FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

DEPARTMENT OF COMMUNICATION TECHNOLOGY AND NETWORK

CSC4202 - 6:

DESIGN AND ALGORITHM ANALYSIS

GROUP PROJECT

REPORT SAFE EVACUATION ROUTE IN POST-LANDSLIDE

LECTURER'S NAME: DR NUR ARZILAWATI BINTI MD YUNUS

NAME	MATRIC NO.
ALISYA ATHIRAH BINTI MOHD HUZZAINNY	211175
ADRIANA HUMAYRA BINTI SHALIZAN	213056
NURHAZWANI BINTI MUHAMMAD RADHI	213515

Table of Content

GitHub link:https://github.com/nurhazwaniradhi/CSC4202_Project_Submission.git

1.0 Scenario

A catastrophic landslide struck around The Mines Shopping Mall in Seri Kembangan, resulting in significant structural damage and blocking several pathways. This unexpected disaster has trapped numerous individuals inside, including ten Universiti Putra Malaysia (UPM) students. These students must find safe routes to return to their college dormitory, Kolej 12, on the university campus. The usual fastest routes are now unsafe, so they need to consider alternative paths that offer the shortest distance while ensuring their safety. The primary challenges in this scenario include navigating blocked roads and paths, dealing with the dynamically changing conditions caused by the landslide, and ensuring the safety of the individuals.

2.0 Importance of scenario

2.1 Safety and Survival

The foremost priority in this scenario is the safety and survival of the trapped individuals. Identifying and navigating safe escape routes are critical to prevent casualties and ensure everyone, including the ten UPM students, can safely evacuate the premises.

2.2 Efficient Resource Allocation

In disaster scenarios, rescue resources are often limited. Efficient allocation of these resources, guided by real-time information and optimal escape route planning, can maximize the number of people saved and ensure timely intervention where it's most needed.

2.3 Dynamic and Unpredictable Conditions

Landslides are dynamic events that can cause the environment to change rapidly. Initially, safe paths can become blocked, and new routes may open up as debris shifts. This necessitates a solution that can adapt in real time to these changing conditions, ensuring that escape plans remain viable throughout the rescue operation.

3.0 Algorithm Suitability Reviews

3.1 Sorting Algorithms

Sorting algorithms, while efficient in organizing data, are not suitable for this scenario due to their inability to handle the dynamic changes and complexities involved in navigating blocked and unblocked paths caused by the landslide. While they could prioritize students based on proximity to exits or the severity of their situation, they cannot adapt to the rapidly changing conditions within the mall.

3.2 Divide and Conquer (DAC) Algorithms

DAC algorithms, although effective in breaking down problems into independent subproblems, are less suitable for this scenario due to their interconnected and dynamic nature. The landslide scenario presents challenges with interdependencies and rapidly evolving conditions, which DAC algorithms may struggle to address efficiently, potentially leading to suboptimal escape routes and delays in evacuation.

3.3 Dynamic Programming (DP) Algorithms

Dynamic Programming (DP) algorithms, although effective in handling overlapping subproblems and optimal substructures, are not well-suited for finding escape routes in this scenario. The rapidly changing conditions and blocked paths due to the landslide require real-time adaptability, which DP algorithms struggle with. Their reliance on storing intermediate results does not effectively address the dynamic nature of the environment, making them less efficient for ensuring the safety of trapped individuals. Thus, DP algorithms are not an optimal choice for this scenario.

3.4 Greedy Algorithms

Greedy algorithms, particularly Dijkstra's algorithm, are well-suited for finding escape routes in a dynamic and unpredictable environment like a landslide and by making decisions based on the best immediate choice, Dijkstra's algorithm efficiently finds the shortest and safest

path. It considers the global impact of local decisions by continually updating path costs, ensuring optimal escape routes and enhancing the safety of those trapped in the mall.

3.5 Graph Algorithms

Graph algorithms, while typically effective for modeling environments like malls and finding shortest paths, are not suitable for this scenario. They struggle to account for the rapidly changing conditions and blocked paths caused by the landslide. These algorithms may not efficiently update in real-time to reflect the dynamic nature of the event, potentially leading to delays in evacuation. Therefore, despite their usual flexibility and efficiency, graph algorithms are not the best choice for ensuring the timely and safe evacuation of individuals in this unpredictable scenario.

