# Relatório ESINF Sprint 1 Projeto 2022/2023

# Análise de Complexidade


## Trabalho realizado por:

**Beatriz Neves - 1211512**

**Clarisse Sousa – 1211434**

**Cláudio Coelho – 1211435**

**Filipe Duarte – 1210959**

**João Castro – 1210816**

**Martim Botelho - 1211523**

No nosso projeto implementámos a classe *Algorithms* que contém a maior parte dos métodos necessários para a realização deste projeto e, portanto, começaremos por analisar as suas complexidades.

## Algoritmo de Kruskall:

```
public static <V,E> Graph<V,E> kruskall(Graph<V,E> g){
    Graph<V,E> mst = new MapGraph<>( directed: false);
    for (V vertex : g.vertices()){
        mst.addVertex(vertex);
    }
    ArrayList<Edge<V, E>> lstEdges = new ArrayList<>(g.edges());
    Collections.sort(lstEdges);
    for (Edge<V,E> edge : lstEdges){
        LinkedList<V> connectedVerts = DepthFirstSearch(mst, edge.getVOrig());
        if (!connectedVerts.contains(edge.getVDest())){
            mst.addEdge(edge.getVOrig(), edge.getVDest(), edge.getWeight());
        }
    }
    return mst;
}
```

Este método tem complexidade no pior caso de $O(m*\log_n)$(m = número de arestas ; n = número de vértices).

## Algoritmo Depth First Search:

```
private static <V, E> void DepthFirstSearch(Graph<V, E> g, V vOrig, boolean[] visited, LinkedList<V> qdfs) {

    if (!g.validVertex(vOrig) || visited[g.key(vOrig)]){
        return;
    }

    qdfs.add(vOrig);
    visited[g.key(vOrig)] = true;

   for (V vAdj : g.adjVertices(vOrig)) {
       DepthFirstSearch(g, vAdj, visited, qdfs);
   }
}
```

Este método tem complexidade no pior caso de $O(n^2)$.

```java
public static <V, E> LinkedList<V> DepthFirstSearch(Graph<V, E> g, V vert) {
    LinkedList<V> qdfs = new LinkedList<>();
    if (g != null && g.numVertices() > 0) {
        boolean[] visited = new boolean[g.numVertices()];
        DepthFirstSearch(g, vert, visited, qdfs);
        return qdfs;
    }
    return null;
}
```

Este método tem complexidade no pior caso de $O(n^2)$.

## Algoritmo all paths:

```java
private static <V, E> void allPaths(Graph<V, E> g, V vOrig, V vDest, boolean[] visited,
                                    LinkedList<V> path, ArrayList<LinkedList<V>> paths) {

    path.add(vOrig);
    visited[g.key(vOrig)] = true;
    for (V vAdj : g.adjVertices(vOrig)){
        if (vAdj == vDest) {
            path.add(vDest);
            paths.add(path);
            path.remove( index: path.size() - 1);
        }else {
            if (!visited[g.key(vAdj)]){
                allPaths(g, vAdj, vDest, visited, path, paths);
            }
        }
    }
    path.remove( index: path.size() - 1);

}
```

Este método tem complexidade no pior caso de $O(n^2)$.

## Algoritmo shortest path Dijkstra Weighted:

```
private static <V, E> void shortestPathDijkstraWeighted(Graph<V, E> g, V vOrig, boolean[] visited, int [] pathKeys, double [] dist) {

    Edge<V, E> edge;
    for (V vertex : g.vertices()) {

        dist[g.key(vertex)] = Double.MAX_VALUE;
        pathKeys[g.key(vertex)] = -1;
        visited[g.key(vertex)] = false;
    }

    dist[g.key(vOrig)] = 0;
    while (g.key(vOrig) != -1) {
        visited[g.key(vOrig)] = true;
        for (V vAdj : g.adjVertices(vOrig)) {
            edge = g.edge(vOrig, vAdj);
            if (!visited[g.key(vAdj)] && dist[g.key(vAdj)] > (dist[g.key(vOrig)] + (Double.parseDouble(edge.getWeight().toString())))) {
                dist[g.key(vAdj)] = (dist[g.key(vOrig)] + (Double.parseDouble(edge.getWeight().toString())));
                pathKeys[g.key(vAdj)] = g.key(vOrig);
            }
        }
        vOrig = g.vertex(getVertMinDistance(dist, visited));
    }

}
```

Este método tem complexidade no pior caso de $O(n^2)$.

