```java
/**
 * The US13 class represents a collection of methods for processing graph data and running the
Kruskal's algorithm.
 * This class provides methods to read input files, generate minimum spanning trees, and generate
Graphviz images.
 */
public class US13 {

    /**
     * Reads a list of input file names from the specified file.
     *
     * @param filename The name of the file containing input file names.
     * @return A list of input file names.
     */
    public static List<String> readInputFiles(String filename) {
        List<String> fileNames = new ArrayList<>();
        try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
            String line;
            while ((line = reader.readLine()) != null) {
                fileNames.add(line.trim());
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return fileNames;
    }

    /**
     * Reads a graph from a CSV file and constructs a list of edges.
     *
     * @param filename The name of the CSV file containing graph data.
     * @return A list of edges representing the graph.
     */
    public static List<Edge> readGraphFromCSV(String filename) {
        List<Edge> edges = new ArrayList<>();
        try (Scanner scanner = new Scanner(new
FileReader("src/main/java/pt/ipp/isep/dei/esoft/project/mdisc/files/" + filename))) {
            while (scanner.hasNextLine()) {
                String[] parts = scanner.nextLine().split(";");
                String source = parts[0];
                String destination = parts[1];
                int weight = Integer.parseInt(parts[2]);
                edges.add(new Edge(source, destination, weight));
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return edges;
    }

    /**
     * Runs the Kruskal's algorithm on the provided list of edges.
     *
     * @param edges The list of edges representing the graph.
     * @return A list of edges representing the minimum spanning tree.
     */
    public static List<Edge> runAlgorithm(List<Edge> edges) {
        List<Edge> mstEdges = new ArrayList<>();
        return calculateMinimumSpanningTree(edges, mstEdges);
    }

    /**
     * Calculates the minimum spanning tree using Kruskal's algorithm.
     *
     * @param edges    The list of edges representing the graph.
     * @param mstEdges An empty list to store the minimum spanning tree edges.
     * @return A list of edges representing the minimum spanning tree.
     */
    public static List<Edge> calculateMinimumSpanningTree(List<Edge> edges, List<Edge> mstEdges) {
        edges.sort(Comparator.comparingInt(Edge::getWeight));

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

        for (Edge edge : edges) {
            vertices.add(edge.getFrom());
            vertices.add(edge.getTo());
        }

        for (Edge edge : edges) {
            String rootX = find(parent, edge.getFrom());
            String rootY = find(parent, edge.getTo());

            // Check if adding this edge creates a cycle
            if (!rootX.equals(rootY)) {
                mstEdges.add(edge);
```

```
                // Union by rank to optimize the tree structure
                union(parent, rank, rootX, rootY);

                // If we have added enough edges, stop
                if (mstEdges.size() == vertices.size() - 1) {
                    break;
                }
            }
        }

        return mstEdges;
    }

    /**
     * Finds the root of the set containing the specified node.
     *
     * @param parent The parent map representing the disjoint-set forest.
     * @param node   The node whose root is to be found.
     * @return The root of the set containing the specified node.
     */
    private static String find(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;
    }

    /**
     * Combines two sets represented by the specified roots.
     *
     * @param parent The parent map representing the disjoint-set forest.
     * @param rank   The rank map representing the ranks of the trees.
     * @param x      The root of the first set.
     * @param y      The root of the second set.
     */
    private static void union(Map<String, String> parent, Map<String, Integer> rank, String x, String
y) {
        String rootX = find(parent, x);
        String rootY = find(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);
        }
    }

    /**
     * Generates Graphviz images for the input and minimum spanning tree graphs.
     *
     * @param fileName The name of the input CSV file.
     * @param edges    The list of edges representing the graph.
     */
    public static void generateGraphvizImages(String fileName, List<Edge> edges) {
        fileName = fileName.replace(".csv", "");
        String dotFileNameInput =
"src/main/java/pt/ipp/isep/dei/esoft/project/mdisc/files/output/input_" + fileName + ".dot";
        try (PrintWriter writer = new PrintWriter(new FileWriter(dotFileNameInput))) {
            writer.println("graph G {");
            for (Edge edge : edges) {
                writer.println(edge.getFrom() + " -- " + edge.getTo() + " [label=" + edge.getWeight()
+ "];");
            }
            writer.println("}");
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Generate MST graph
        List<Edge> mstEdges = runAlgorithm(edges);
        String dotFileNameOutput =
"src/main/java/pt/ipp/isep/dei/esoft/project/mdisc/files/output/mst_" + fileName + ".dot";
        try (PrintWriter writer = new PrintWriter(new FileWriter(dotFileNameOutput))) {
            writer.println("graph G {");
            for (Edge edge : mstEdges) {
                writer.println(edge.getFrom() + " -- " + edge.getTo() + " [label=" + edge.getWeight()
+ "];");
```

```
            }
            writer.println("}");
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Generate images
        String inputImageFileName = "input_" + fileName + ".png";
        String mstImageFileName = "mst_" + fileName + ".png";

        try {
            ProcessBuilder processBuilder = new ProcessBuilder("dot", "-Tpng", dotFileNameInput, "-
o", "src/main/java/pt/ipp/isep/dei/esoft/project/mdisc/files/output/" + inputImageFileName);
            Process process = processBuilder.start();
            int exitCode = process.waitFor();
            if (exitCode != 0) {
                throw new IOException("Graphviz process exited with non-zero status: " + exitCode);
            }

            processBuilder = new ProcessBuilder("dot", "-Tpng", dotFileNameOutput, "-o",
"src/main/java/pt/ipp/isep/dei/esoft/project/mdisc/files/output/" + mstImageFileName);
            process = processBuilder.start();
            exitCode = process.waitFor();
            if (exitCode != 0) {
                throw new IOException("Graphviz process exited with non-zero status: " + exitCode);
            }
        } catch (IOException | InterruptedException e) {
            e.printStackTrace();
        }
    }

    /**
     * The main method to execute the operations on the input files.
     *
     * @param args Command-line arguments (not used).
     */
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the file name: ");
        String fileName = scanner.nextLine();
        List<String> inputFiles =
readInputFiles("src/main/java/pt/ipp/isep/dei/esoft/project/mdisc/files/" + fileName);

        for (String inputFile : inputFiles) {
            List<Edge> edges = readGraphFromCSV(inputFile);
            generateGraphvizImages(inputFile, edges);
        }
    }
}
```

