

CSC4202

STRATEGIC RELIEF SUPPLY ALLOCATION IN DISASTER AREAS



PRESENTED BY
MOHAMMAD FARHAN (211483)
AFIQ FIKRI BIN MOHD ZAHIR (211323)
MUHAMMAD ZIKRI BIN ZAKARIA (213118)

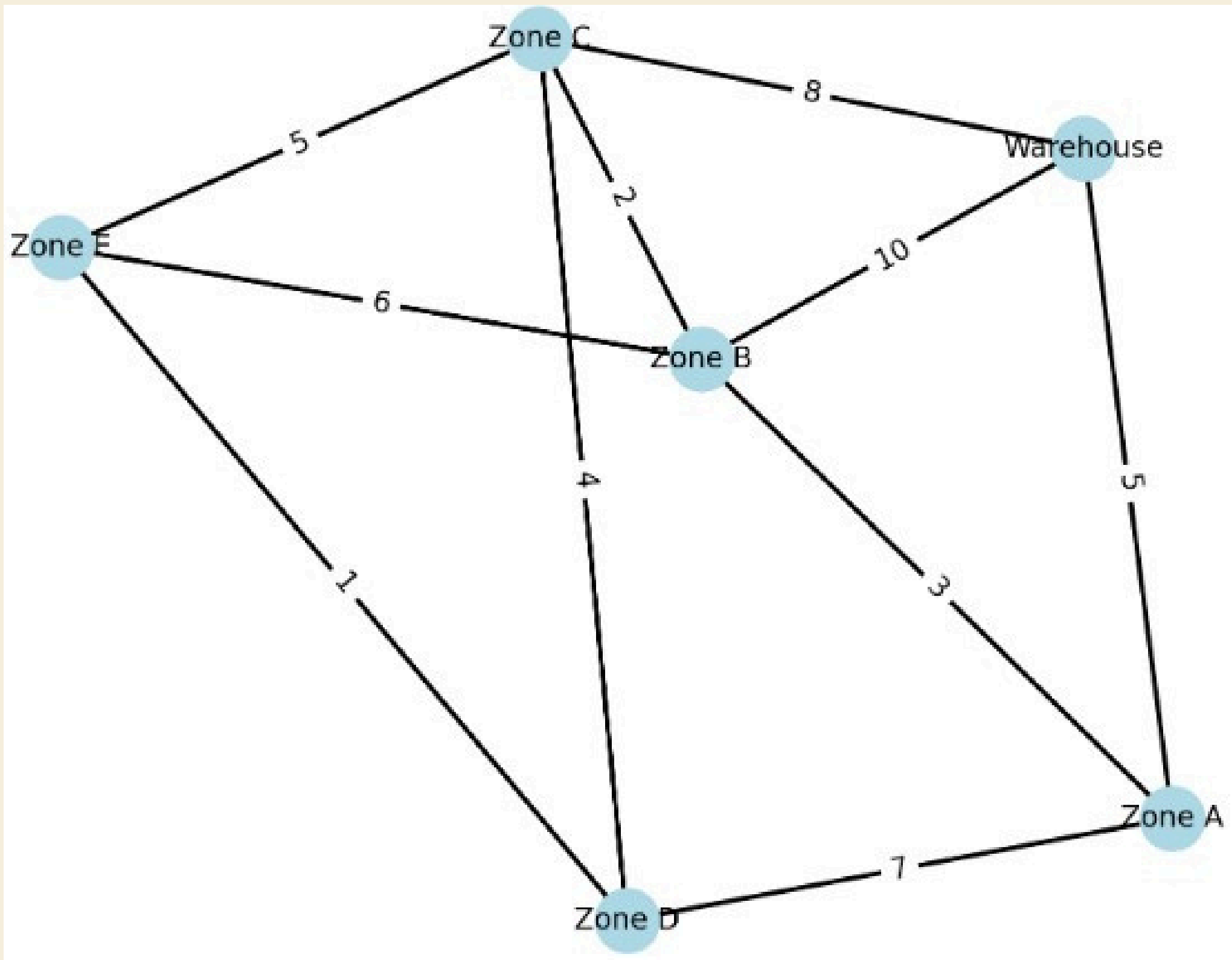


SCENARIO DESCRIPTION

In the aftermath of a devastating earthquake in a city where the government needs to efficiently distribute relief supplies to various affected zones. The city is divided into several zones, where each zone need for supplies (water, food, medical kits, etc.).

