Source code from US 17

```
public class US17 implements Runnable{

    static final String MATRIX_STR = "datasets/us17_matrix.csv";
    static final String POINTS_STR = "datasets/us17_points_names.csv";
    static final String ASSEMBLY_POINT_STR = "AP";
    static final String NONE = "-";
    static final String GROUP_OPEN_CHAR = "(";
    static final String GROUP_CLOSE_CHAR = ")";
    static final String VERTEX_SEPARATOR = ",";
    static final String CSV_SEPARATOR = ";";//é pedida a ",", no entanto, esta está a
servir de separador decimal
    static final String DIJKSTRA_US17_FILENAME = "dijkstra_us17";
    static final String US17_DOT_FILENAME = "us17_dot";

    /**
     * Constantes necessárias para a aplicação
     */
    static final String OUTPUT_FOLDER = "output-files";    //pasta de ficheiros de
output
    static final String OUTPUT_TXT_EXTENSION = ".txt";     //extensão de ficheiros de
texto
    static final String OUTPUT_CSV_EXTENSION = ".csv";     //extensão de ficheiros
csv
    static final String MST_STRING = "MST";                //string utilizada para
concatenar no nome dos ficheiros de saída
    static final String GRAPH_STRING = "GRAPH";            //string utilizada para
concatenar no nome dos ficheiros de saída
    static final String IMAGE_EXTENSION_STRING = ".svg";   //extensão de ficheiros de
imagem
    static final String UNIX_DIRECTORY_SEPARARTOR = "/"; // separador de diretorios
UNIX
    static final String WINDOWS_DIRECTORY_SEPARARTOR = "\\"; // separador de
diretorios WINDOWS
    static final String GNUPLOT_SCRIPT_FILE_NAME = "script.gp"; // nome do ficheiro
de script do GNUPLOT


    public void run() {

        try {
            presentOptions();
        } catch (FileNotFoundException e) {
            throw new RuntimeException(e);
        }
    }

    private static void presentOptions() throws FileNotFoundException {
        String[] verticesNames = US17.readPointNamesData(POINTS_STR);
        double[][] matrix = US17.readEdgesData(MATRIX_STR);
        US17.printPointAndMatrixData(verticesNames, matrix);

        printInitalGraphToTXTFile(matrix, verticesNames, US17_DOT_FILENAME +
OUTPUT_TXT_EXTENSION);
        plotGraph(US17_DOT_FILENAME + OUTPUT_TXT_EXTENSION, "us17_initial_graph" +
IMAGE_EXTENSION_STRING);

        String opt = "";

        System.out.println("Choose one of the following options:");
        System.out.println("1 - All routes");
```

```
        System.out.println("2 - Choose a sign");
        System.out.println("0 - Exit");

        Scanner read = new Scanner(System.in);

        do {
            opt = read.nextLine();
            if (!opt.equals("1") && !opt.equals("2") && !opt.equals("0"))
                System.out.println("Insert the right option!");
        } while (!opt.equals("1") && !opt.equals("2") && !opt.equals("0"));

        switch (opt) {
            case "1":
                proceedToAllRoutes(verticesNames, matrix);
                break;
            case "2":
                proceedToOneRoute(verticesNames, matrix);
                break;
        }
    }

    private static void proceedToOneRoute(String[] verticesNames, double[][] matrix)
{

        printPointAndMatrixData(verticesNames, matrix);

        Scanner read = new Scanner(System.in);
        List<String> vertices = List.of(verticesNames);
        System.out.println("\nChoose the initial sign?");
        String vertex = read.nextLine();

        if (vertices.contains(vertex)) {
            us17Routine(matrix, verticesNames, vertex);
        } else
            System.out.println("Vertex not found!");
    }

    private static void us17Routine(double[][] matrix, String[] verticesNames, String
vertex) {
        double[][] copy = deepCopyMatrixDouble(matrix);
        double[] marcas = new double[verticesNames.length];
        String[] antecessor = new String[verticesNames.length];

        applyDijkstraAlg(copy, verticesNames, ASSEMBLY_POINT_STR, marcas,
antecessor);

        System.out.println(printShortestRoutesToCSV(verticesNames, antecessor,
