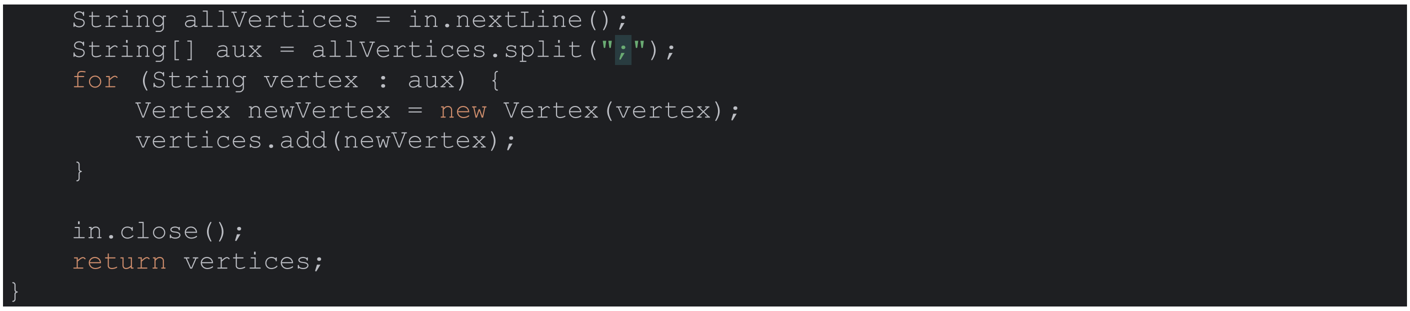
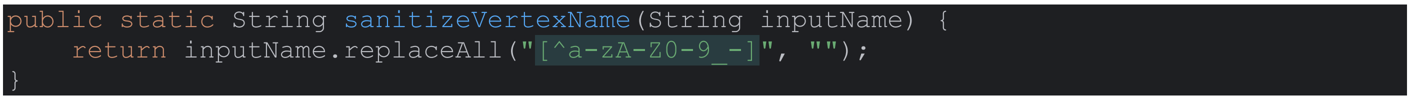
```
private static void US18() throws IOException, InterruptedException {
    Scanner sc = new Scanner(System.in);
    StringBuilder inputVerticesFile = new StringBuilder(getFileVertices(sc));
    StringBuilder inputFileWeights = new StringBuilder(getFileWeight(sc));
    StringBuilder fileWeights = new
StringBuilder("src/main/java/PI_MDISC_Group_072/Input/" + inputFileWeights + ".csv");
    StringBuilder fileVertices = new
StringBuilder("src/main/java/PI_MDISC_Group_072/Input/" + inputVerticesFile + ".csv");

    ArrayList<Vertex> vertices = readVertexFile(fileVertices);
    int i = 0;
    for (Vertex vertex : vertices) {
        String sanitizedVertexName = sanitizeVertexName(vertex.getV());
        vertices.get(i).setV(sanitizedVertexName);
        i++;
    }
    int[][] weights = readWeightFile(fileWeights);
    ArrayList<Edge> graphEdges = new ArrayList<>();
    makeEdges(graphEdges, vertices, weights);
    Graph graph = addEdges(graphEdges);
    createGraph(graph, inputVerticesFile);
    ArrayList<String> MPList = new ArrayList<>();
    sortMP(MPList, vertices);

    if (MPList.isEmpty()) {
        System.out.println("There is no Meeting Point in the file!");
    } else {
        List<Graph> evacuationRoutes;
        System.out.println("Insert the vertex you want to know the shortest path to the
AP or 'done'(if you want to stop):");
        String vertex = sc.nextLine();
        while (!vertex.equalsIgnoreCase("done")) {
            if (!vertex.equalsIgnoreCase("done")) {
                if (isPartOfVertices(vertices, vertex)) {
