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:

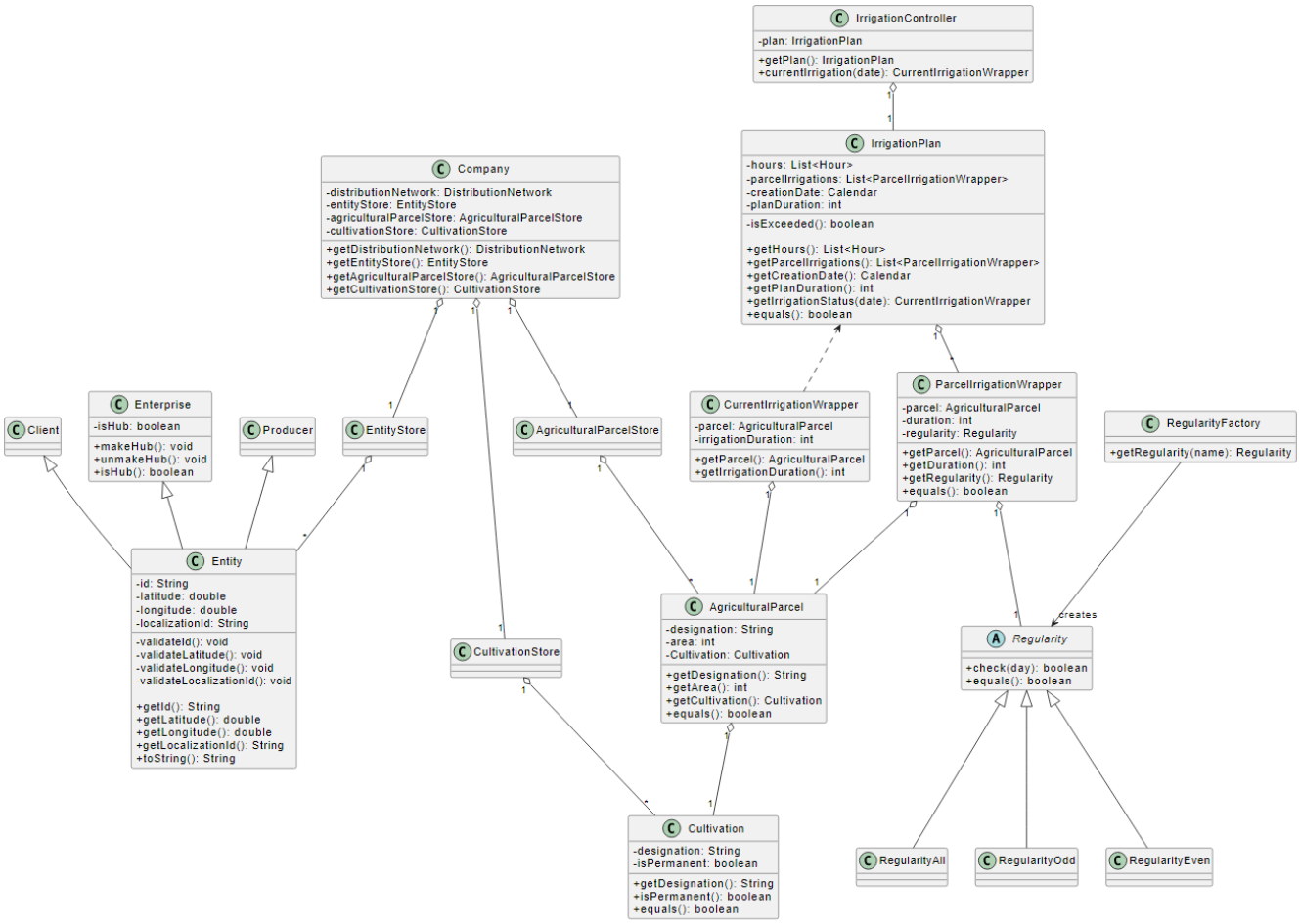

Figura 1 _ Digrama de classes

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 $O(V^2)$.

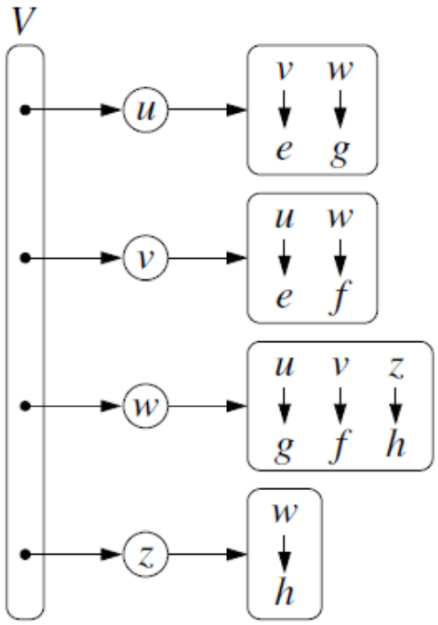

Figura 2 _ Representação gráfica de um mapa de adjacências

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

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.

```java
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.

```java
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();
```

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);
  }
```

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)).

```
/**
   * 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;
  }
}
```

Possíveis melhorias

Para além das melhorias futuras que já estão disponíveis e vão ser implementadas no próximo *sprint*, 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. Em algumas das funcionalidades faria sentido, dentro de algumas regras, recorrer ao uso de heurísticas. As heurísticas usam aproximações que melhoram a performance dos algoritmos em deterioramento de um resultado completamente correto. Em alguns casos neste projeto poderia fazer sentido usar estas aproximações, por exemplo, no cálculo de vértices mais próximos caso o grafo tivesse dimensões elevadas.