marcas, vertex, DIJKSTRA_US17_FILENAME));


        printGraphToTXTFile(matrix, verticesNames, antecessor, US17_DOT_FILENAME +
OUTPUT_TXT_EXTENSION, vertex);
        plotGraph(US17_DOT_FILENAME + OUTPUT_TXT_EXTENSION, "point_" + vertex +
IMAGE_EXTENSION_STRING);
    }

    private static void proceedToAllRoutes(String[] verticesNames, double[][] matrix)
{
        for (String v : verticesNames) {
            us17Routine(matrix, verticesNames, v);
        }
    }

    static void plotGraph(String inputFile, String outputFile) {
```

```
        boolean isWindows =
System.getProperty("os.name").toLowerCase().startsWith("windows");

        try {

            if (isWindows) {

                ProcessBuilder pb1 = new ProcessBuilder("dot", "-Tsvg", OUTPUT_FOLDER
+ WINDOWS_DIRECTORY_SEPARARTOR + inputFile, "-o", OUTPUT_FOLDER +
WINDOWS_DIRECTORY_SEPARARTOR + outputFile);
                pb1.inheritIO();
                Process process1 = pb1.start();
                process1.waitFor();

            } else {

                ProcessBuilder pb1 = new ProcessBuilder("dot", "-Tsvg", OUTPUT_FOLDER
+ UNIX_DIRECTORY_SEPARARTOR + inputFile, "-o", OUTPUT_FOLDER +
UNIX_DIRECTORY_SEPARARTOR + outputFile);
                pb1.inheritIO();
                Process process1 = pb1.start();
                process1.waitFor();

            }
        } catch (IOException | InterruptedException e) {
            e.printStackTrace();
        }
    }

    static void printGraphToTXTFile(double[][] graphWeights, String[] vertices,
String[] antecessores, String filename, String vertice) {

        String[] verticesF = new String[vertices.length];
        String[] antecessoresF = new String[antecessores.length];
        filtraCaminhoMaisCurtoDoVertice(verticesF, antecessoresF, vertices,
antecessores, vertice);


        String line;
        boolean isWindows =
System.getProperty("os.name").toLowerCase().startsWith("windows");
        String outputFolderAndFile;

        try {

            if (isWindows) {
                outputFolderAndFile = String.format("%s%s", OUTPUT_FOLDER +
WINDOWS_DIRECTORY_SEPARARTOR, filename);
            } else {
                outputFolderAndFile = String.format("%s%s", OUTPUT_FOLDER +
UNIX_DIRECTORY_SEPARARTOR, filename);
            }


            PrintWriter printToFile = new PrintWriter(new File(outputFolderAndFile));
            printToFile.println(String.format("graph %s {\n",
filename.split("\\.")[0]));

            for (int i = 0; i < graphWeights.length; i++) {
                for (int j = 0; j < i; j++) {
                    if (graphWeights[i][j] > 0) {
                        //if (isPresent(i, j, vertices, antecessor)) {
                        if (isPresent(i, j, verticesF, antecessoresF)) {
```

```
                            line = String.format("\t%s -- %s [label=%d][color=red]",
vertices[i].trim(), vertices[j].trim(), (int) graphWeights[i][j]);
                        } else {
                            line = String.format("\t%s -- %s [label=%d]",
vertices[i].trim(), vertices[j].trim(), (int) graphWeights[i][j]);
                        }
                        printToFile.println(line);
                    }
                }
            }
            printToFile.println("\n}");

            printToFile.close();
        } catch (FileNotFoundException e) {
            System.out.println("Unable to write file.");
        }
    }

    static void printInitalGraphToTXTFile(double[][] graphWeights, String[] vertices,
String filename) {


        String line;
        boolean isWindows =
System.getProperty("os.name").toLowerCase().startsWith("windows");
        String outputFolderAndFile;

        try {

            if (isWindows) {
                outputFolderAndFile = String.format("%s%s", OUTPUT_FOLDER +
WINDOWS_DIRECTORY_SEPARARTOR, filename);
            } else {
                outputFolderAndFile = String.format("%s%s", OUTPUT_FOLDER +
UNIX_DIRECTORY_SEPARARTOR, filename);
            }