                    System.out.println("Vertex not found!");
                    System.out.println("Please insert a valid vertex that you want to
know the shortest path to the AP or 'done'(if you want to stop):");
                } else {
                    evacuationRoutes = DijkstraUS18(graphEdges, vertex, MPList);
                    List<Integer> totalCosts = new ArrayList<>();
                    for (Graph route : evacuationRoutes) {
                        totalCosts.add(route.getTotalCost());
                    }
                    bubbleSortCosts(totalCosts, evacuationRoutes);
                    Graph route = evacuationRoutes.get(0);
                    makeGraphCsv(route, vertex);
                    createGraphDisktra(graph, inputVerticesFile, route.getEdges(),
vertex);
                    System.out.println("Insert the vertex you want to know the shortest
path to the AP or 'done'(if you want to stop):");

                }
            }
            vertex = sc.nextLine();

        }
    }


}
```

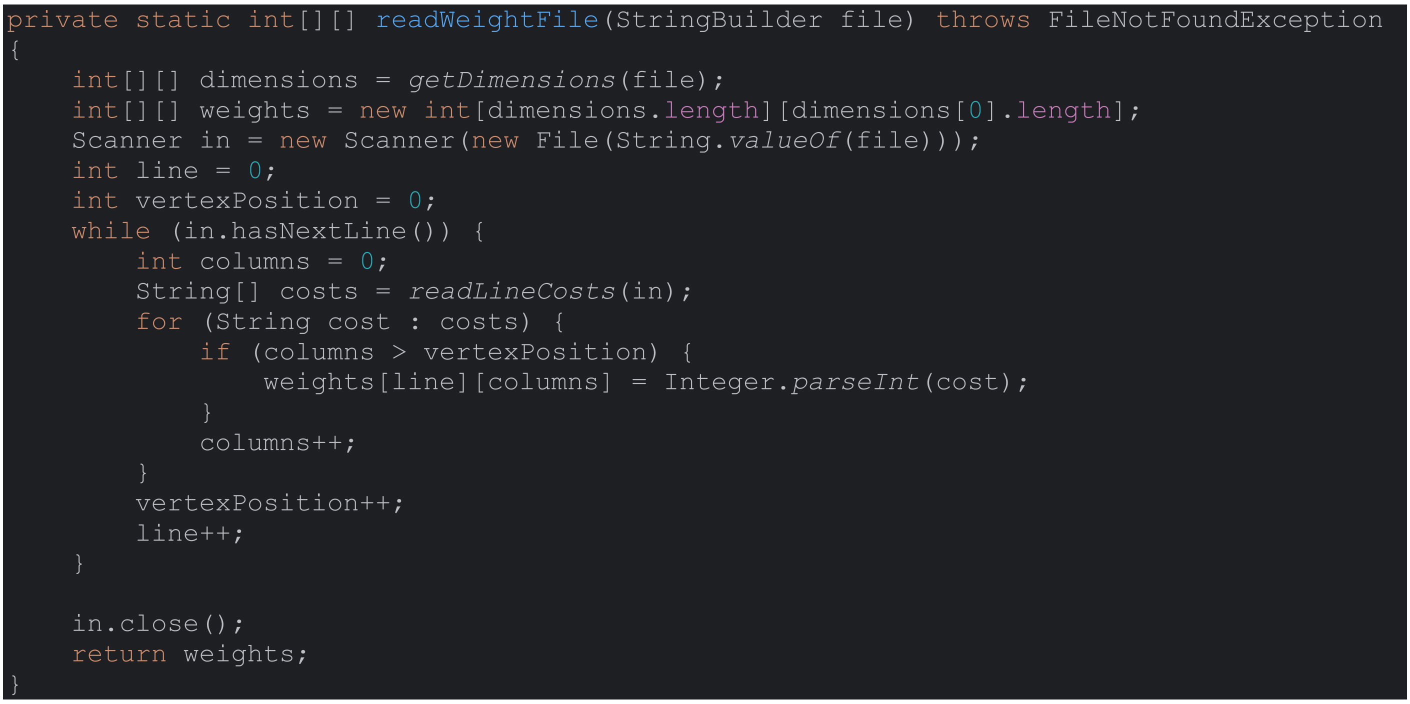
```
private static ArrayList<Vertex> readVertexFile(StringBuilder file) throws
FileNotFoundException {
    ArrayList<Vertex> vertices = new ArrayList<>();
    Scanner in = new Scanner(new File(String.valueOf(file)));
```



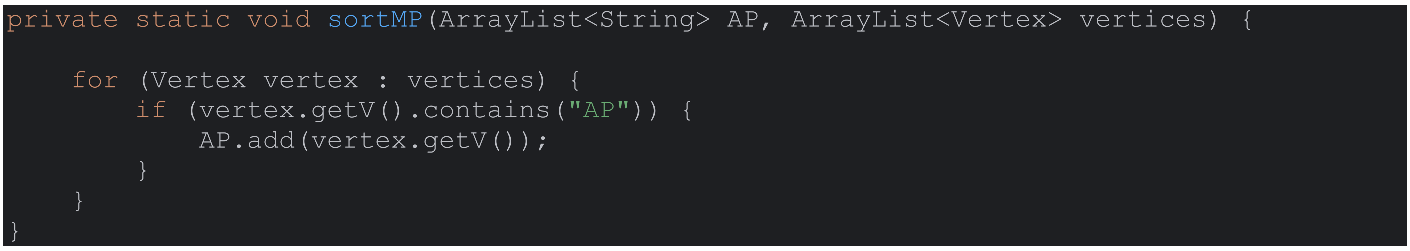
```
    String allVertices = in.nextLine();
    String[] aux = allVertices.split(";");
    for (String vertex : aux) {
        Vertex newVertex = new Vertex(vertex);
        vertices.add(newVertex);
    }

    in.close();
    return vertices;
}
```



