US309 - Análise de Complexidade

```
public void createDispatchList(int day) {
   Firm hub;
   Producer producer;
   //iterate clients basket order
   for (Map.Entry<Client, Cabaz> order : cabazes.getCabazC().get(day).entrySet()) {
      //get nearest HUB
      ArrayList<LinkedList<User>> paths = new ArrayList<>();
      ArrayList<Integer> dists = new ArrayList<>();
      Algorithms.shortestPaths(distributionNetwork.getGraph(), order.getKey(), Integer::compare,
Integer::sum, 0, paths, dists);
      hub = distributionNetwork.findNearestHUB(paths, dists);
      //iterate through basket products
      for (Map.Entry<Product, Float> orderProduct : order.getValue().getProducts().entrySet()) {
         producer = findProducerAvailableProduct(day, orderProduct);
         Delivery delivery = new Delivery(orderProduct.getKey(), orderProduct.getValue(), producer,
hub);
         addToMap(day, order.getKey(), delivery);
         removeProducts(producer, orderProduct, day);
      }
   }
}
```

Complexidade: O(CM(ElogV)P)

C – Número de clientes

M – Número máximo de produtos pedidos entre os clientes

E – Arestas do grafo

V – Vértices do grafo

P – Número de produtores passado por parâmetro (limite N mais próximos)

O algoritmo itera todos os clientes que fizeram uma encomenda num determinado dia. Para cada cliente, é calculado o HUB mais próximo, e os N produtores mais próximos (caso ainda não tenham sido calculados) (shortestPaths O(ElogV)). Depois, são iterados todos os produtos pedidos desse cliente é calculado o produtor que melhor satisfaz o pedido (findNearestProducerAvailableProduct). A encomenda é adicionada e os produtos removidos.

```
public Producer findNearestProducerAvailableProduct(Firm hub, int day, Map.Entry<Product, Float>
product, HashMap<Firm, TreeSet<DistributionNetwork.DistancePathPair>> closestProducersToHUB)
{
   float tempQuantity;
   float highestQuantity = 0;
   Producer bestProducer = null;
   Producer p;
   for (DistributionNetwork.DistancePathPair distancePathPair : closestProducersToHUB.get(hub))
{//iterate producers by closest to HUB
      tempQuantity = 0;                    //until limit or no more producers
      p = (Producer) distancePathPair.getPath().getLast();
      for (int i = 0; i <= 2 && day - i >= 1; i++) {//iterate available products
         if (cabazes.getCabazP().get(day - i).containsKey(p) &&
               cabazes.getCabazP().get(day - i).get(p).getProducts().containsKey(product.getKey())) {
            tempQuantity += cabazes.getCabazP().get(day -
i).get(p).getProducts().get(product.getKey());
            if (tempQuantity >= product.getValue()) {//found the closest producer with quantity
               return p;
            }
            if (tempQuantity > highestQuantity) {//check for producer with higher quantity
               highestQuantity = tempQuantity;
               bestProducer = p;
            }
         }
      }
   }
   return bestProducer;
}
```

Complexidade: O(P)

P – Número de produtores passado por parâmetro (limite N mais próximos)

No primeiro ciclo *for,* são iterados todos os N produtores mais próximos do HUB. Para cada iteração, é calculado a quantidade total do produto disponível. Caso a quantidade disponível satisfaça a quantidade pedida, é devolvido esse produtor. Se não for encontrado nenhum produtor, e devolvido aquele que melhor satisfaça o pedido.

```
public Firm findNearestHUB(ArrayList<LinkedList<User>> paths, ArrayList<Integer> dists) {
   TreeSet<DistancePathPair> set = new TreeSet<>();
   for (int i = 0; i < dists.size(); i++)
      if (paths.get(i).getLast() instanceof Firm && ((Firm) paths.get(i).getLast()).isHUB())
         set.add(new DistancePathPair(dists.get(i), paths.get(i)));

   return (Firm) set.first().getPath().getLast();
}
```

Complexidade: O(V Log n)