## Algoritmo shortest path Dijkstra Unweighted:

```
private static <V, E> void shortestPathDijkstraUnweighted(Graph<V, E> g, V vOrig, int [] pathKeys, int [] dist) {

    ArrayList<V> queue_aux = new ArrayList<>();
    for (V vertex : g.vertices()) {

        dist[g.key(vertex)] = Integer.MAX_VALUE;
        pathKeys[g.key(vertex)] = -1;
    }
    queue_aux.add(vOrig);
    dist[g.key(vOrig)] = 0;
    while (!queue_aux.isEmpty()) {
        vOrig = queue_aux.remove( index: 0);
        for (V vAdj : g.adjVertices(vOrig)) {
            if (dist[g.key(vAdj)] == Integer.MAX_VALUE) {
                dist[g.key(vAdj)] = dist[g.key(vOrig)] + 1;
                pathKeys[g.key(vAdj)] = g.key(vOrig);
                queue_aux.add(vAdj);
            }
        }
    }
}
```

Este método tem complexidade no pior caso de $O(n^2)$.

Algoritmo para obter a distância mínima entre vértices:

```
private static int getVertMinDistance(double[]dist, boolean[]visited){
    double min = Double.POSITIVE_INFINITY;
    int v, min_index = -1;
    for(v = 0; v < visited.length; v++){
        if(!visited[v] && dist[v] < min){
            min = dist[v];
            min_index = v;
        }
    }
    return min_index;
}
```

Este método tem complexidade no pior caso de O(n).

Algoritmo shortest path weighted:

```
public static <V, E> double shortestPathWeighted(Graph<V, E> g, V vOrig, V vDest, LinkedList<V> shortPath) {

    shortPath.clear();
    if (!g.validVertex(vOrig) || !g.validVertex(vDest)) {
        return 0;
    }

    if (vOrig.equals(vDest)) {
        shortPath.add(vDest);
        return 0;
    }

    int numVertices = g.numVertices();
    boolean[] visited = new boolean[numVertices]; //default value: false
    int[] pathKeys = new int[numVertices];
    double[] dist = new double[numVertices];

    for (int i = 0; i < numVertices; i++) {
        dist[i] = Double.MAX_VALUE;
        pathKeys[i] = -1;
    }
    shortestPathDijkstraWeighted(g, vOrig, visited, pathKeys, dist);
    if (pathKeys[g.key(vDest)] == -1) {
        return 0;
    }
    getPath(g, vOrig, vDest, pathKeys, shortPath);
    return dist[g.key(vDest)];
}
```

Este método tem complexidade no pior caso de $O(n^2)$.

Algoritmo shortest path weighted boolean:

```java
public static <V, E> boolean shortestPathsWeighted(Graph<V, E> g, V vOrig, ArrayList<LinkedList<V>> paths, ArrayList<Double> dists) {

    paths.clear();
    dists.clear();

    if (!g.validVertex(vOrig)) {
        return false;
    }

    int numVertices = g.numVertices();
    boolean[] visited = new boolean[numVertices]; //default value: false
    int[] pathKeys = new int[numVertices];
    double[] dist = new double[numVertices];

    for (int i = 0; i < numVertices; i++) {
        dist[i] = Double.MAX_VALUE;
        pathKeys[i] = -1;
    }

    shortestPathDijkstraWeighted(g,vOrig, visited, pathKeys, dist);

    for (int i = 0; i < numVertices; i++) {
        paths.add(null);
        dists.add(null);
    }
    for (int i = 0; i < numVertices; i++) {
        LinkedList<V> shortPath = new LinkedList<>();

        if (Double.compare(dist[i], Double.MAX_VALUE) != 0) {
            getPath(g, vOrig, g.vertices().get(i), pathKeys, shortPath);
        }
        paths.set(i, shortPath);
        dists.set(i, dist[i]);
    }
    return true;

}
```

Este método tem complexidade no pior caso de $O(n^2)$.

