um caminho cujo comprimento está muito próximo do ideal). No entanto, a sua eficácia está dependente da escolha do produtor inicial (que é aleatória). Utilizemos um exemplo simples: consideramos um grafo cujos vértices são as capitais de distrito de Portugal e as arestas as distâncias médias entre elas. Se quisermos obter o caminho de expedição onde as paragens correspondem a produtores no Porto, Lisboa e Faro, o algoritmo apresentava um caminho ótimo caso o produtor inicial escolhido fosse Porto ou Faro. No entanto, se o ponto inicial fosse Lisboa, seriamos obrigados, por exemplo, a ir a Faro, voltar a Lisboa, e só depois seguir para Norte.

Posto isto, e no sentido de diminuir as probabilidades de ser escolhido um produtor "central", deveríamos escolher um produtor localizado num dos extremos geográficos do conjunto de produtores, para que o sentido de procura seja sempre o mesmo, evitando mudanças de sentido, e, consequentemente, possíveis visitas repetidas.

A complexidade total da funcionalidade é $O(n\ b^d)$, seja $n$ o número de entidades envolvidas na lista de expedição (produtores e hubs), e $b^d$ a complexidade do A*, na medida em que é calculado, para cada entidade o caminho mais curto para uma outra entidade, até que sejam todas percorridas.

### *US311 – Para uma lista de expedição calcular estatísticas* _ *Tomás Lopes*

O objetivo desta US é calcular diversas estatísticas para uma lista de expedição.

Depois de efetuados os cálculos que definem a lista de expedição, podemos obter algumas estatísticas, nomeadamente:

- por cabaz, determinar:
  - 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, determinar:
  - nº de cabazes totalmente satisfeitos
  - nº de cabazes parcialmente satisfeitos
  - nº de fornecedores distintos que forneceram todos os seus cabazes
- por produtor, determinar:
  - 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*, determinar:
  - nº de clientes distintos que recolhem cabazes em cada *hub*
  - nº de produtores distintos que fornecem cabazes para o *hub*

Para isso precisamos de métodos para determinar todas as entidades de um dado tipo que se encontrem presentes na lista de expedição.

```
public Set<Producer> getProducers() {
  Set<Producer> producers = new LinkedHashSet<>();

  for (Basket basket : this.baskets)
    for (Producer producer : basket.getProducers())
      producers.add(producer);

  return producers;
}
```

*Figura 41 - Excerto do método getProducers*

Este exemplo mostra uma amostra de código que determina todos os produtores que se encontram numa dada lista de expedição, todos os outros métodos, tanto para clientes e como para *hubs*. Neste código é usado um *Set* como estrutura adicional, já que é pretendida uma coleção sem duplicados. No caso, é usado um *LinkedHashSet* para facilitar, posteriormente, a testabilidade dos métodos que fazem uso do mesmo.

Todos estes métodos são de complexidade $O(b \times e)$, onde $b$ representa o número de cabazes (*baskets*) presentes na lista de expedição e $e$ o número de entidades (clientes, produtores ou *hubs*) que se pretende determinar.

Quanto às estatísticas para cabazes, temos:

```
public int getNumberOfFullySatisfiedProducts(Basket basket) {
   return basket.getNumberOfFullySatisfiedProducts();
 }

 public int getNumberOfPartiallySatisfiedProducts(Basket basket) {
   return basket.getNumberOfPartiallySatisfiedProducts();
 }

 public int getNumberOfNotSatisfiedProducts(Basket basket) {
   return basket.getNumberOfNotSatisfiedProducts();
 }

 public double getPercentageOfFullySatisfiedProducts(Basket basket) {
   return basket.getNumberOfFullySatisfiedProducts() / (double) basket.getNumberOfProducts();
 }

 public int getNumberOfDistinctProducersForBasket(Basket basket) {
   return basket.getProducers().size();
 }
```

*Figura 42 - Métodos usados para as estatísticas de cabazes*

A complexidade de todos estes métodos é $O(n)$, onde $n$ representa a quantidade de produtos num dado cabaz. A única exceção é o último método que se limita a retornar o *keySet* do mapa que representa os produtos entregues para o cabaz, que é de complexidade $O(1)$, pois o mapa está agrupado por produtores.

Quanto às estatísticas para clientes, temos:

```java
public int getNumberOfFullyFulfilledBasketsByClient(Client client) {
  int count = 0;

  for (Basket basket : baskets) {
    if (!basket.isFromClient(client))
      continue;

    if (basket.isFullyFulfilled())
      count++;
  }

  return count;
}

public int getNumberOfPartiallyFulfilledBasketsByClient(Client client) {
  int count = 0;

  for (Basket basket : baskets) {
    if (!basket.isFromClient(client))
      continue;

    if (basket.isPartiallyFulfilled())
      count++;
  }

  return count;
}

public int getNumberOfDistinctProducersThatSupplyAllClientsBasket(Client client) {
  Set<Producer> producers = new HashSet<>();

  for (Basket basket : baskets) {
    if (!basket.isFromClient(client))
      continue;

    Set<Producer> basketProducers = basket.getProducers();
    producers.addAll(basketProducers);
  }

  return producers.size();
}
```

*Figura 43 - Métodos usados para as estatísticas de clientes*

Os primeiros dois métodos apresentados na imagem acima são de complexidade $O(b \times p)$, pois os métodos *isFullyFulfilled()* e *isPartiallyFulFilled()* são de complexidade $O(p)$ pois é necessário iterar por todos os produtores do mapa de produtos entregues.