Explicação do código da US13

Luís Aquino - 1221286

Nuno Teixeira - 1231375

Marta Domingues - 1231183

Mariana Sousa - 1230792

1DC

Reestruturando a explicação do código de modo a tornar todos os métodos que criamos mais compreensíveis com todos os detalhes e especificações:

1. **readInputFiles(String filename)**: Este método lê uma lista de nomes de arquivos de entrada de um arquivo especificado. Posteriormente, retorna uma lista de nomes de arquivos.

1.1. Parâmetro:

* **filename:** Uma **string** que representa o nome do arquivo que contém a lista de nomes de arquivos de entrada.

1.2. Retorno:

* Uma lista de **strings** contendo os nomes dos arquivos de entrada.

1.3. Funcionamento:

*O método abre o arquivo especificado usando um **BufferedReader,** que permite ler linhas do arquivo de forma eficiente;

*Ele lê cada linha do arquivo e adiciona o nome do arquivo à lista **fileNames;**

*Qualquer espaço em branco ao redor do nome do arquivo é removido usando o método **trim()** antes de adicioná-lo à lista;

*Se ocorrerem exceções de E/S durante a leitura do arquivo, como se o arquivo não existir, o método imprimirá a pilha de chamadas de exceção usando **e.printStackTrace()** e retornará uma lista vazia.

1.4. Uso:

* Podemos chamar esta função fornecendo o nome do arquivo que contém a lista de nomes de arquivos de entrada. Ele retornará uma lista de **strings** contendo os nomes desses arquivos

2. **readGraphFromCSV(String filename):** Este método lê um grafo de um arquivo CSV e constrói uma lista de arestas. Retornando assim uma lista de objetos **Edge** representando o grafo.