## Algoritmo shortest path unweighted boolean:

```java
public static <V, E> boolean shortestPathsUnweighted(Graph<V, E> g, V vOrig, ArrayList<LinkedList<V>> paths, ArrayList<Integer> dists) {

    paths.clear();
    dists.clear();

    if (!g.validVertex(vOrig)) {
        return false;
    }

    int numVertices = g.numVertices();
    int[] pathKeys = new int[numVertices];
    int[] dist = new int[numVertices];

    for (int i = 0; i < numVertices; i++) {
        dist[i] = Integer.MAX_VALUE;
        pathKeys[i] = -1;
    }

    shortestPathDijkstraUnweighted(g,vOrig, pathKeys, dist);

    for (int i = 0; i < numVertices; i++) {
        paths.add(null);
        dists.add(null);
    }
    for (int i = 0; i < numVertices; i++) {
        LinkedList<V> shortPath = new LinkedList<>();
        if (dist[i] != Integer.MAX_VALUE) {
            getPath(g, vOrig, g.vertices().get(i), pathKeys, shortPath);
        }
        paths.set(i, shortPath);
        dists.set(i, dist[i]);
    }
    return true;

}
```

Este método tem complexidade no pior caso de $O(n^2)$.

## Algoritmo para obter um Path:

```java
private static <V, E> void getPath(Graph<V, E> g, V vOrig, V vDest,
                                   int [] pathKeys, LinkedList<V> path) {

    path.push(vDest);
    int vKey = pathKeys[g.key(vDest)];
    if (vKey != -1){
        vDest = g.vertices().get(vKey);
        getPath(g, vOrig, vDest, pathKeys, path);
    }
}
```

Este método tem complexidade no pior caso de O(n).

Para além da classe *Algorithms* também foi implementada a classe *MatrixGraph* que contém outra parte dos métodos necessários para a realização deste projeto e, portanto, segue a análise de complexidade de alguns dos seus métodos.

## Algoritmo para obter os vértices adjacentes:

```
public Collection<V> adjVertices(V vert) {
    int index = key(vert);
    if (index == -1)
        return null;

    ArrayList<V> outVertices = new ArrayList<>();
    for (int i = 0; i < numVerts; i++)
        if (edgeMatrix[index][i] != null)
            outVertices.add(vertices.get(i));
    return outVertices;
}
```

Este método tem complexidade no pior caso de O(n).

## Algoritmo para obter os edges:

```
public Collection<Edge<V, E>> edges() {

    Collection<Edge<V,E>> edges = new LinkedList<>();
    for (Edge<V, E>[] edgeA : edgeMatrix){
        for (Edge<V,E> edge : edgeA)
            if (edge != null) {
                edges.add(edge);
            }
    }
    return edges;
}
```

Este método tem complexidade no pior caso de $O(n^2)$.

Algoritmo para obter outgoing edges:

```
public Collection<Edge<V, E>> outgoingEdges(V vert) {

    Collection<Edge<V,E>> edgeCollection = new ArrayList<>();

    for (Edge<V, E>[] edgeA : edgeMatrix){
        for (Edge<V,E> edge : edgeA) {
            if (edge != null && vert == edge.getVOrig()) {
                edgeCollection.add(edge);
            }
        }
    }
    return edgeCollection;

}
```

Este método tem complexidade no pior caso de $O(n^2)$.

Algoritmo para obter incoming edges:

```
public Collection<Edge<V, E>> incomingEdges(V vert) {
    Collection <Edge<V, E>> ce = new ArrayList<>();
    int vertKey = key(vert);
    if (vertKey == -1)
        return ce;

    for (int i = 0; i < numVerts; i++)
        if (edgeMatrix[i][vertKey] != null)
            ce.add(edgeMatrix[i][vertKey]);
    return ce;
}
```

Este método tem complexidade no pior caso de O(n).