            PrintWriter printToFile = new PrintWriter(new File(outputFolderAndFile));
            printToFile.println(String.format("graph %s {\n",
filename.split("\\.")[0]));

            for (int i = 0; i < graphWeights.length; i++) {
                for (int j = 0; j < i; j++) {
                    if (graphWeights[i][j] > 0) {
                        //if (isPresent(i, j, vertices, antecessor)) {
                        line = String.format("\t%s -- %s [label=%d]",
vertices[i].trim(), vertices[j].trim(), (int) graphWeights[i][j]);
                        printToFile.println(line);
                    }
                }
            }
            printToFile.println("\n}");

            printToFile.close();
        } catch (FileNotFoundException e) {
            System.out.println("Unable to write file.");
        }
    }

    private static void filtraCaminhoMaisCurtoDoVertice(String[] verticesF, String[]
antecessorF, String[] vertices, String[] antecessores, String vertice) {

        int index = posicaoVertice(vertices, vertice);
```

```
        String antecessor;

        do {
            antecessor = antecessores[index];

            verticesF[index] = vertices[index];
            antecessorF[index] = antecessores[index];

            index = posicaoVertice(vertices, antecessor);

        } while (!antecessor.equals(NONE));

    }

    private static boolean isPresent(int i, int j, String[] vertices, String[]
antecessores) {
        return vertices[i] != null &&
                antecessores[i] != null &&
                vertices[j] != null &&
                antecessores[j] != null && (vertices[i].equals(antecessores[j]) ||
vertices[j].equals(antecessores[i]));
    }

    static String printShortestRoutesToCSV(String[] vertices, String[]
previousVertices, double[] weights, String vertex, String name) {

        String csvLine, requested = "";
        List<String> csvLines = new ArrayList<>();

        for (String v : vertices) {

            csvLine = getCsvLine(v, vertices, previousVertices, weights);

            if (v.equalsIgnoreCase(vertex)) {
                requested = csvLine;
            }

            csvLines.add(csvLine);
            printDataToFile(csvLines, name);
        }
        return requested;
    }

    private static String getCsvLine(String v, String[] vertices, String[]
antecessores, double[] marcas) {
        String csvLine = GROUP_OPEN_CHAR;
        int index = posicaoVertice(vertices, v);
        double custo = marcas[index];
        String antecessor;

        do {
            antecessor = antecessores[index];
            csvLine = csvLine + vertices[index];

            if (!antecessor.equals(NONE))
                csvLine = csvLine + VERTEX_SEPARATOR + " ";

            index = posicaoVertice(vertices, antecessor);

        } while (!antecessor.equals(NONE));
        return csvLine + String.format("%s%s %.2f", GROUP_CLOSE_CHAR, CSV_SEPARATOR,
custo);
    }
```

```
    private static void printDataToFile(List<String> csvLines, String name) {
        try {
            PrintWriter prtToFile = new PrintWriter(new File(OUTPUT_FOLDER +
UNIX_DIRECTORY_SEPARARTOR + name + OUTPUT_CSV_EXTENSION));

            for (String line : csvLines) {
                prtToFile.println(line);
            }

            prtToFile.close();

        } catch (FileNotFoundException e) {
            System.out.println("Unable to print to file");
        }
    }

    static double[][] deepCopyMatrixDouble(double[][] matrix) {
        double[][] aux = new double[matrix.length][matrix[0].length];
        for (int i = 0; i < matrix.length; i++) {
            for (int j = 0; j < matrix[i].length; j++) {
                aux[i][j] = matrix[i][j];
            }
        }
        return aux;
    }

    static void applyDijkstraAlg(double[][] matriz, String[] vertices, String
pInicial, double[] marcas, String[] antecessor) {

        boolean[] visitados = new boolean[vertices.length];

        for (int i = 0; i < vertices.length; i++) {
            marcas[i] = Double.MAX_VALUE;