The relief supplies are stored in a central warehouse and need to be delivered quickly. The goal is to find the shortest route to minimize the total delivery time while ensuring that all zones receive the necessary supplies.

MAP AND DELIVERY ROUTES



Nodes represent the central warehouse and the zones (Zone A, Zone B, Zone C, Zone D, Zone E).

Edges represent the possible delivery routes between these points, with weights indicating the delivery times in minutes.

IMPORTANCE OF FINDING AN OPTIMAL SOLUTION

1. TIMELINESS



Efficient distribution ensures that relief reaches the affected zones quickly, which is critical for saving lives and reducing suffering.


2. RESOURCE MANAGEMENT



Optimal use of available resources (trucks, supplies) ensures that no zone is left without aid while avoiding wastage or overstocking in any area.


IMPORTANCE OF FINDING AN OPTIMAL SOLUTION

3. COST EFFICIENCY



Minimizing delivery times and optimizing routes can significantly reduce fuel and operational costs, allowing more resources to be allocated to relief efforts.

4. COORDINATION



An optimal solution ensures better coordination among relief teams, avoiding duplication of efforts and ensuring comprehensive coverage of all affected zones.

MODEL DEVELOPMENT

DATA TYPE

- **Graph Representation**
- **Zones**
- **Edge**

Example:

(Warehouse, A, 5),
meaning 5 units of
time to travel from
the Warehouse to
zone A.

OBJECTIVE FUNCTION

Minimize Total Delivery Time:

Minimize the sum
of the weights of
the edges in the
MST to ensure all
zones are
connected with
minimal total
travel time.

CONSTRAIN

Connectivity

**Non-negative
Weights**

**Graph is
Connected**

Algorithm Comparison

ALGORITHM	SUITABILITY	STRENGTHS	WEAKNESSES
Greedy Algorithms	Very High	<ul style="list-style-type: none">• Simple and easy to implement• Efficient in terms of time and space• Directly applicable to pathfinding and shortest path problems	Non-optimal for some problems as it focus on local optimization.
Dynamic Programming (DP)	Moderate	Suitable for problems with multiple optimal sub paths	Implementation can be complex
Divide and Conquer (DAC)	Low to Moderate	Useful for breaking down complex problems	May not handle interconnected paths efficiently
Sorting	Low	Simple and well understood	Does not address the pathfinding problem

ALGORITHM SPECIFICATION

Prim's Algorithm (Greedy Algorithm) is chosen because it is well-suited for finding the Minimum Spanning Tree (MST) in a connected, undirected graph, ensuring that all zones are connected with the minimum total edge weight.



Simplicity

Greedy algorithms are typically easy to understand and implement.

Optimal for Certain Problems

For specific problems, such as minimum spanning tree or shortest path (Dijkstra's algorithm), greedy algorithms provide optimal solutions.

STRENGTHS OF GREEDY

Efficiency

Usually run in polynomial time, making it suitable for large datasets.

Local Optimization

Make a series of choices, each of which looks best at the moment, often leading to a good solution quickly.