Algoritmo para adicionar vertex:

```java
public boolean addVertex(V vert) {
    int vertKey = key(vert);
    if (vertKey != -1)
        return false;

    vertices.add(vert);
    numVerts++;
    resizeMatrix();
    return true;
}
```

Este método tem complexidade no pior caso de O(n).

Algoritmo para adicionar edge:

```java
public boolean addEdge(V vOrig, V vDest, E weight) {
    if (vOrig == null || vDest == null) throw new RuntimeException("Vertices cannot be null!");
    if (edge(vOrig, vDest) != null)
        return false;

    if (!validVertex(vOrig))
        addVertex(vOrig);

    if (!validVertex(vDest))
        addVertex(vDest);

    int vOrigKey = key(vOrig);
    int vDestKey = key(vDest);

    edgeMatrix[vOrigKey][vDestKey] = new Edge<>(vOrig, vDest, weight );
    numEdges++;
    if (!isDirected) {
        edgeMatrix[vDestKey][vOrigKey] = new Edge<>(vDest, vOrig, weight );
        numEdges++;
    }
    return true;
}
```

Este método tem complexidade no pior caso de O(n).

Para além da classe *Algorithms* e *MatrixGraph*, também foi implementada a classe *MapGraph* que contém outra das partes dos métodos necessários para a realização deste projeto e, portanto, segue a análise de complexidade de alguns dos seus métodos.

## Método que verifica se um vértice é válido ou não:

```
public boolean validVertex(V vert) { return (mapVertices.get(vert) != null);}
```

Este método tem complexidade no pior caso de O(n).

## Método que retorna os vértices adjacentes:

```
public Collection<V> adjVertices(V vert) { return mapVertices.get(vert).getAllAdjVerts(); }
```

Este método tem complexidade no pior caso de O(n).

## Método edges:

```
public Collection<Edge<V, E>> edges() {

    ArrayList<Edge<V, E>> le = new ArrayList<>(numEdges);

    for (MapVertex<V, E> mv : mapVertices.values())
        le.addAll(mv.getAllOutEdges());

    return le;
}
```

Este método tem complexidade no pior caso de O(n).

## Método edge:

```
public Edge<V, E> edge(V vOrig, V vDest) {

    if (!validVertex(vOrig) || !validVertex(vDest))
        return null;

    MapVertex<V, E> mv = mapVertices.get(vOrig);

    return mv.getEdge(vDest);
}
```

Este método tem complexidade no pior caso de O(n).

## Método add Vertex:

```
public boolean addVertex(V vert) {

    if (vert == null) throw new RuntimeException("Vertices cannot be null!");
    if (validVertex(vert))
        return false;

    MapVertex<V, E> mv = new MapVertex<>(vert);
    vertices.add(vert);
    mapVertices.put(vert, mv);
    numVerts++;

    return true;
}
```

Este método tem complexidade no pior caso de O(1).

Método add edge:

```java
public boolean addEdge(V vOrig, V vDest, E weight) {

    if (vOrig == null || vDest == null) throw new RuntimeException("Vertices cannot be null!");
    if (edge(vOrig, vDest) != null)
        return false;

    if (!validVertex(vOrig))
        addVertex(vOrig);

    if (!validVertex(vDest))
        addVertex(vDest);

    MapVertex<V, E> mvo = mapVertices.get(vOrig);
    MapVertex<V, E> mvd = mapVertices.get(vDest);

    Edge<V, E> newEdge = new Edge<>(mvo.getElement(), mvd.getElement(), weight);
    mvo.addAdjVert(mvd.getElement(), newEdge);
    numEdges++;

    //if graph is not direct insert other edge in the opposite direction
    if (!isDirected)
        // if vDest different vOrig
        if (edge(vDest, vOrig) == null) {
            Edge<V, E> otherEdge = new Edge<>( mvd.getElement(), mvo.getElement(), weight);
            mvd.addAdjVert(mvo.getElement(), otherEdge);
            numEdges++;
        }

    return true;
}
```

Este método tem complexidade no pior caso de O(n).

Por fim, será analisada a complexidade dos métodos criados para resolver o proposto pelas US's.