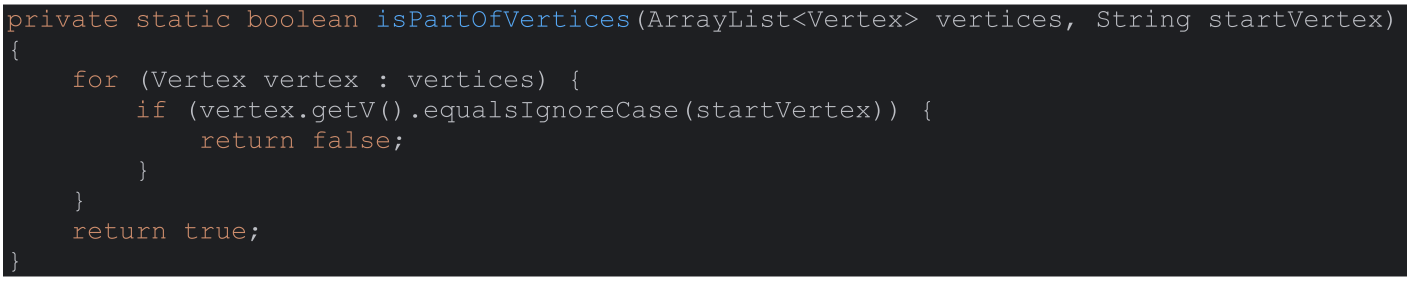
```
public static String sanitizeVertexName(String inputName) {
    return inputName.replaceAll("[^a-zA-Z0-9_-]", "");
}
```



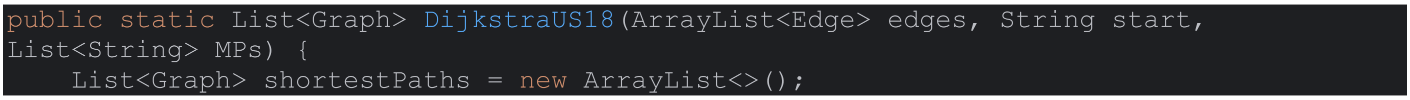
```
private static int[][] readWeightFile(StringBuilder file) throws FileNotFoundException
{
    int[][] dimensions = getDimensions(file);
    int[][] weights = new int[dimensions.length][dimensions[0].length];
    Scanner in = new Scanner(new File(String.valueOf(file)));
    int line = 0;
    int vertexPosition = 0;
    while (in.hasNextLine()) {
        int columns = 0;
        String[] costs = readLineCosts(in);
        for (String cost : costs) {
            if (columns > vertexPosition) {
                weights[line][columns] = Integer.parseInt(cost);
            }
            columns++;
        }
        vertexPosition++;
        line++;
    }

    in.close();
    return weights;
}
```



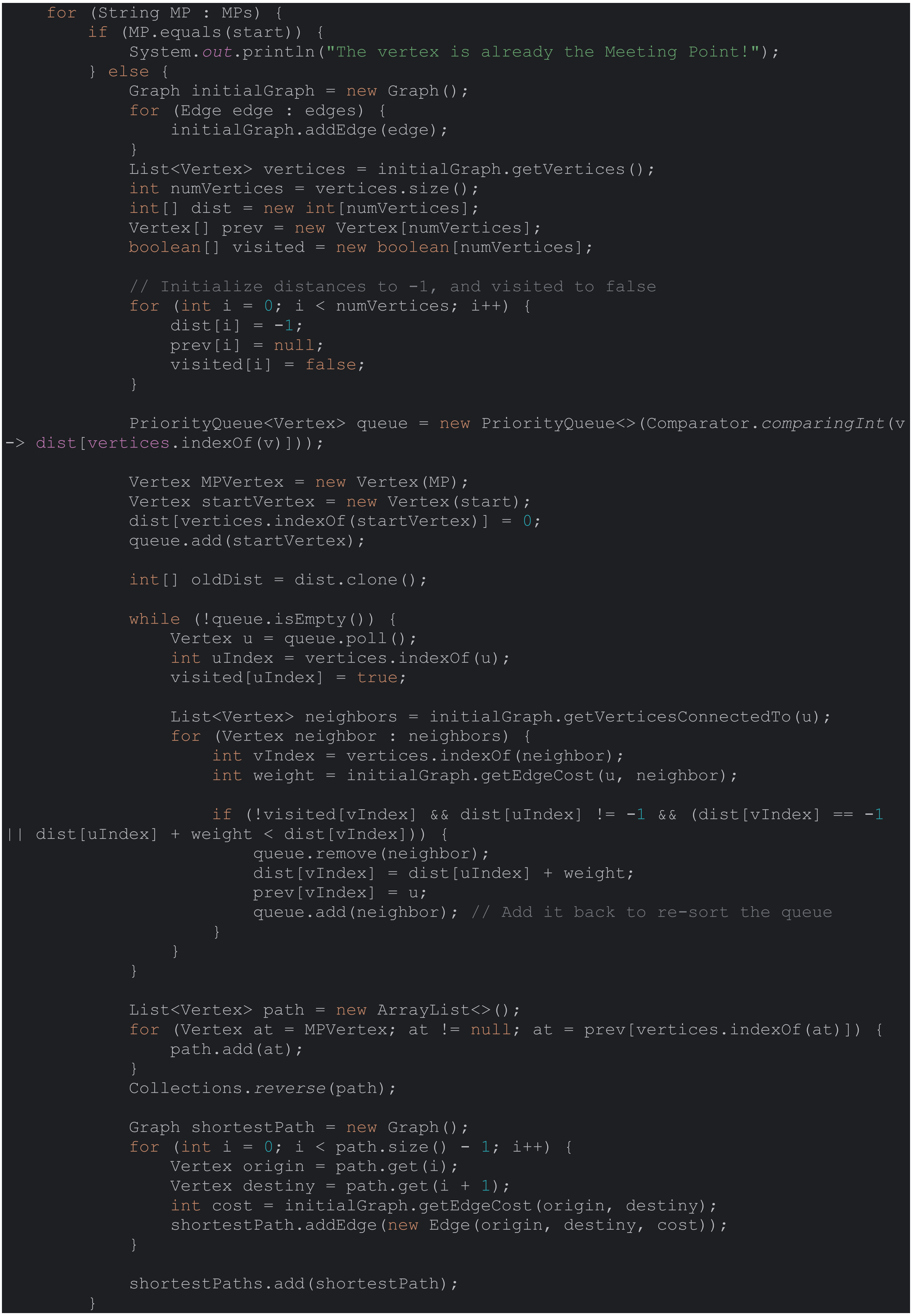
```
private static void sortMP(ArrayList<String> AP, ArrayList<Vertex> vertices) {

    for (Vertex vertex : vertices) {
        if (vertex.getV().contains("AP")) {
            AP.add(vertex.getV());
        }
    }
}
```



```
private static boolean isPartOfVertices(ArrayList<Vertex> vertices, String startVertex)
{
    for (Vertex vertex : vertices) {
        if (vertex.getV().equalsIgnoreCase(startVertex)) {
            return false;
        }
    }
    return true;
}
```



```
public static List<Graph> DijkstraUS18(ArrayList<Edge> edges, String start,
List<String> MPs) {
    List<Graph> shortestPaths = new ArrayList<>();
```



```java
    for (String MP : MPs) {