4.0 Designing Algorithm To Solve Problem

Dijkstra's algorithm is chosen for this problem due to its effectiveness in handling graph-based pathfinding problems that require optimization of multiple factors such as distance and safety. Dijkstra's algorithm systematically explores the safest and shortest paths by evaluating each possible route, ensuring both time and space efficiency.

In the context of a catastrophic landslide at The Mines Shopping Mall in Seri Kembangan, where roads and pathways are blocked, Dijkstra's algorithm effectively addresses the urgent need to find safe evacuation routes for the trapped UPM students. Its ability to provide real-time, optimal solutions based on the evolving landscape and immediate hazards ensures that decisions are made quickly and effectively.

4.1 Algorithm Design

4.1.1 State Representation

The algorithm uses a simple state representation focusing on the current location and the paths leading from it. Each state includes current location which is the node representing the current position. Next is visited locations, where a set of locations that have been visited to avoid cycles.

4.1.2 Recurrence Relation

The core of Dijkstra's algorithm is the recurrence relation, which calculates the shortest path by continually updating the shortest known distance to each node.

$$dist[u] = min(dist[u], dist[v] + weight(u, v)$$

total_distance(v)=minDistance(u)+distance(p)+safetyScore(p)

4.1.3 Optimization Function

To extract the safest path, the algorithm continually updates the current location to the next location determined by the greedy choice. The greedy choice involves selecting the path with the lowest safety score, and if multiple paths have the same safety score, the one with the shortest distance is chosen. This process ensures that at each step, the decision made is locally optimal, thereby building a globally efficient and safe route.

While currentLocation != destination:

Choose nextPath with the lowest safety score (and shortest distance in case of a tie)

Move to nextPath.target

Mark currentLocation as visited

Update totalSafetyScore and totalDistance

4.2 Data type

Table 1 below shows the entity, attribute and the data type used for the Java coding usind Dijktra's algorithm.

Entity	Attribute	Data Type
Location	name paths	String List<Path>

	previous	Locacation
Path	target distance safetyScore	Location Double Double
Visited	locations	Set<Location>
PathFinder	start	Location
Dijktra's	nextPath totalDistance totalSafety	Path Double Double

Table 1. Entity, attribute and the data type used in the Java coding

4.3 Objective Function

The objective function aims to maximize the overall safety and efficiency of evacuation for the UPM students trapped in The Mines Shopping Mall. This safety is quantified as a combination of the distance traveled and the safety score of the paths taken where distance(p) represents the distance of the path segment p and safetyScore(p) represents the safety score of the path segment p.

Objective Function: Minimize $\sum$ (distance(p) + safetyScore(p))

The goal is to minimize the cumulative sum of the distances and safety scores for the path from the starting location (The Mines Shopping Mall) to the destination (Kolej 12), ensuring the safest and most efficient route for evacuation.

4.4 Constraints

4.4.1 Path Constraints

The paths chosen for evacuation must be currently available and navigable, taking into account the ongoing landslide and any resulting debris or hazards.

4.4.2 Safety Score

The cumulative safety score of the chosen path must be minimized to ensure the highest safety for the evacuees. The formula would be such as shown below.

$$\sum_{p \in paths} distance(p) + \sum_{p \in paths} safetyScore(p) \leq safetyScore(p) \leq safeThreshold$$

4.5 Requirement and Constraint

4.5.1 Objective Requirements

The objective requirement is to maximize overall safety by finding the optimal path from The Mines Shopping Mall to Kolej 12, taking into account both distance and safety scores.

4.5.2 Space Complexity

The space required for storing locations and paths is proportional to the number of locations (vertices) and paths (edges). The formula would be O(V+E) with V = no of locations and E = no of paths.

4.5.3 Time Complexity

For each location, iterate over all connected paths to update the shortest path estimates using a priority queue. The time complexity would be O((V+E)logV) with V = no of locations and E = no of paths.

4.5.4 Value Constraints

The value constraints is that all distances and safety scores must be positive to ensure correct path calculations. Meaning, each location must have valid paths to other reachable locations to ensure a valid path can be found from the starting location to the destination.

4.6 Illustration Problem

Figure 1 below shows the paths that can be taken from The Mines Shopping Mall to Kolej 12 for ten of Universiti Putra Malaysia (UPM) students to reach to their college. The value in black colour represents the distance in kilometres while the value in pink respresents the safety score of the paths itself with one being the most safe and ten being the least safe.