2.1. Parâmetro:
* **filename:** Uma string que representa o nome do arquivo CSV que contém os dados do grafo.

2.2. Retorno:
* Uma lista de objetos **Edge** representando as arestas do grafo.

2.3. Funcionamento:

* O método abre o arquivo CSV especificado usando um **Scanner**, que permite ler dados de entrada de várias fontes.

*Ele itera sobre cada linha do arquivo CSV.

*Para cada linha, ele divide a linha em partes usando o delimitador **;** usando o método **split()**
*Em seguida, ele extrai o nome de nó de origem, o nome do nó de destino e o peso da aresta a partir das partes extraídas.

*Ele cria um objeto **Edge** com esses dados e o adiciona à lista de arestas.

*Se ocorrerem exceções de E/S durante a leitura do arquivo, como se o arquivo não existisse ou se o formato do arquivo estiver incorreto, o método imprimirá a pilha de chamadas de exceções usando **e.printStackTrace()** e retornará uma lista vazia.

2.4. Uso:
* Pode-se chamar essa função fornecendo o nome do arquivo CSV que contém os dados do grafo. Ele retornará uma lista de arestas representado o grafo.

Concluindo, ao chamar o readGraphFromCSV("grapah_data.csv"), vamos obter uma lista contendo os objetos **Edge** representando as arestas do grafo, onde cada objeto **Edge** contém informações sobre a origem, o destino e o peso da aresta.

3. **runAlgorithm(List<Edge> edges):** Este método executa o algoritmo de Kruskal na lista de arestas fornecida. Ele retorna uma lista de arestas representando a árvore de spanning mínima.

3.1. Parâmetro:

* **edges**: Uma lista de objetos **Edge** representando as arestas do grafo.

3.2. Retorno:

* Uma lista de objetos **Edge** representando as arestas de árvore de spanning mínima.

3.3. Funcionamento:

* O método simplesmente chama outra função chamada **calculateMinimumSpanningTree(edges, mstEdges)** , passando a lista de arestas fornecida como argumento, juntamente com uma lista vazia **mstEdges** que será preenchida com as arestas de árvore de spanning mínima.

*Ele retorna o resultado da chamada dessa função.

3.4. Uso:

* Podemos chamar esta função fornecendo uma lista de arestas representando o grafo. Ele retornará uma lista de arestas representando a árvore de spanning mínima gerado pelo algoritmo de Kruskal.

Essencialmente, esta função é uma camada de abstração que permite ao usuário executar o algoritmo de Kruskal de forma mais simples, fornecendo apenas a lista de arestas do grafo como entrada e obtendo a árvore de spanning mínima como saída.

4. **calculateMinimumSpanningTree(List<Edge> edges, List<Edge> mstEdges):** Este método calcula a árvore de spanning mínima usando o algoritmo de Kruskal. Ele classifica as arestas por peso, e então itera sobre elas, adicionando arestas à árvore do spanning mínima se elas não forem ciclos.

4.1. Parâmetros:

* **edges**: Uma lista de objetos **Edge** representando todas as arestas do grafo.

* **mstEdges:** Uma lista vazia de objetos **Edge** onde as arestas da árvore de spanning mínima serão armazenadas.

4.2. Retorno:

* Uma lista de objetos **Edge** representando a árvore de spanning mínima.

4.3. Funcionamento:

*Primeiro, as arestas na lista **edges** são classificadas por peso, em ordem crescente.

* Em seguida, são criadas duas estruturas de dados auxiliares:

1 - Um **Map** chamado **parent** para manter o conjunto de elementos e seus representantes na floresta de conjuntos disjuntos.

2 - Um **Map** chamado **rank** para manter a classificação (profundidade) de cada nó na floresta de conjuntos disjuntos.

3 - Um **Set** chamado **vertices** para armazenar todos os vértices do grafo.

* Em seguida, para cada aresta na lista **edges**, verifica-se se a adição dessa aresta à árvore de spanning mínima cria um ciclo. Isso é feito verificando se os nós de origem e destino da aresta estão no mesmo conjunto ou não (usando a **função find()**).