        if (MP.equals(start)) {
            System.out.println("The vertex is already the Meeting Point!");
        } else {
            Graph initialGraph = new Graph();
            for (Edge edge : edges) {
                initialGraph.addEdge(edge);
            }
            List<Vertex> vertices = initialGraph.getVertices();
            int numVertices = vertices.size();
            int[] dist = new int[numVertices];
            Vertex[] prev = new Vertex[numVertices];
            boolean[] visited = new boolean[numVertices];

            // Initialize distances to -1, and visited to false
            for (int i = 0; i < numVertices; i++) {
                dist[i] = -1;
                prev[i] = null;
                visited[i] = false;
            }

            PriorityQueue<Vertex> queue = new PriorityQueue<>(Comparator.comparingInt(v
-> dist[vertices.indexOf(v)]));

            Vertex MPVertex = new Vertex(MP);
            Vertex startVertex = new Vertex(start);
            dist[vertices.indexOf(startVertex)] = 0;
            queue.add(startVertex);

            int[] oldDist = dist.clone();

            while (!queue.isEmpty()) {
                Vertex u = queue.poll();
                int uIndex = vertices.indexOf(u);
                visited[uIndex] = true;

                List<Vertex> neighbors = initialGraph.getVerticesConnectedTo(u);
                for (Vertex neighbor : neighbors) {
                    int vIndex = vertices.indexOf(neighbor);
                    int weight = initialGraph.getEdgeCost(u, neighbor);

                    if (!visited[vIndex] && dist[uIndex] != -1 && (dist[vIndex] == -1