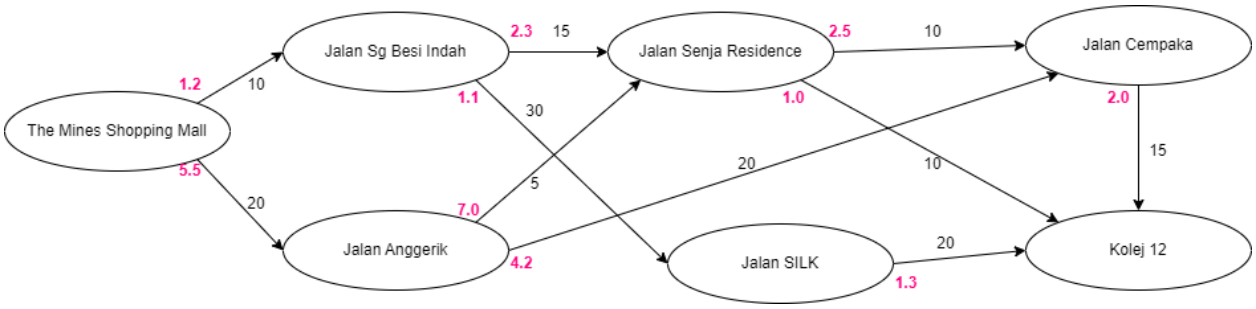

Figure 1. Distance and safety score from The Mines Shopping Mall to Kolej 12

4.7 Algorithm Paradigm

Greedy algorithms build up a solution piece by piece, always choosing the next piece that offers the most immediate benefit. In the context of pathfinding, this means always expanding the shortest known distance first. Dijkstra's algorithm is an example of a greedy algorithm because it makes a series of locally optimal (greedy) choices with the hope of finding the global optimum, which in this case is the shortest path from the start node to the destination node.

4.8 Pseudocode

```
class Location {
    name: String
    paths: List of Path
    minDistance: Double (initially set to infinity)
    previous: Location (initially set to null)

    function compareTo(other: Location) -> Integer
        return compare minDistance with other.minDistance
}

class Path {
    target: Location
    distance: Double
```

```
    safetyScore: Double
}

function findSafestPath(start: Location, destination: Location)
    start.minDistance = 0
    queue = new PriorityQueue()
    queue.add(start)

    while queue is not empty
        current = queue.poll()

        for each path in current.paths
            next = path.target
            weight = path.distance + path.safetyScore
            distanceThroughCurrent = current.minDistance + weight

            if distanceThroughCurrent < next.minDistance
                queue.remove(next)
                next.minDistance = distanceThroughCurrent
                next.previous = current
                queue.add(next)

    printPaths(destination)

function printPaths(destination: Location)
    path = getPathTo(destination)
    print "Safest Path:"
    print "=========================="
    printPathDetails(path)

function getPathTo(target: Location) -> List of Location
    path = new List()
    location = target
    while location is not null
        path.add(location)
        location = location.previous
    reverse(path)
    return path

function printPathDetails(path: List of Location)
```

```
    totalSafetyScore = 0.0
    totalDistance = 0.0

    prev = null
    for each loc in path
        if prev is not null
            pathToLoc = findPath(prev, loc)
            totalSafetyScore += pathToLoc.safetyScore
            totalDistance += pathToLoc.distance
            print "From " + prev.name + " to " + loc.name +
                  " | Distance: " + pathToLoc.distance +
                  " | Safety Score: " + pathToLoc.safetyScore
        prev = loc

    print "Total Distance: " + totalDistance
    print "Total Safety Score: " + totalSafetyScore

function findPath(from: Location, to: Location) -> Path
    for each path in from.paths
        if path.target equals to
            return path

function main()
    mall = new Location("The Mines Shopping Mall")
    kolej12 = new Location("Kolej 12")
    checkpoint1 = new Location("Jalan Sg Besi Indah")
    checkpoint2 = new Location("Jalan Anggrerik")
    checkpoint3 = new Location("Jalan Senja Residence")
    checkpoint4 = new Location("Jalan SILK")
    checkpoint5 = new Location("Jalan Cempaka")

    addPaths(mall, [
        new Path(checkpoint1, 10, 1.2),
        new Path(checkpoint2, 20, 5.5)
    ])
    addPaths(checkpoint1, [
        new Path(checkpoint3, 15, 2.3),
        new Path(checkpoint4, 30, 1.1)
    ])
    addPaths(checkpoint2, [
```

```
    new Path(checkpoint3, 5, 7.0),
    new Path(checkpoint5, 25, 4.2)
  ])
  addPaths(checkpoint3, [
    new Path(kolej12, 10, 1.0),
    new Path(checkpoint5, 10, 2.5)
  ])
  addPaths(checkpoint4, [
    new Path(kolej12, 20, 1.3)
  ])
  addPaths(checkpoint5, [
    new Path(kolej12, 15, 2.0)
  ])

  findSafestPath(mall, kolej12)

function addPaths(location: Location, paths: List of Path)
  for each path in paths
    location.paths.add(path)
```