            visitados[i] = false;
            antecessor[i] = "-";
        }

        int indice1 = posicaoVertice(vertices, pInicial);
        int indice2 = caminhoMaisCurto(indice1, matriz);

        marcas[indice1] = 0;

        while (indice1 >= 0) {
            if (indice2 >= 0) {
                if (marcas[indice2] > marcas[indice1] + matriz[indice1][indice2] &&
!visitados[indice1]) {
                    marcas[indice2] = marcas[indice1] + matriz[indice1][indice2];
                    matriz[indice1][indice2] = 0;
                    matriz[indice2][indice1] = 0;
                    antecessor[indice2] = vertices[indice1];
                    indice1 = marcaMinima(marcas, visitados);
                    indice2 = caminhoMaisCurto(indice1, matriz);
                } else {
                    matriz[indice1][indice2] = 0;
                    matriz[indice2][indice1] = 0;
                    indice1 = marcaMinima(marcas, visitados);
                    indice2 = caminhoMaisCurto(indice1, matriz);
                }
            } else {

                visitados[indice1] = true;
                indice1 = marcaMinima(marcas, visitados);
                if (indice1 >= 0)
                    indice2 = caminhoMaisCurto(indice1, matriz);
```

```
            }
        }
    }


    private static int marcaMinima(double[] marcas, boolean[] visitados) {
        double valAux = Double.MAX_VALUE;
        int index = -1;

        for (int i = 0; i < marcas.length; i++) {
            if (marcas[i] <= valAux && !visitados[i]) {
                valAux = marcas[i];
                index = i;
            }
        }

        return index;
    }

    private static int caminhoMaisCurto(int indice1, double[][] matriz) {
        double valAux = Double.MAX_VALUE;
        int index = -1;

        for (int i = 0; i < matriz[indice1].length; i++) {
            if (matriz[indice1][i] < valAux && matriz[indice1][i] > 0) {
                valAux = matriz[indice1][i];
                index = i;
            }
        }
        return index;
    }

    private static int posicaoVertice(String[] vertices, String pInicial) {
        for (int i = 0; i < vertices.length; i++) {
            if (vertices[i].equals(pInicial))
                return i;
        }
        return -1;
    }

    static void printPointAndMatrixData(String[] pointNamesData, double[][] matrix) {
        //print first line
        System.out.printf("%20s", "/");
        for (int i = 0; i < pointNamesData.length; i++) {
            System.out.printf("%20s", pointNamesData[i]);
        }
        System.out.print("\n");

        //print subsequent lines
        for (int i = 0; i < matrix.length; i++) {
            System.out.printf("%20s", pointNamesData[i]);
            for (int j = 0; j < matrix[i].length; j++) {
                System.out.printf("%20.2f", matrix[i][j]);
            }
            System.out.print("\n");
        }

        //System.out.print("\n");
    }

    /**
     * Auxiliary method to get point names data from file
     *
```

```
     * @param points
     * @return
     * @throws FileNotFoundException
     */
    public static String[] readPointNamesData(String points) throws
FileNotFoundException {
        String[] pointNames = null;

        Scanner readFile = new Scanner(new File(points));

        System.out.println("File found!");

        if (readFile.hasNext()) {
            pointNames = readFile.nextLine().split(";");
        }

        if (pointNames != null) {
            for (int i = 0; i < pointNames.length; i++) {
                pointNames[i] = pointNames[i].replace("\uFEFF", "");
            }
        }

        return pointNames;
    }

    /**
     * Auxiliary method to get point names data from file
     *
     * @param matrix
     * @return
     * @throws FileNotFoundException
     */
    public static double[][] readEdgesData(String matrix) throws
FileNotFoundException {
        List<double[]> edgesDataAux = new ArrayList<>();
        String[] auxStrVector;