O algoritmo percorre todos os caminhos e distâncias passadas por parâmetro (obtidos usando shortestPaths que tem complexidade O(ElogV)), colocando num TreeSet, caso o destino seja um HUB, ordenando os caminhos pela distancia. Retorna o HUB com caminho menor

```
public TreeSet<DistancePathPair> findNearestNeighboursByType(int limit,
ArrayList<LinkedList<User>> paths, ArrayList<Integer> dists, Class<?> clazz) {
   TreeSet<DistancePathPair> set = new TreeSet<>();
   for (int i = 0; i < dists.size(); i++)
      if (paths.get(i).getLast().getClass().equals(clazz))
         set.add(new DistancePathPair(dists.get(i), paths.get(i)));

   return set.stream().limit(limit).collect(TreeSet::new, (m, e) -> m.add(new
DistancePathPair(e.getDistance(), e.getPath())), Set::addAll);
}
```

Complexidade: O(V Log n)

O algoritmo percorre todos os caminhos e distâncias passadas por parâmetro (obtidos usando shortestPaths que tem complexidade O(ElogV)), colocando num TreeSet, caso o destino seja do tipo passado por parâmetro, ordenando os caminhos pela distancia. Retorna o vértice do tipo passado por parâmetro com caminho menor

US308 - Análise de Complexidade

```
public Producer findProducerAvailableProduct(int day, Map.Entry<Product, Float> product) {
   float tempQuantity;
   float highestQuantity = 0;
   Producer bestProducer = null;
   for (User user : distributionNetwork.getGraph().vertices()) {//iterate producers by closest to HUB
until limit or no more producers
      if (user instanceof Producer) {
         tempQuantity = 0;
         for (int i = 0; i <= 2 && day - i >= 1; i++) {//iterate available products
            if (cabazes.getCabazP().get(day - i).containsKey(user) &&
                  cabazes.getCabazP().get(day - i).get(user).getProducts().containsKey(product.getKey()))
{
               tempQuantity += cabazes.getCabazP().get(day -
i).get(user).getProducts().get(product.getKey());
               if (tempQuantity >= product.getValue()) {//found the closest producer with quantity
                  return (Producer) user;
               }
               if (tempQuantity > highestQuantity) {//check for producer with higher quantity
                  highestQuantity = tempQuantity;
                  bestProducer = (Producer) user;
               }
            }
         }
      }
   }
   return bestProducer;
}
```

Complexidade: O(CM(ElogV)P)

C – número de clientes

M – numero máximo de produtos pedidos entre os clientes

E – arestas do grafo

V – vértices do grafo

P – número total de produtores

O algoritmo e semelhante ao findNearestProducerAvailableProduct, mas não são calculados os produtores mais próximos do HUB.

```
public Producer findProducerAvailableProduct(int day, Map.Entry<Product, Float> product) {
   float tempQuantity;
   float highestQuantity = 0;
   Producer bestProducer = null;
   for (User user : distributionNetwork.getGraph().vertices()) {//iterate producers by closest to HUB
until limit or no more producers
      if (user instanceof Producer) {
         tempQuantity = 0;
         for (int i = 0; i <= 2 && day - i >= 1; i++) {//iterate available products
            if (cabazes.getCabazP().get(day - i).containsKey(user) &&
                  cabazes.getCabazP().get(day - i).get(user).getProducts().containsKey(product.getKey()))
{
               tempQuantity += cabazes.getCabazP().get(day -
i).get(user).getProducts().get(product.getKey());
               if (tempQuantity >= product.getValue()) {//found the closest producer with quantity
                  return (Producer) user;
               }
               if (tempQuantity > highestQuantity) {//check for producer with higher quantity
                  highestQuantity = tempQuantity;
                  bestProducer = (Producer) user;
               }
            }
         }
      }
   }
   return bestProducer;
}
```

Complexidade: O(P)

P – número total de produtores

O algoritmo e semelhante ao findNearestProducerAvailableProduct, mas procura todos os produtores.