## US302

```
private static boolean verifyConnectivity() {

    Localization user = storeGraph.vertices().get(0);
    LinkedList<Localization> reachableVertex = Algorithms.DepthFirstSearch(storeGraph, user);
    return storeGraph.vertices().size() == reachableVertex.size();
}
```

Este método tem complexidade no pior caso de $O(n^2)$.

```
public static ArrayList<ClientProducerDist> shortestPathsUnweighted() {
    if (verifyConnectivity()) {
        ArrayList<LinkedList<Localization>> paths = new ArrayList<>();
        ArrayList<Integer> dists = new ArrayList<>();
        ArrayList<ClientProducerDist> clientProducerDists = new ArrayList<>();

        for (Localization client : clientsList) {
            if (Algorithms.shortestPathsUnweighted(storeGraph, client, paths, dists)) {
                for (int producer = 0; producer < paths.size(); producer++) {
                    if ((paths.get(producer).getLast().getUser().getUserCode() == 'P')) {
                        clientProducerDists.add(new ClientProducerDist(client.getUser(), paths.get(producer).getLast().getUser(), dists.get(producer)))
                    }
                }
            } else {
                System.out.println("invalid Vertex");
            }
        }
        System.out.println("Graph Connected");
        return clientProducerDists;
    }else {
        System.out.println("Graph not connected");
    }
    return null;
}
```

Este método tem complexidade no pior caso de $O(n^4)$

US303

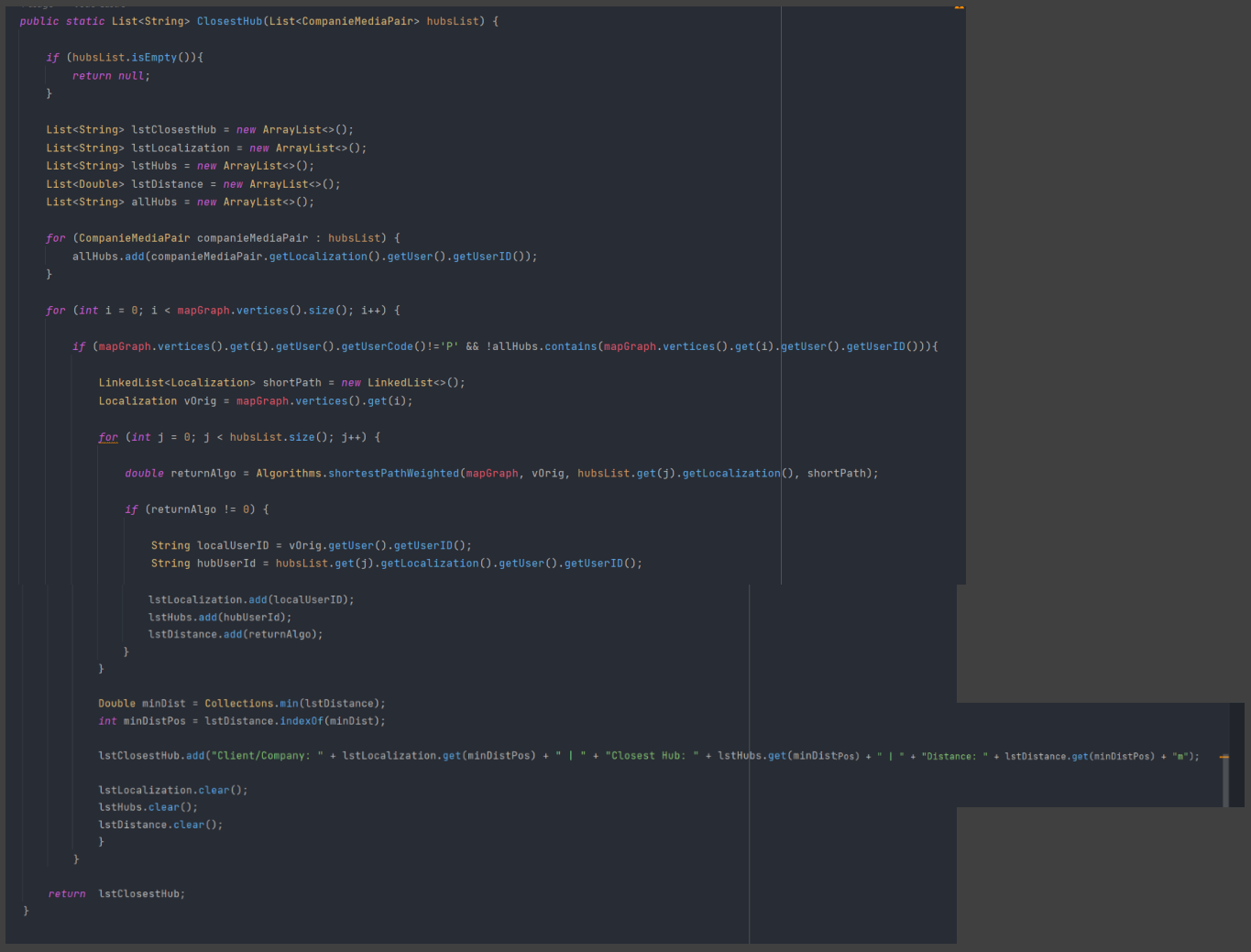
```java
public static List<CompanieMediaPair> US303(int n) {
    ArrayList<CompanieMediaPair> companieMediaPairs = new ArrayList<>();
    CompanieMediaPair companieMediaPair;
    double acumulador = 0;
    int contador;
    for (Localization l : companiesList) {
        contador = 0;
        Algorithms.shortestPathsWeighted(mapGraph, l, paths, dists);
        for (int x = 0; x < paths.size(); x++) {
            if (paths.get(x).getLast().getUser().getUserCode() != 'E') {
                acumulador += dists.get(x);
                contador++;
            }
        }
        companieMediaPair = new CompanieMediaPair(l, distancia: acumulador / contador);
        companieMediaPairs.add(companieMediaPair);
    }

    Collections.sort(companieMediaPairs);
    List<CompanieMediaPair> newList;
    if (n < companieMediaPairs.size()) {
        newList = companieMediaPairs.subList(0, n);
    } else {
        newList = companieMediaPairs;
    }

    GraphStore.hubs = new ArrayList<>(newList);
    return newList;
}
```