        double[] auxFltVector;
        int i = 0;
        Scanner readFile = new Scanner(new File(matrix));

        System.out.println("File found!");

        while (readFile.hasNext()) {
            auxStrVector = readFile.nextLine().split(";");

            auxFltVector = new double[auxStrVector.length];

            for (int j = 0; j < auxStrVector.length; j++) {
                try {
                    auxFltVector[j] =
Double.parseDouble(auxStrVector[j].replace("\uFEFF", ""));
                } catch (NumberFormatException e) {
                    //System.err.println("Error parsing float value at line: " + i);
                    auxFltVector[j] = 0;
                }
            }

            edgesDataAux.add(auxFltVector);

            i++;
        }

        double[][] edgesData = new
double[edgesDataAux.size()][edgesDataAux.get(0).length];
```

```
        for (int j = 0; j < edgesDataAux.size(); j++) {
            edgesData[j] = edgesDataAux.get(j);
        }

        return edgesData;
    }
}
```

Source code from US18

```
public class US18 implements Runnable{
    static final String MATRIX_STR = "datasets/us18_matrix.csv";
    static final String POINTS_STR = "datasets/us18_points_names.csv";
    private static final String DIJKSTRA_US18_FILENAME = "dijkstra_us18";
    static final String US18_DOT_FILENAME = "us18_dot";


    public void run() {

        try {
            presentOptions();
        } catch (FileNotFoundException e) {
            throw new RuntimeException(e);
        }

    }

    private static void presentOptions() throws FileNotFoundException {
        String[] verticesNames = US17.readPointNamesData(POINTS_STR);
        double[][] matrix = US17.readEdgesData(MATRIX_STR);
        US17.printPointAndMatrixData(verticesNames, matrix);

        US17.printInitalGraphToTXTFile(matrix, verticesNames, US18_DOT_FILENAME +
US17.OUTPUT_TXT_EXTENSION);
        US17.plotGraph(US18_DOT_FILENAME + US17.OUTPUT_TXT_EXTENSION,
"us18_initial_graph" + US17.IMAGE_EXTENSION_STRING);


        String opt = "";

        System.out.println("Choose one of the following options:");
        System.out.println("1 - Routes for all signs");
        System.out.println("2 - Choose a sign");
        System.out.println("0 - Exit");

        Scanner read = new Scanner(System.in);

        do {
            opt = read.nextLine();
            if (!opt.equals("1") && !opt.equals("2") && !opt.equals("0"))
                System.out.println("Insert the right option!");
        } while (!opt.equals("1") && !opt.equals("2") && !opt.equals("0"));

        switch (opt) {
            case "1":
                proceedToAllVertices(verticesNames, matrix);
                break;
            case "2":
                proceedToOneVertex(verticesNames, matrix);
                break;
        }
```

```
    }

    private static void proceedToOneVertex(String[] verticesNames, double[][] matrix)
{
        Scanner read = new Scanner(System.in);
        List<String> vertices= List.of(verticesNames);
        System.out.println("\nChoose the initial sign?");
        String vertex = read.nextLine();
        if (vertices.contains(vertex)) {
            us18Routine(verticesNames, matrix, vertex);
        }else
            System.out.println("Vertex not found!");
    }

    private static void proceedToAllVertices(String[] verticesNames, double[][]
matrix) {
        Scanner read = new Scanner(System.in);

        String opt = "";
        System.out.println("Proceed to all vertices? (y/n)");
        do {
            opt = read.nextLine();
            if (!opt.equalsIgnoreCase("y") && !opt.equalsIgnoreCase("n"))
                System.out.println("Insert the correct option!");
        } while (!opt.equalsIgnoreCase("y") && !opt.equalsIgnoreCase("n"));

        if (opt.equalsIgnoreCase("y")) {
            List<Integer> apIndexes = identifyAP_indexes(verticesNames);

            List<String> verticesNamesCopy = new ArrayList<>();

            for (int i = 0; i < verticesNames.length; i++) {