* Se a adição da aresta não criar um ciclo, ela é adicionada à lista **mstEdges** e os conjuntos a que os eles de origem e destino pertencem são unidos (usando a função **union()**).

* O processo continua até que todas as arestas tenham sido processadas ou até que a árvore de spanning mínima contenha (V-1) arestas, onde V é o número de vértices no grafo.

* Finalmente, a lista **mstEdges** contendo as arestas da árvore de spanning mínima é retornada.

4.4. Uso:

* Você pode chamar esta função fornecendo uma lista de arestas representando o grafo e uma lista vazia para armazenar as arestas da árvore de spanning mínima. Ela retornará a árvore de spanning mínima calculada pelo algoritmo de Kruskal.

Concluindo, esta função é essencial para o algoritmo de Kruskal, pois é onde ocorre a principal lógica de seleção das arestas para construir a árvore de spanning mínima.

5. **find(Map<String, String> parent, String node)** e **union(Map<String, String> parent, Map<String, Integer> rank, String x, String y):** Estes métodos são auxiliares para o algoritmo de Kruskal. **find** encontra o representante do conjunto de um nó e union une dois conjuntos usando a estratégia de união por classificação.

5.1. Método find(Map<String, String> parent, String node):

Este método é uma parte essencial do algoritmo de Kruskal e é usado para encontrar o representante (raiz) do conjunto ao qual um nó pertence na estrutura de dados de conjuntos disjuntos. Aqui está o funcionamento detalhado:

5.1.1. Parâmetro:

* **parent**: Um mapa que mapeia cada nó para o seu pai na árvore de conjuntos disjuntos.

* **node**: O nó para o qual queremos encontrar o representante (raiz) do conjunto.

5.1.2. Retorno:

* Retorna o representante (raiz) do conjunto ao qual o nó pertence.

5.1.3. Funcionamento:

* Se o nó não estiver presente no mapa **parent**, isso significa que é uma raiz de um conjunto. Nesse caso, o próprio nó é retornado como representante.

* Caso contrário, o método entra em um loop onde ele verifica se o nó atual tem um pai diferente dele mesmo. Se sim, ele sobe na árvore até encontrar a raiz, atualizando o nó atual para o seu pai em cada iteração.

* Quando a raiz é encontrada (ou seja, quando o pai do nó é ele mesmo), a raiz é retornada como representante do conjunto.

5.2. Método union(Map<String, String> parent, Map<String, Integer> rank, String x, String y):

Este método é usado para unir dois conjuntos usando a estratégia de união por classificação. Aqui está o funcionamento detalhado:

5.2.1. Parâmetros:

* **parent**: Um mapa que mapeia cada nó para o seu pai na árvore de conjuntos disjuntos.

* **rank**: Um mapa que mapeia cada nó para a sua classificação (profundidade) na árvore de conjuntos disjuntos.

* **x** e **y**: Os representantes (raízes) dos conjuntos que queremos unir.

5.2.2. Retorno:

* Não há retorno explícito, pois a função atualiza o mapa **parent** com a nova união.

5.2.3. Funcionamento:

* Primeiro, os representantes dos conjuntos aos quais **x** e **y** pertencem são encontrados usando a função **find().**

* Se **x** e **y** já pertencerem ao mesmo conjunto, não é necessário fazer nada, pois eles já estão unidos.

* Caso contrário, a união é realizada:

1- Se o conjunto representado por **x** tiver uma classificação (profundidade) menor do que o conjunto representado por **y**, **x** se torna o pai de **y**.

2- Se o conjunto representado por **y** tiver uma classificação menor do que o conjunto representado por **x**, **y** se torna o pai de **x**.

3- Se ambos tiverem a mesma classificação, um dos conjuntos é escolhido como o pai do outro, e a classificação do conjunto pai é incrementada.

Esses métodos são cruciais para a implementação eficiente do algoritmo de Kruskal, garantindo a correta união e busca nos conjuntos disjuntos durante a construção da árvore de spanning mínima.