Este método tem complexidade no pior caso de $O(n^4)$

## US304



```
public static List<String> ClosestHub(List<CompanieMediaPair> hubsList) {

    if (hubsList.isEmpty()){
        return null;
    }

    List<String> lstClosestHub = new ArrayList<>();
    List<String> lstLocalization = new ArrayList<>();
    List<String> lstHubs = new ArrayList<>();
    List<Double> lstDistance = new ArrayList<>();
    List<String> allHubs = new ArrayList<>();

    for (CompanieMediaPair companieMediaPair : hubsList) {
        allHubs.add(companieMediaPair.getLocalization().getUser().getUserID());
    }

    for (int i = 0; i < mapGraph.vertices().size(); i++) {

        if (mapGraph.vertices().get(i).getUser().getUserCode()!='P' && !allHubs.contains(mapGraph.vertices().get(i).getUser().getUserID())){

            LinkedList<Localization> shortPath = new LinkedList<>();
            Localization vOrig = mapGraph.vertices().get(i);

            for (int j = 0; j < hubsList.size(); j++) {

                double returnAlgo = Algorithms.shortestPathWeighted(mapGraph, vOrig, hubsList.get(j).getLocalization(), shortPath);

                if (returnAlgo != 0) {

                    String localUserID = vOrig.getUser().getUserID();
                    String hubUserId = hubsList.get(j).getLocalization().getUser().getUserID();

                    lstLocalization.add(localUserID);
                    lstHubs.add(hubUserId);
                    lstDistance.add(returnAlgo);
                }
            }

            Double minDist = Collections.min(lstDistance);
            int minDistPos = lstDistance.indexOf(minDist);

            lstClosestHub.add("Client/Company: " + lstLocalization.get(minDistPos) + " | " + "Closest Hub: " + lstHubs.get(minDistPos) + " | " + "Distance: " + lstDistance.get(minDistPos) + "m");

            lstLocalization.clear();
            lstHubs.clear();
            lstDistance.clear();
            }
        }

    return  lstClosestHub;
}
```