                if (!apIndexes.contains(i))
                    verticesNamesCopy.add(verticesNames[i]);
            }

            System.out.println("Smallest routes");
            for (String v : verticesNamesCopy) {
                us18Routine(verticesNames, matrix, v);
            }
        }
    }

    private static void us18Routine(String[] verticesNames, double[][] matrix, String
vertice) {

        String pathAndCost;
        List<String> smalestPaths = new ArrayList<>();
        List<Double> smalestCosts = new ArrayList<>();
        List<double[][]> workedMatrixes = new ArrayList<>();
        List<String[]> workedVnames = new ArrayList<>();
        List<String[]> workedPrevVertixes = new ArrayList<>();


        List<Integer> apIndexes = identifyAP_indexes(verticesNames);


        //itera por cada vertice de AP removendo os restantes
        for (Integer i : apIndexes) {
            List<Integer> apIndexesAux = new ArrayList<>(List.copyOf(apIndexes));
            apIndexesAux.remove(i);
            double[][] workingMatrix = removeUnwantedMatrixIndexes(apIndexesAux,
matrix);
            String[] workingVNames = removeUnwantedVerticesIndexes(apIndexesAux,
```

```
verticesNames);

            double[][] copy = US17.deepCopyMatrixDouble(workingMatrix);
            double[] weights = new double[workingVNames.length];
            String[] prevVertices = new String[workingVNames.length];


            //aplica o dijkstra à matrix e vertices depois de removidos os valores
dos restantes APs
            US17.applyDijkstraAlg(copy, workingVNames, verticesNames[i], weights,
prevVertices);

            pathAndCost = US17.printShortestRoutesToCSV(workingVNames, prevVertices,
weights, vertice, DIJKSTRA_US18_FILENAME + "_" + verticesNames[i]);

            Double cost =
Double.parseDouble(pathAndCost.split(US17.CSV_SEPARATOR)[1].trim().replace(",",
"."));


            smalestPaths.add(pathAndCost);
            smalestCosts.add(cost);
            workedMatrixes.add(workingMatrix);
            workedVnames.add(workingVNames);
            workedPrevVertixes.add(prevVertices);
        }

        int smalestIndex = getSmallestIndex(smalestCosts);
        System.out.println(smalestPaths.get(smalestIndex));
    }

    private static int getSmallestIndex(List<Double> smallestCosts) {

        Double smallest = smallestCosts.get(0);
        int index = 0;
        for (int i = 1; i < smallestCosts.size(); i++) {
            if (smallestCosts.get(i) < smallest) {
                smallest = smallestCosts.get(i);
                index = i;
            }
        }
        return index;
    }

    private static String[] removeUnwantedVerticesIndexes(List<Integer> apIndexes,
String[] verticesNames) {
        String[] aux = new String[verticesNames.length - apIndexes.size()];
        int deleted = 0;

        for (int i = 0; i < verticesNames.length; i++) {
            if (!apIndexes.contains(i))
                aux[i - deleted] = verticesNames[i];
            else
                deleted++;
        }
        return aux;
    }

    private static double[][] removeUnwantedMatrixIndexes(List<Integer> apIndexes,
double[][] matrix) {
        int newSize = matrix.length - apIndexes.size();
        double[][] aux = new double[newSize][newSize];
        int newRow = 0, newCol;
```

```
        for (int i = 0; i < matrix.length; i++) {
            if (apIndexes.contains(i)) continue;
            newCol = 0;
            for (int j = 0; j < matrix[0].length; j++) {
                if (apIndexes.contains(j)) continue;
                aux[newRow][newCol] = matrix[i][j];
                newCol++;
            }
            newRow++;
        }
        return aux;
    }

    private static List<Integer> identifyAP_indexes(String[] verticesNames) {
        List<Integer> aux = new ArrayList<>();
        for (int i = 0; i < verticesNames.length; i++) {
            if (verticesNames[i].startsWith(US17.ASSEMBLY_POINT_STR))
                aux.add(i);
        }
        return aux;
    }

}
```