5.0 Algorithm Specification

The chosen Dijkstra's algorithm is particularly suitable for finding the shortest and safest path in a graph-like structure of locations and paths. Dijkstra's algorithm efficiently manages pathfinding by continuously selecting the path that offers the least cumulative distance and safety score. This greedy algorithm ensures both time and space efficiency, crucial in emergency situations where quick and safe evacuation routes are necessary.

5.1 Input and Output

The input for the coding **is** a set of location objects, each with a name and a list of paths to neighboring locations and each path has a target location, a distance and a safety score (a measure of safety, with lower values indicating safer paths)

Meanwhile, the output is the safest path from the starting location (The Mines Shopping Mall) to the destination (Kolej 12). The output includes the sequence of locations in the safest path, the total distance of the safest path and the total safety score of the safest path.

5.2 Recurrence Relation

The recurrence relation for the safest path using Dijkstra's algorithm is used to update the shortest path estimate from the starting location to each other location. This is done by considering both the distance and safety score of the paths.

For a given current location `u` and a neighboring location `v` connected by a path with distance `d` and safety score `s`, the total distance through `u` to `v` is calculated as: total_distance(v)=minDistance(u)+d+s

If total_distance(v) is less than the current minDistance of v, then we update minDistance(v) and set previous(v) to u: minDistance(v)=total_distance(v) and previous(v)=u

The equation components will be explain more below.

5.2.1 total_distance(v)

This represents the cumulative distance to reach location v from the starting location via the current location u. It includes the minDistance to reach u, plus the distance d and safety score s of the path from u to v.

5.2.2 minDistance(u)

This is the minimum cumulative distance from the starting location to the current location u that has been determined so far. Initially, \text{minDistance} for the start location is set to 0, and for all other locations, it is set to infinity.

5.2.3 d

The distance of the path from location u to location v. This is a fixed value that represents the physical length or cost associated with traveling from u to v.

5.2.4 s

The safety score of the path from location u to location v. This score is added to the distance to account for the safety of the path, with lower values indicating safer paths.

By incorporating the safety score, the algorithm balances between the shortest path and the safest path.

5.2.5 minDistance(v)

This is the minimum cumulative distance from the starting location to the neighboring location v that has been determined so far. If total_distance(v) is found to be less than the current minDistance(v), it indicates that a safer and/or shorter path to v has been found via u.

5.2.6 previous(v)

This is a reference to the previous location on the optimal path to v. It is updated to u if a new shorter or safer path to v is found through u. This reference helps in reconstructing the path from the destination back to the start location once the algorithm completes.

