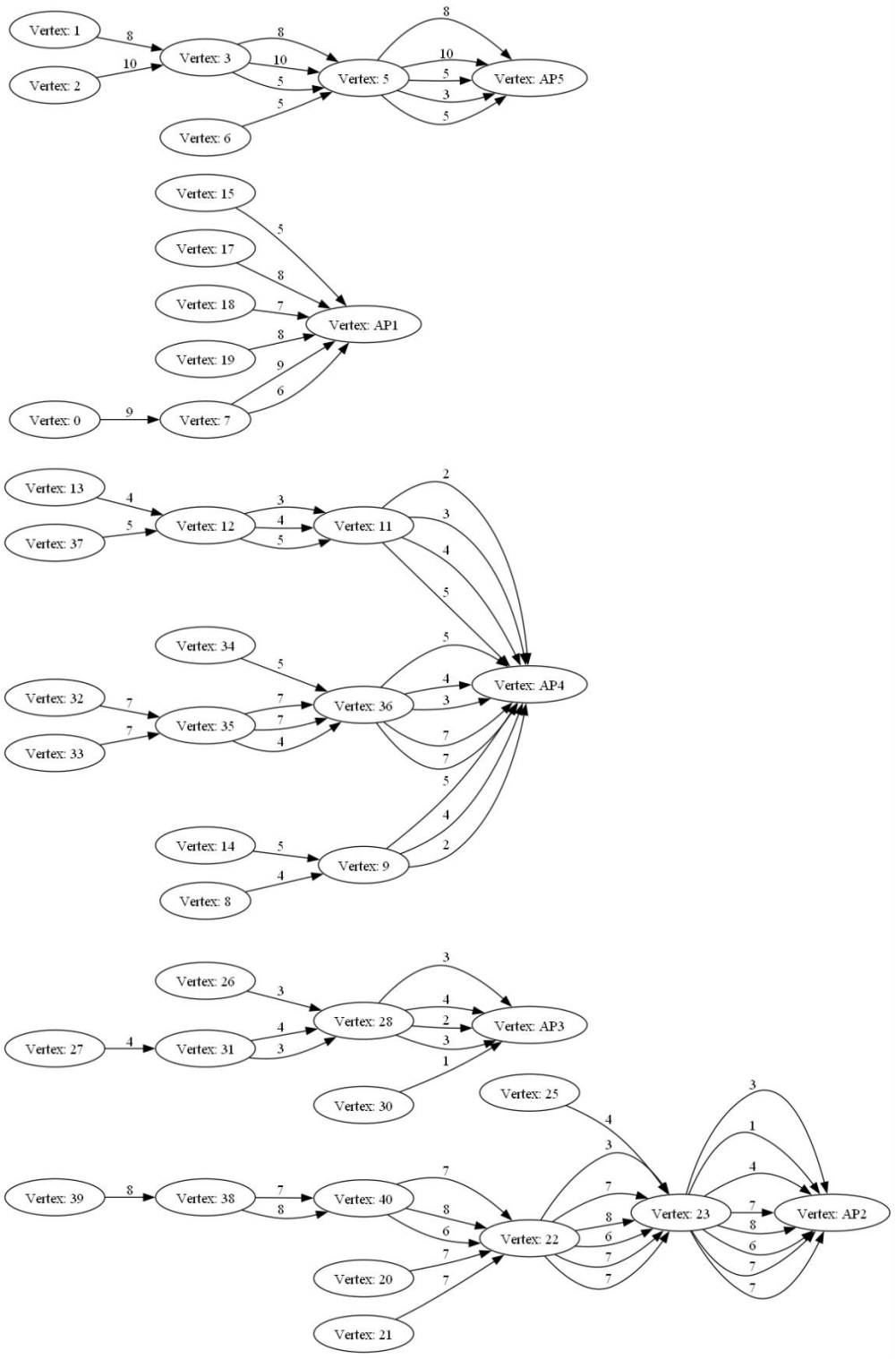
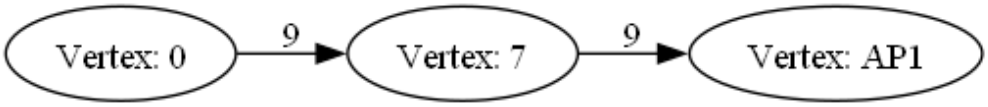
InputGraph