|| dist[uIndex] + weight < dist[vIndex])) {
                        queue.remove(neighbor);
                        dist[vIndex] = dist[uIndex] + weight;
                        prev[vIndex] = u;
                        queue.add(neighbor); // Add it back to re-sort the queue
                    }
                }
            }

            List<Vertex> path = new ArrayList<>();
            for (Vertex at = MPVertex; at != null; at = prev[vertices.indexOf(at)]) {
                path.add(at);
            }
            Collections.reverse(path);

            Graph shortestPath = new Graph();
            for (int i = 0; i < path.size() - 1; i++) {
                Vertex origin = path.get(i);
                Vertex destiny = path.get(i + 1);
                int cost = initialGraph.getEdgeCost(origin, destiny);
                shortestPath.addEdge(new Edge(origin, destiny, cost));
            }

            shortestPaths.add(shortestPath);
        }
    }
    return shortestPaths;
}
```

```
private static void bubbleSortCosts(List<Integer> totalCosts, List<Graph>
evacuationRoutes) {
    int n = totalCosts.size();
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (totalCosts.get(j) > totalCosts.get(j + 1)) {
                int temp = totalCosts.get(j);
                Graph tempGraph = evacuationRoutes.get(j);
                evacuationRoutes.set(j, evacuationRoutes.get(j + 1));
                totalCosts.set(j, totalCosts.get(j + 1));
                evacuationRoutes.set(j + 1, tempGraph);
                totalCosts.set(j + 1, temp);
            }
        }
    }
}
```