Este método tem complexidade no pior caso de $O(n^4)$

## US305

```
public static Graph<Localization, Float> Us305(Graph<Localization, Float> g){
    Graph<Localization, Float> noCompanies = noCompanies(g);
    return Algorithms.kruskall(noCompanies);
}
```

Este método tem complexidade no pior caso de $O(n^2)$

```java
private static Graph<Localization, Float> noCompanies(Graph<Localization, Float> g){
    ArrayList<Edge<Localization, Float>> listaEdges = new ArrayList<>(g.edges());
    ArrayList<Edge<Localization, Float>> listaNovaArestas = new ArrayList<>();
    ArrayList<Localization> listaNovaVertices = new ArrayList<>();
    Edge<Localization, Float> edgeAuxiliar;

    for (Edge<Localization, Float> edge : listaEdges){
        if (edge.getVDest().getUser().getUserCode() == 'E'){
            for (Localization user : g.adjVertices(edge.getVDest())){
                edgeAuxiliar = g.edge(edge.getVDest(), user);

                if (!(edgeAuxiliar.getVDest() == edge.getVOrig())) {
                    listaNovaArestas.add(new Edge<>(edge.getVOrig(), user, weight: edge.getWeight() + edgeAuxiliar.getWeight()));
                    listaNovaArestas.add(new Edge<>(user,edge.getVOrig(), weight: edge.getWeight() + edgeAuxiliar.getWeight()));
                    listaNovaVertices.add(edge.getVOrig());
                }
            }
        }else if (!(edge.getVOrig().getUser().getUserCode() == 'E')){
            listaNovaArestas.add(edge);
        }
    }

    Graph<Localization, Float> graphWithNoCompanies = new MapGraph<>( directed: false);
    for (Localization localization : listaNovaVertices){
        graphWithNoCompanies.addVertex(localization);
    }

    for (Edge<Localization, Float> edge : listaNovaArestas){
        graphWithNoCompanies.addEdge(edge.getVOrig(), edge.getVDest(), edge.getWeight());
    }

    return graphWithNoCompanies;
}
```

Este método tem complexidade no pior caso de $O(n^2)$

US306

```java
public static void rega() {
    Date currentDate = new Date(System.currentTimeMillis());
    Time currentTime = new Time(currentDate.getHours(), currentDate.getMinutes(), currentDate.getSeconds());
    boolean diaPar = false;
    Time[] horas = GraphStore.irrigationSystem.getHoras();
    boolean existeRegaAtiva = false;
    if (currentDate.getDate() % 2 == 0) {
        diaPar = true;
    }

    for (IrrigationObject io : GraphStore.irrigationSystem.getIrrigationObjectList()) {
        for (int x = 0; x < horas.length; x++) {
            Time timeComDuracao = new Time(horas[x].getTime() + (long) io.getDuracoes() * 60000);
            if (currentTime.after(horas[x]) && currentTime.before(timeComDuracao)) {
                Time tempoQueFalta = new Time(timeComDuracao.getTime() - currentTime.getTime());
                if (diaPar && (io.getRegularidades().equals("p") || io.getRegularidades().equals("t"))) {
                    existeRegaAtiva = true;
                    System.out.printf("A parcela %s esta a ser regada. Faltam %d minutos para terminar.%n", io.getParcelas(), tempoQueFalta.getTime() / 60000);
                } else if (!diaPar && (io.getRegularidades().equals("i") || io.getRegularidades().equals("t"))) {
                    existeRegaAtiva = true;
                    System.out.printf("A parcela %s esta a ser regada. Faltam %d minutos para terminar.%n", io.getParcelas(), tempoQueFalta.getTime() / 60000);
                }
            }
        }
    }
    if (!existeRegaAtiva) {
        System.out.println("Nenhuma parcela esta a ser regada.");
    }

}
```

Este método tem complexidade no pior caso de $O(n^2)$