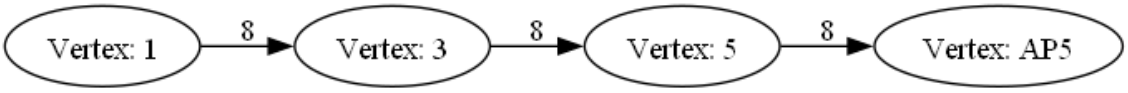

Vertex: 1
Vertex: 2
Vertex: 3
Vertex: 5
Vertex: AP5
Vertex: 6
Vertex: 15
Vertex: 17
Vertex: 18
Vertex: 19
Vertex: AP1
Vertex: 0
Vertex: 7
Vertex: 13
Vertex: 37
Vertex: 12
Vertex: 11
Vertex: 34
Vertex: 32
Vertex: 33
Vertex: 35
Vertex: 36
Vertex: AP4
Vertex: 14
Vertex: 8
Vertex: 9
Vertex: 26
Vertex: 27
Vertex: 31
Vertex: 28
Vertex: AP3
Vertex: 30
Vertex: 25
Vertex: 39
Vertex: 38
Vertex: 40
Vertex: 22
Vertex: 23
Vertex: AP2
Vertex: 20
Vertex: 21

ShortestRoutes

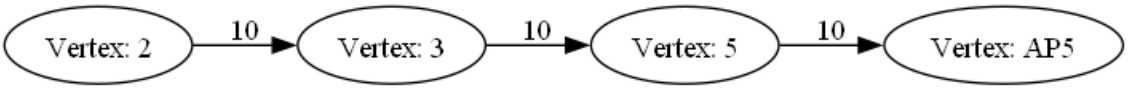
0



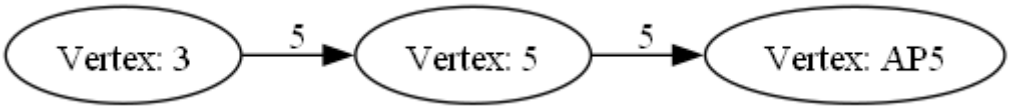
1



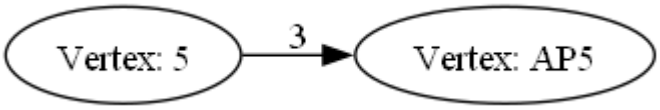
-2



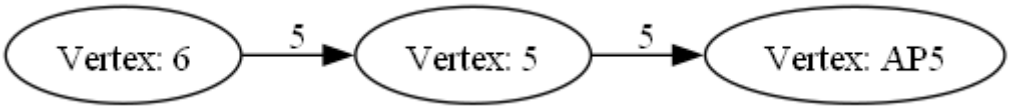
-3:



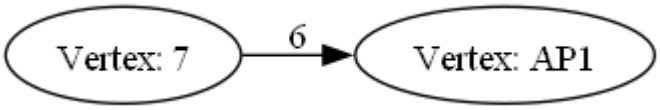
-5:



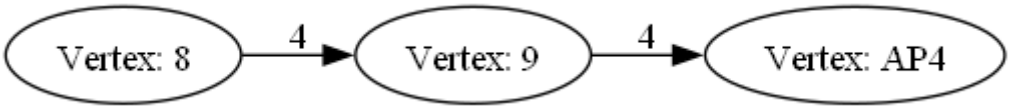
6:



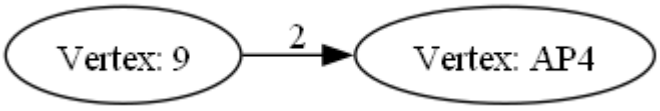
7:



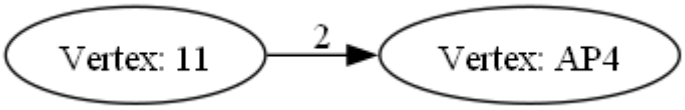
8



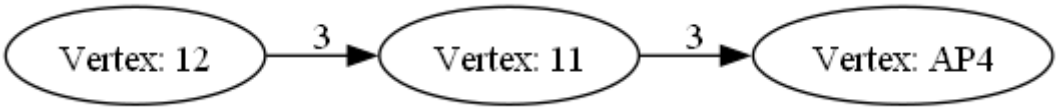
9



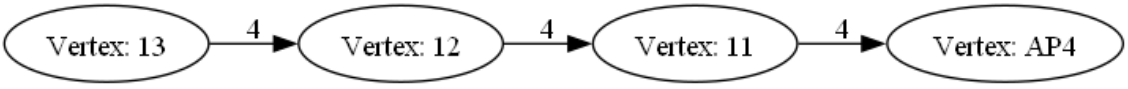
11



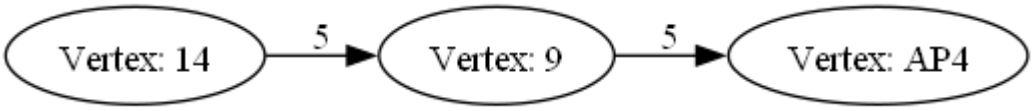
12



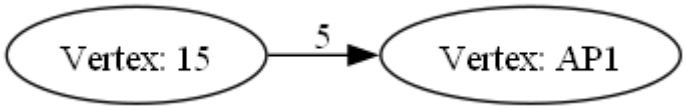
13



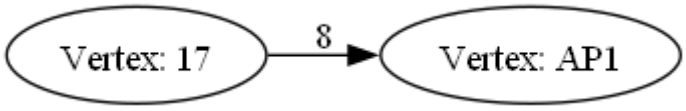
14



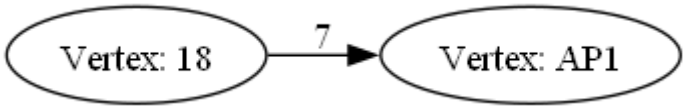
15



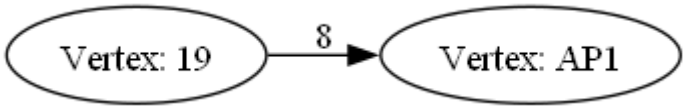
17



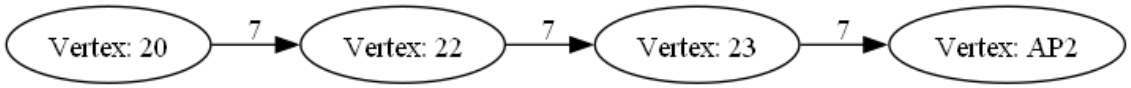
18



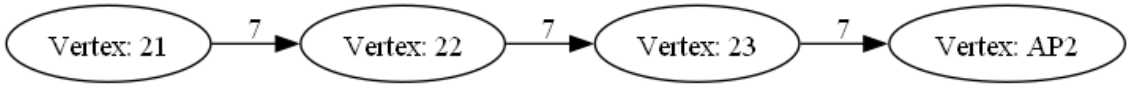
19



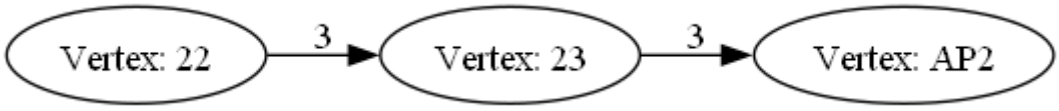
20



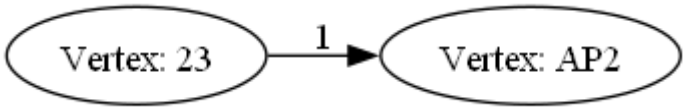
21



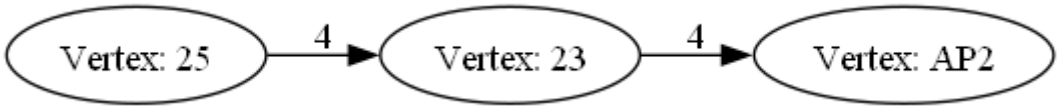
22



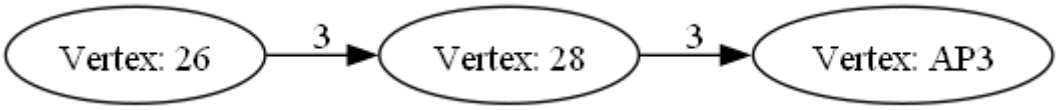
23



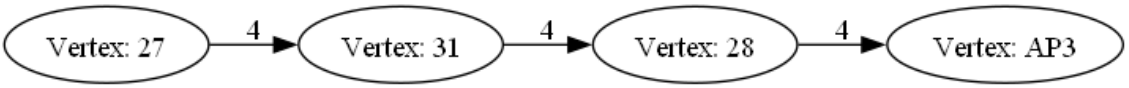
25



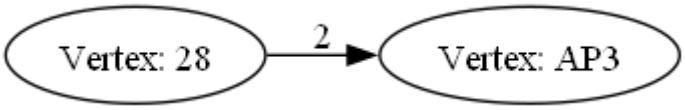
26



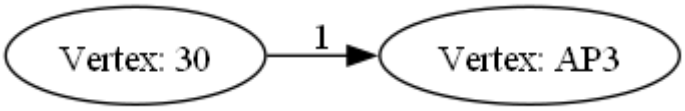
27



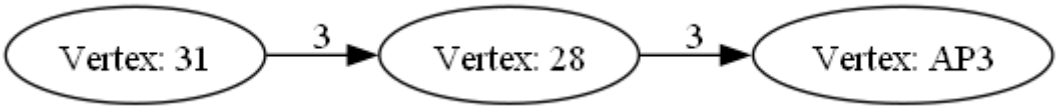
28



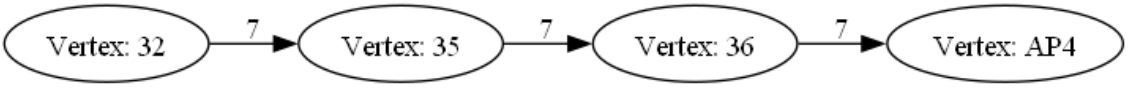
30



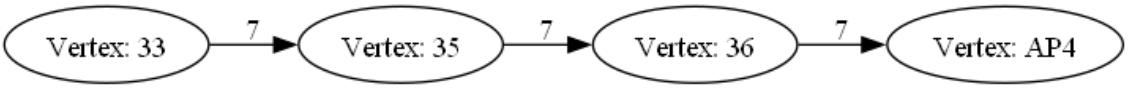
31



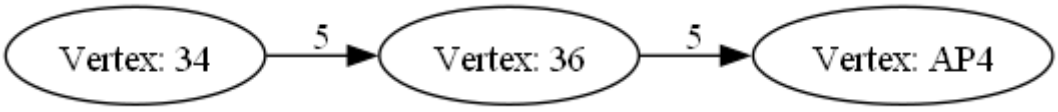
32



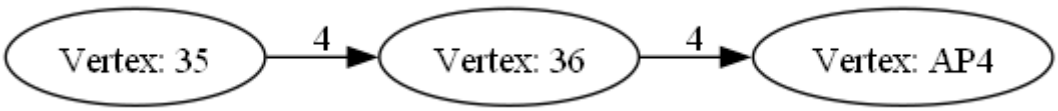