6.0 Program Java Language

```
import java.util.*;

class Location implements Comparable<Location> {
   String name; // Name of the location
   ArrayList<Path> paths; // List of paths leading to other locations
   double minDistance = Double.POSITIVE_INFINITY; // Minimum distance to this
location
   Location previous; // Previous location on the path

   Location(String name) {
      this.name = name;
      this.paths = new ArrayList<>();
   }

   @Override
   public String toString() {
      return name;
```

```
    }

    @Override
    public int compareTo(Location other) {
        return Double.compare(minDistance, other.minDistance);
    }
}

class Path {
    Location target; // Destination location of this path
    double distance; // Distance of this path
    double safetyScore; // Safety score of this path

    Path(Location target, double distance, double safetyScore) {
        this.target = target;
        this.distance = distance;
        this.safetyScore = safetyScore;
    }
}

public class SafePathFinder {

    static void findSafestPath(Location start, Location destination) {
        // Initialize the start location's minimum distance to 0
        start.minDistance = 0;

        // Priority queue to process locations based on their minimum distance
        PriorityQueue<Location> queue = new PriorityQueue<>();
        queue.add(start);

        // Process the queue until it's empty
        while (!queue.isEmpty()) {
            Location current = queue.poll(); // Get the location with the smallest minDistance

            // Iterate through each path from the current location
            for (Path path : current.paths) {
                Location next = path.target; // Target location of the current path
                double weight = path.distance + path.safetyScore; // Total weight (distance +
safety score)
```

```
            double distanceThroughCurrent = current.minDistance + weight; // Cumulative distance through current location

            // If a shorter path to 'next' is found, update its minDistance and previous location
            if (distanceThroughCurrent < next.minDistance) {
                queue.remove(next);
                next.minDistance = distanceThroughCurrent;
                next.previous = current;
                queue.add(next);
            }
        }
    }

    // Print the details of the path from start to destination
    printPaths(destination);
  }

  static void printPaths(Location destination) {
    List<Location> path = getPathTo(destination); // Get the path to the destination

    System.out.println("Safest Path:");
    System.out.println("==========================");
    printPathDetails(path); // Print the path details
  }

  static List<Location> getPathTo(Location target) {
    List<Location> path = new ArrayList<>(); // List to store the path locations
    for (Location location = target; location != null; location = location.previous) {
      path.add(location); // Add each location to the path list
    }
    Collections.reverse(path); // Reverse the list to get the correct order from start to destination
    return path;
  }

  static void printPathDetails(List<Location> path) {
    double totalSafetyScore = 0.0; // Total safety score of the path
    double totalDistance = 0.0; // Total distance of the path
```

```
        Location prev = null;
        for (Location loc : path) {
            if (prev != null) {
                Path pathToLoc = null;
                // Find the path from prev to loc
                for (Path p : prev.paths) {
                    if (p.target.equals(loc)) {
                        pathToLoc = p;
                        break;
                    }
                }

                if (pathToLoc != null) {
                    totalSafetyScore += pathToLoc.safetyScore;
                    totalDistance += pathToLoc.distance;
                    // Print the segment details
                    System.out.println(String.format("From %s to %s | Distance: %.2f | Safety
Score: %.2f",
                            prev.name, loc.name, pathToLoc.distance, pathToLoc.safetyScore));
                }
            }
            prev = loc;
        }

        // Print the total distance and safety score
        System.out.println(String.format("\nTotal Distance: %.2f", totalDistance));
        System.out.println(String.format("Total Safety Score: %.2f", totalSafetyScore));
    }

    public static void main(String[] args) {
        // Create instances of locations
        Location mall = new Location("The Mines Shopping Mall");
        Location kolej12 = new Location("Kolej 12");
        Location checkpoint1 = new Location("Jalan Sg Besi Indah");
        Location checkpoint2 = new Location("Jalan Anggrerik");
        Location checkpoint3 = new Location("Jalan Senja Residence");
        Location checkpoint4 = new Location("Jalan SILK");
        Location checkpoint5 = new Location("Jalan Cempaka");

        // Add paths to each location
```

```
        mall.paths.addAll(Arrays.asList(
            new Path(checkpoint1, 10, 1.2),
            new Path(checkpoint2, 20, 5.5)
        ));
        checkpoint1.paths.addAll(Arrays.asList(
            new Path(checkpoint3, 15, 2.3),
            new Path(checkpoint4, 30, 1.1)
        ));
        checkpoint2.paths.addAll(Arrays.asList(
            new Path(checkpoint3, 5, 7.0),
            new Path(checkpoint5, 25, 4.2)
        ));
        checkpoint3.paths.addAll(Arrays.asList(
            new Path(kolej12, 10, 1.0),
            new Path(checkpoint5, 10, 2.5)
        ));
        checkpoint4.paths.addAll(Arrays.asList(
            new Path(kolej12, 20, 1.3)
        ));
        checkpoint5.paths.addAll(Arrays.asList(
            new Path(kolej12, 15, 2.0)
        ));

        // Find the safest path from the mall to kolej12
        findSafestPath(mall, kolej12);
    }
}
```

7.0 Algorithm's Correctness Analysis With Time Complexity

7.1 Correctness Analysis

The correctness of the Dijkstra's algorithm used to find the safest path in the given scenario can be analyzed through three main parts: Initialization, Recurrence Relation, and Optimization Function

7.1.1 Initialization

The algorithm initializes by setting the minimum distance to the start location (The Mines Shopping Mall) to 0 and all other locations to infinity. This ensures that the starting point is the source of the pathfinding and all other locations are initially unreachable.