6. **generateGraphvizImages(String fileName, List<Edge> edges):** Este método gera imagens Graphviz para o grafo de entrada e a árvore de spanning mínima. Ele cria arquivos **.dot** com a representação dos grafos e chama o Graphviz para gerar imagens PNG a partir desses arquivos.

6.1. Parâmetros:

* **fileName**: O nome do arquivo de entrada, que é usado para nomear as imagens geradas.

* **edges**: Uma lista de objetos Edge representando as arestas do grafo.

6.2. Funcionamento:

* Primeiro, o nome do arquivo de entrada é modificado para remover a extensão ".csv", já que esse nome será usado como base para nomear os arquivos **.dot** e as imagens **.png** geradas.

* Em seguida, é criado um arquivo **.dot** para representar o grafo de entrada e outro arquivo **.dot** para representar a árvore de spanning mínima.

* Para cada arquivo **.dot**, as arestas são escritas no formato DOT, onde cada aresta é representada por uma linha no formato "nó_de_origem -- nó_de_destino [label=peso_da_aresta];".
* Depois que os arquivos **.dot** são criados e preenchidos com as informações das arestas, o Graphviz é chamado duas vezes para gerar imagens **.png** a partir desses arquivos. Isso é feito usando o utilitário da linha de comando **dot**, que é parte do Graphviz.

* O processo de geração das imagens é realizado através de um objeto **ProcessBuilder** que chama o comando **dot** com os devidos argumentos, como o tipo de saída (-Tpng para PNG), o nome do arquivo **.dot** de entrada e o nome do arquivo **.png** de saída.

* O código verifica se o processo de geração das imagens foi bem-sucedido verificando o código de saída do processo. Se o código de saída for diferente de zero, indica que ocorreu um erro durante a geração da imagem e uma exceção é lançada.

6.3. Uso:

* Esta função é chamada após a leitura dos dados do grafo e a execução do algoritmo de Kruskal. Ela gera representações visuais do grafo de entrada e da árvore de spanning mínima, facilitando a visualização e a compreensão dessas estruturas.

Essa função é útil para gerar visualizações gráficas dos grafos, o que pode ser especialmente útil para entender melhor as características do grafo e a estrutura da árvore de spanning mínima encontrada pelo algoritmo de Kruskal.

7. **main(String[] args):** Este é o método principal que solicita ao usuário o nome do arquivo de entrada, lê os arquivos de entrada, gera as árvores de spanning mínimas correspondentes e gera imagens Graphviz para eles. Ele interage com o usuário através da entrada do console.

7.1. Parâmetros:

* **args**: É um array de strings que contém os argumentos passados para o programa a partir da linha de comando. Por exemplo, se você executar o programa com o comando **java NomeDoPrograma arg1 arg2**, então **args** será um array contendo os elementos "**arg1**" e "**arg2**".

7.2. Funcionamento:

* O método **main** começa solicitando ao usuário o nome do arquivo de entrada através do console, usando um objeto **Scanner**.

* Em seguida, chama o método **readInputFiles** para ler os nomes dos arquivos de entrada a partir do arquivo especificado pelo usuário.

* Para cada arquivo de entrada lido, o método lê o grafo do arquivo CSV correspondente usando o método **readGraphFromCSV**.

* Em seguida, chama o método **generateGraphvizImages** para gerar imagens Graphviz para o grafo de entrada e a árvore de spanning mínima correspondente.

* Este processo é repetido para cada arquivo de entrada fornecido pelo usuário.

7.3. Uso:

Este método é o ponto de entrada principal do programa Java. Ele coordena a execução das operações principais do programa, como leitura de arquivos, processamento de dados, execução de algoritmos e geração de saída. Ele interage com o usuário para obter informações de entrada e chama outros métodos para realizar as operações necessárias com base nessa entrada.

Em resumo, o método **main** é responsável por orquestrar a execução do programa, coordenando as diferentes operações e garantindo que tudo seja executado de acordo com a lógica definida pelo programador.