33



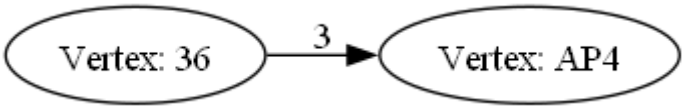
34



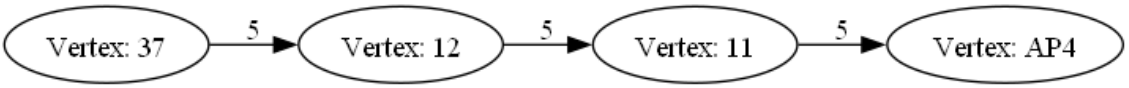
35



36

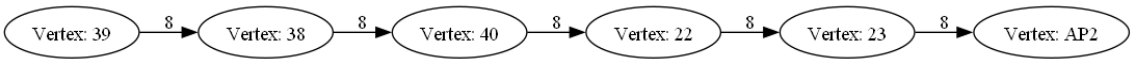


37

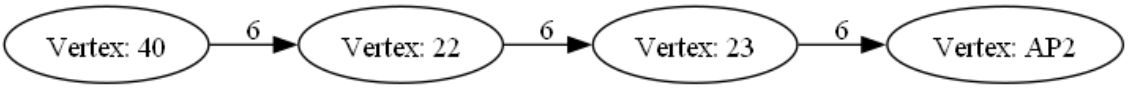


38

39



40



Implementation

Using the Dijkstra Alghorithm to calculate the Evacuation route:

```
private Path calculateShortestPath(List<String> points, int[][] matrix, String start) {
    int size = points.size();
    int startIndex = points.indexOf(start);

    int[] distances = new int[size];
    Arrays.fill(distances, Integer.MAX_VALUE);
    distances[startIndex] = 0;

    boolean[] visited = new boolean[size];
    String[] predecessors = new String[size];

    for (int i = 0; i < size; i++) {
        int u = findMinimumDistance(distances, visited);
        visited[u] = true;

        for (int v = 0; v < size; v++) {
            if (!visited[v] && matrix[u][v] != 0 && distances[u] != Integer.MAX_VALUE && distances[u] +
matrix[u][v] < distances[v]) {
                distances[v] = distances[u] + matrix[u][v];
                predecessors[v] = points.get(u);
            }
        }
    }

    int minDistance = Integer.MAX_VALUE;
    String nearestAP = null;
    for (String point : points) {
        if (point.startsWith("AP")) {
            int apIndex = points.indexOf(point);
            if (distances[apIndex] < minDistance) {
                minDistance = distances[apIndex];
                nearestAP = point;
            }
        }
    }

    List<String> path = new ArrayList<>();
    for (String point = nearestAP; point != null; point = predecessors[points.indexOf(point)]) {
        path.add(0, point);
```

```
    }

    return new Path(path, minDistance);
}
private static int findMinimumDistance(int[] distances, boolean[] visited) {
    int minIndex = -1;
    for (int i = 0; i < distances.length; i++) {
        if (!visited[i] && (minIndex == -1 || distances[i] < distances[minIndex])) {
            minIndex = i;
        }
    }
    return minIndex;
}
```