Já o método *getNumberOfDistinctProducersThatSupplyAllClientsBasket()* é de complexidade $O(b)$, pois o método *basket.getProducers()* é de complexidade *O(1)*.

Quanto às estatísticas para produtores, temos:

```java
public int getNumberOfFullySuppliedBasketsByProducer(Producer producer) {
    int count = 0;

    for (Basket basket : baskets)
      if (basket.isFullySuppliedBy(producer))
        count++;

    return count;
  }

  public int getNumberOfPartiallySuppliedBasketsByProducer(Producer producer) {
    int count = 0;

    for (Basket basket : baskets)
      if (basket.isPartiallySuppliedBy(producer))
        count++;

    return count;
  }

public int getNumberOfDistinctClients(Producer producer) {
    Set<Client> clients = new HashSet<>();

    for (Basket basket : baskets) {
      if (!basket.isFullySuppliedBy(producer) && !basket.isPartiallySuppliedBy(producer))
        continue;

      Client client = basket.getClient();
      clients.add(client);
    }

    return clients.size();
  }

  public int getNumberOfDistinctHubs(Producer producer) {
    Set<Enterprise> hubs = new HashSet<>();

    for (Basket basket : baskets) {
      if (!basket.isFullySuppliedBy(producer) && !basket.isPartiallySuppliedBy(producer))
        continue;

      Enterprise hub = basket.getHub();
      hubs.add(hub);
    }

    return hubs.size();
  }

public int getNumberOfOutOfStockProducts(Producer producer) {
    int count = 0;
    Map<Product, Double> totalSuppliedQuantity = new HashMap<>();

    for (Basket basket : baskets) {
      for (Product product : basket.getProducts()) {
        Double suppliedQuantity = basket.getQuantityOfSuppliedProduct(producer, product);

        Double currentSuppliedStock = totalSuppliedQuantity.get(product) == null ? 0
            : totalSuppliedQuantity.get(product);

        totalSuppliedQuantity.put(product, currentSuppliedStock + suppliedQuantity);
      }
    }

    for (Product product : totalSuppliedQuantity.keySet()) {
      Double availableStock = producer.getNonExpiredQuantityUntilDate(product, day);
      Double suppliedQuantity = totalSuppliedQuantity.get(product);

      if (availableStock.compareTo(suppliedQuantity) <= 0)
        count++;
    }

    return count;
  }
```

*Figura 44 - Métodos usados para as estatísticas de produtores*

Todos os métodos apresentados são de complexidade $O(b \times p)$, onde $b$ representa o número de cabazes e $p$ o número de produtores, devido à estrutura do mapa usado internamente pela classe *DeliveredProducts*.

Quanto às estatísticas para *hubs*, temos:

```java
public int getNumberOfDistinctClients(Enterprise hub) {
  Set<Client> clients = new HashSet<>();

  for (Basket basket : baskets) {
    if (!basket.isFromHub(hub))
      continue;

    Client client = basket.getClient();
    clients.add(client);
  }

  return clients.size();
}

public int getNumberOfDistinctProducers(Enterprise hub) {
  Set<Producer> producers = new HashSet<>();

  for (Basket basket : baskets) {
    if (!basket.isFromHub(hub))
      continue;

    Set<Producer> basketProducers = basket.getProducers();
    producers.addAll(basketProducers);
  }

  return producers.size();
}
```

*Figura 45 - Métodos usados para as estatísticas de hubs*

O método *getNumberOfDistinctClients* é de complexidade $O(b)$ em que $b$ representa o número de cabazes, pois a operação de inserir num *HashSet* é $\approx O(1)$.[2]

Já o método *getNumberOfDistinctProducers*, apesar de não tão evidente também é de complexidade $O(b)$. O método *basket.getProducers()* usa o método *keySet()* do mapa usado internamente pela estrutura. Este método é de complexidade $O(1)$, logo a complexidade total do método é também $O(b)$.

[2] Considerando que é usada uma boa função de *hash*

# Possíveis melhorias

Existem algumas questões que podem ser melhoradas de um ponto de vista técnico na aplicação desenvolvida.

Alguns dos algoritmos tendem a sofrer, a nível de performance, com o escalar do tamanho do *input* dado.

Como referido nesta secção de melhorias no Sprint passado, faria sentido recorrer ao uso de heurísticas, o que foi feito. As heurísticas usam aproximações que melhoram a performance dos algoritmos em deterioramento de um resultado completamente correto. Em alguns casos neste projeto faz sentido usar estas aproximações como, por exemplo, no cálculo de vértices mais próximos caso o grafo tivesse dimensões elevadas.