Métodos que implementam o Kruskal's algorithm:

```
public static List<Aresta> encontrarArvoreGeradora(List<Aresta> edges,
List<Aresta> mstEdges) {

    edges.sort(Comparator.comparingInt(Aresta::getPeso));

    Map<String, String> parent = new HashMap<>();
    Map<String, Integer> rank = new HashMap<>();
    Set<String> vertices = new HashSet<>();


    for (Aresta edge : edges) {
        vertices.add(edge.getOrigem());
        vertices.add(edge.getDestino());
    }

    for (Aresta edge : edges) {
        String rootX = findRoot(parent, edge.getOrigem());
        String rootY = findRoot(parent, edge.getDestino());

        if (!rootX.equals(rootY)) {
            mstEdges.add(edge);

            modifiedUnion(parent, rank, rootX, rootY);

            if (mstEdges.size() == vertices.size() - 1) {
                break;
            }
        }
    }

    return mstEdges;
}


private static String findRoot(Map<String, String> parent, String
node) {
    if (!parent.containsKey(node)) {
        parent.put(node, node);
    }
    while (!parent.get(node).equals(node)) {
        node = parent.get(node);
    }
    return node;
}


private static void modifiedUnion(Map<String, String> parent,
Map<String, Integer> rank, String x, String y) {
    String rootX = findRoot(parent, x);
    String rootY = findRoot(parent, y);

    if (rootX.equals(rootY)) {
        return;
    }

    if (rank.getOrDefault(rootX, 0) < rank.getOrDefault(rootY, 0)) {
        parent.put(rootX, rootY);
    } else if (rank.getOrDefault(rootX, 0) > rank.getOrDefault(rootY,
```

```
0)) {
        parent.put(rootY, rootX);
    } else {
        parent.put(rootY, rootX);
        rank.put(rootX, rank.getOrDefault(rootX, 0) + 1);
    }
}
```

Classe que importa o csv e faz a respetiva divisão dos dados:

```
public class ImportarCsv {
    public static List<Aresta> lerGrafoDeCSV(String nomeArquivo)
throws FileNotFoundException {

        List<Aresta> arestas = new ArrayList<>();

        Scanner scanner = new Scanner(new File(nomeArquivo));

        while (scanner.hasNextLine()) {
            String[] linha = scanner.nextLine().split(";");
            String origem = linha[0];
            String destino = linha[1];
            int peso = Integer.parseInt(linha[2]);
            arestas.add(new Aresta(origem, destino, peso));
        }

        return arestas;
    }
}
```

Classe Aresta:

```
public class Aresta {
    private String origem;
    private String destino;
    private int peso;

    public Aresta(String origem, String destino, int peso) {
        this.origem = origem;
        this.destino = destino;
        this.peso = peso;
    }

    public String getOrigem() {
        return origem;
    }

    public int getPeso() {
        return peso;
    }

    public String getDestino() {
        return destino;
    }

}
```