The initialization steps would be to **s**et start.minDistance = 0 and Initialize a priority queue and add the start location to it. This initialization correctly sets up the initial conditions for the algorithm, ensuring that the search begins from the start location with the appropriate initial distance.

7.1.2 Recurrence Relation

The recurrence relation in Dijkstra's algorithm updates the shortest path estimate for each location based on the current shortest known paths. For each location u (current location), examine each neighboring location v (target location through path p) and update its minimum distance if a shorter path is found.

total_distance(v)=minDistance(u)+distance(p)+safetyScore(p)

If total_distance(v) is less than minDistance(v), then: minDistance(v)=total_distance(v)

previous(v)=u

This step ensures that the shortest known path to each location is continually improved. By iterating over all paths from the current location and updating the shortest paths, the algorithm guarantees that the minimum distance and safest path are accurately found.

7.1.3 Optimization Function

For each location u, the algorithm updates the minDistance to each neighboring location v through the path weight w:

minDistance(v)=min(minDistance(v),minDistance(u)+w)

The optimization ensures that each location's shortest path is finalized once it is processed. By consistently expanding the location with the smallest minDistance, Dijkstra's algorithm ensures that each location's shortest path is accurately determined, providing the optimal route from The Mines Shopping Mall to Kolej 12.

7.2 Time Complexity Analysis

7.2.1 Best Case Complexity

The best-case scenario occurs when each location has direct access to the start location (The Mines Shopping Mall), minimizing the number of priority queue operations required. The time complexity is O((V+E)logV). This efficiency arises because the priority queue operations are logarithmic relative to the number of locations (V), making it optimal for scenarios where paths are straightforward and direct. When each location has a direct path to the start location, the algorithm quickly determines the shortest paths without extensive exploration. The priority queue efficiently manages path expansions, ensuring that each location is processed in optimal order to maintain the shortest path property.

7.2.2 Average Case Complexity

In typical scenarios, locations may have varying degrees of connectivity and distance from the start location (The Mines Shopping Mall). The average case complexity considers a mix of direct and indirect paths. The time complexity is O((V+E)logV)

The average-case time complexity of Dijkstra's algorithm remains O((V+E)logV). This ensures consistent performance across scenarios where paths vary in length and complexity. Despite the variation in path lengths and connectivity, the algorithm's priority queue-based approach ensures that paths are explored efficiently. Each location is processed based on its current shortest known distance, optimizing the discovery of shortest and safest routes.

7.2.3 Worst Case Complexity

The worst-case scenario involves densely interconnected locations and paths, requiring extensive exploration and queue operations. The time complexity is O((V+E)logV).

Even in the worst case, where the graph has maximum connectivity and paths, Dijkstra's algorithm maintains a time complexity of O((V+E)logV). This worst-case scenario ensures that the algorithm's efficiency in pathfinding is preserved across challenging network structures. The algorithm's worst-case performance is bounded by its ability to manage priority queue operations efficiently. Despite increased path complexities and connectivity, Dijkstra's algorithm guarantees optimal path determination by systematically expanding the shortest known paths from the start location (The Mines Shopping Mall).

7.3 Space Complexity

The space complexity of Dijkstra's algorithm is O(V+E), where V represents the number of locations and E represents the number of paths in the graph. The algorithm requires storage for each location and path in the graph, ensuring that distances and paths are accurately maintained during pathfinding. Additionally, the priority queue requires space proportional to the number of locations (V), managing locations based on their current shortest distances.

In conclusion, the Dijkstra's algorithm effectively finds the safest evacuation route during a landslide at The Mines Shopping Mall. By minimizing both distance and safety risks, the algorithm provides real-time, optimal pathfinding even in dynamic conditions. Its consistent time complexity, O((V+E)log V), ensures efficient performance, making it a reliable and robust solution for emergency evacuations.