Results in csv:

```
private void writeShortestPath(String filename, String point, Path shortestPath) throws IOException {
    try (BufferedWriter writer = Files.newBufferedWriter(Paths.get(filename))) {
        String path = shortestPath.getPoints().stream()
                .map(p -> "Vertex: " + p)
                .collect(Collectors.joining(", "));
        writer.write(String.format("(%s); Cost: %d%n", path, shortestPath.getDistance()));
    }
}
private static int[][] readMatrix(String filename, int size) throws IOException {
    int[][] matrix = new int[size][size];
    List<String> lines = Files.readAllLines(Paths.get(filename));

    for (int i = 0; i < size; i++) {
        String[] values = lines.get(i).split(DELIMITER);
        for (int j = 0; j < size; j++) {
            String value = values[j];
            if (value.startsWith("¥uFEFF")) {
                value = value.substring(1);
            }
            matrix[i][j] = Integer.parseInt(value);
        }
    }
    return matrix;
}
```

```
private String getUserInput() {
    Scanner scanner = new Scanner(System.in);
    System.out.println("Enter the number of the point: ");
    return scanner.nextLine();
}



private static void writeShortestPaths(String filename, Map<String, Path> shortestPaths) throws
IOException {
    try (BufferedWriter writer = Files.newBufferedWriter(Paths.get(filename))) {
        for (Map.Entry<String, Path> entry : shortestPaths.entrySet()) {
            String path = entry.getValue().getPoints().stream()
                    .map(point -> "Vertex: " + point)
                    .collect(Collectors.joining(", "));
            writer.write(String.format("(%s); Cost: %d%n", path, entry.getValue().getDistance()));
        }
    }
}

private static List<String> readPoints(String filename) throws IOException {
    String content = Files.readString(Paths.get(filename));
    if (content.startsWith("¥uFEFF")) {
        content = content.substring(1);
    }
    return Arrays.asList(content.split(DELIMITER));
}
private static int[][] readMatrix(String filename, int size) throws IOException {
    int[][] matrix = new int[size][size];
    List<String> lines = Files.readAllLines(Paths.get(filename));

    for (int i = 0; i < size; i++) {
        String[] values = lines.get(i).split(DELIMITER);
        for (int j = 0; j < size; j++) {
            String value = values[j];
            if (value.startsWith("¥uFEFF")) {
                value = value.substring(1);
            }
            matrix[i][j] = Integer.parseInt(value);
        }
```

```
    }
    return matrix;
}


private String getUserInput() {
    Scanner scanner = new Scanner(System.in);
    System.out.println("Enter the number of the point: ");
    return scanner.nextLine();
}



private void writeShortestPath(String filename, String point, Path shortestPath) throws IOException {
    try (BufferedWriter writer = Files.newBufferedWriter(Paths.get(filename))) {
        String path = shortestPath.getPoints().stream()
                .map(p -> "Vertex: " + p)
                .collect(Collectors.joining(", "));
        writer.write(String.format("(%s); Cost: %d%n", path, shortestPath.getDistance()));
    }
}
```

GraphGenerate:

```
package mdisc.sprintc;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.nio.file.Paths;

public class GraphGenerator {

    public void generateGraph(String csvFile, String graphTitle) throws IOException,
InterruptedException {
        String dotFile = "src/main/java/mdisc/sprintc/graphsAndPng/" + graphTitle + ".dot";
        String pngFile = "src/main/java/mdisc/sprintc/graphsAndPng/" + graphTitle + ".png";
```

```
        try (BufferedReader reader = new BufferedReader(new FileReader(csvFile));
             BufferedWriter writer = new BufferedWriter(new FileWriter(dotFile))) {

            writer.write("digraph " + graphTitle + " {¥n");
            writer.write("rankdir=LR;¥n");

            String line;
            while ((line = reader.readLine()) != null) {
                String[] parts = line.split("; Cost: ");
                String path = parts[0].substring(1, parts[0].length() - 1);
                int cost = Integer.parseInt(parts[1]);

                String[] vertices = path.split(", ");
                for (int i = 0; i < vertices.length - 1; i++) {
                    writer.write("¥"" + vertices[i] + "¥" -> ¥"" + vertices[i + 1] + "¥" [label=¥"" + cost +
"¥"];¥n");
                }
            }

            writer.write("}¥n");
        }

        Process process = new ProcessBuilder("dot", "-Tpng", dotFile, "-o", pngFile).start();
        process.waitFor();
    }
}



    GraphGenerator graphGenerator = new GraphGenerator();
    try {
        graphGenerator.generateGraph("src/main/java/mdisc/sprintc/output/us18_allpoints.csv",
"AllPoints");
        graphGenerator.generateGraph("src/main/java/mdisc/sprintc/output/us18_output.csv",
"Output");
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```