Métodos que geram os Grafos:

```
public static void gerarOutputExcel(String caminhoCsv, String
nomeArquivoOutput) throws IOException {
    List<Aresta> arestas = ImportarCsv.lerGrafoDeCSV(caminhoCsv);
    List<Aresta> mstEdges = encontrarArvoreGeradora(arestas, new
ArrayList<>());
    int totalCost = 0;

    try (PrintWriter writer = new PrintWriter(new
File(nomeArquivoOutput))) {
        StringBuilder sb = new StringBuilder();
        sb.append("Origem, Destino, Peso\n");

        for (Aresta aresta : mstEdges) {
            sb.append(aresta.getOrigem()).append(",");
            sb.append(aresta.getDestino()).append(",");
            sb.append(aresta.getPeso()).append("\n");
            totalCost += aresta.getPeso();
        }

        System.out.println("Cost of a Minimun spanning tree = " +
totalCost);
        writer.write(sb.toString());
    }

    // Gera o arquivo DOT
    generateDotFile(mstEdges.toArray(new Aresta[0]), nomeArquivoOutput
+ ".dot");

    // Renderiza o arquivo DOT como imagem PNG
    renderDotFile(nomeArquivoOutput + ".dot", nomeArquivoOutput +
".png");

}

public static void generateDotFile(Aresta[] resultado, String
filename) throws IOException {
    try (FileWriter writer = new FileWriter(filename)) {
        writer.write("graph G {\n");
        for (Aresta aresta : resultado) {
            if (aresta.getPeso() != 0) {
                writer.write(aresta.getOrigem() + " -- " +
aresta.getDestino() + " [label=\"" + aresta.getPeso() + "\"];\n");
            }
        }
        writer.write("}\n");
    }
}

public static void renderDotFile(String dotFileName, String
outputFileName) throws IOException {
    String[] cmd = {"dot", "-Tpng", dotFileName, "-o",
outputFileName};
    ProcessBuilder pb = new ProcessBuilder(cmd);
    pb.redirectErrorStream(true);
    Process process = pb.start();

    try {
        process.waitFor();
    } catch (InterruptedException e) {
```

```
        e.printStackTrace();
    }
}
```

Métodos que correm os testes e geram o respetivo grafo:

```
public static void runAlgorithmTests(String ficheiroInput, String
ficheiroOutputCsv, String ficheiroOutputPng) {
    try (PrintWriter csvWriter = new PrintWriter(new
File(ficheiroOutputCsv))) {
        csvWriter.println("Tamanho da Entrada,Tempo de Execução
(ms)");

        for (int i = 1; i <= 30; i++) {
            List<Aresta> arestas =
ImportarCsv.lerGrafoDeCSV(ficheiroInput + i + ".csv");

            long startTime = System.currentTimeMillis();
            List<Aresta> mstEdges = encontrarArvoreGeradora(arestas,
new ArrayList<>());
            long endTime = System.currentTimeMillis();
            long executionTime = endTime - startTime;

            csvWriter.println(arestas.size() + "," + executionTime);
        }

        csvWriter.flush();

        List<String> lines =
Files.readAllLines(Paths.get(ficheiroOutputCsv));
        int[] inputSizes = new int[lines.size() - 1];
        long[] executionTimes = new long[lines.size() - 1];

        for (int i = 1; i < lines.size(); i++) {
            String[] parts = lines.get(i).split(",");
            inputSizes[i - 1] = Integer.parseInt(parts[0]);
            executionTimes[i - 1] = Long.parseLong(parts[1]);
        }

        generateExecutionTimeGraphic(inputSizes, executionTimes,
ficheiroOutputPng);

    } catch (IOException ex) {
        System.err.println("Erro ao escrever no arquivo CSV: " +
ex.getMessage());
    }
}

public static void generateExecutionTimeGraphic(int[] inputSizes,
long[] executionTimes, String outputFileName) {
    try {
        String gnuplotCommands = "set terminal png\n"
                + "set output '" + outputFileName + "'\n"
                + "set title 'Tempos de Execução'\n"
                + "set xlabel 'Tamanho da Entrada'\n"
                + "set ylabel 'Tempo de Execução (ms)'\n"
                + "plot '-' with lines title 'Tempo de Execução'\n";

        for(int i = 0; i < inputSizes.length; ++i) {
            gnuplotCommands += inputSizes[i] + " " + executionTimes[i]
```

```
+ "\n";
        }

        gnuplotCommands += "e\n";

        ProcessBuilder processBuilder = new ProcessBuilder("gnuplot");
        Process process = processBuilder.start();
        process.getOutputStream().write(gnuplotCommands.getBytes());
        process.getOutputStream().flush();
        process.getOutputStream().close();
        int exitCode = process.waitFor();
    } catch (InterruptedException | IOException e) {
        e.printStackTrace();
    }
}
```