PSEUDOCODE

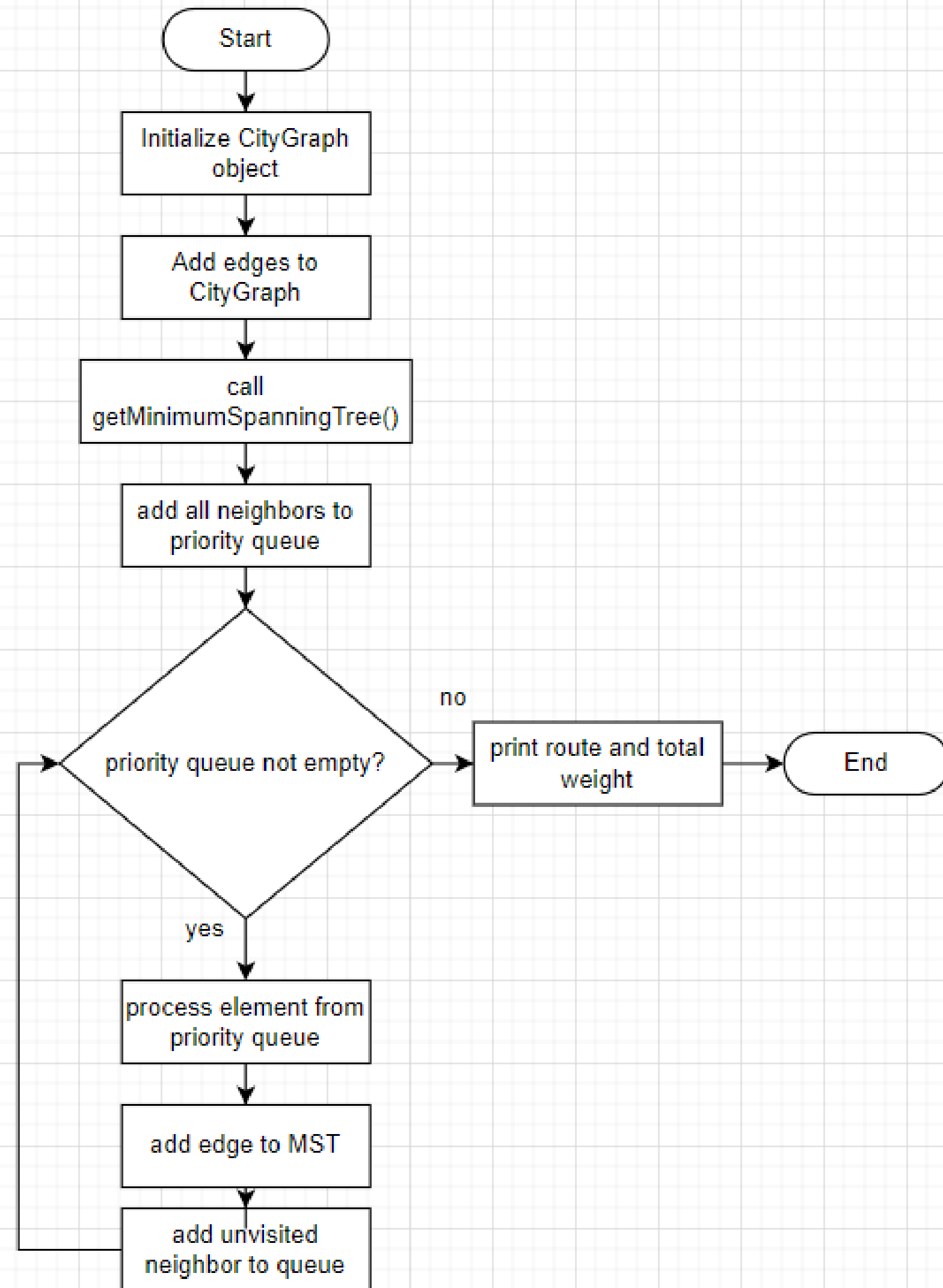
Initialization:

1. Start with an arbitrary node (in this case, the "Warehouse").
2. Initialize an empty list to store the Minimum Spanning Tree (MST) edges.
3. Use a priority queue (min-heap) to keep track of the edges with the smallest weights.
4. Maintain a set of visited nodes to avoid cycles.

Iterative Process:

1. Continuously select the edge with the smallest weight from the priority queue.
2. If the destination node of this edge has not been visited, add it to the MST and mark it as visited.
3. Add all edges from the newly visited node to the priority queue, provided the destination nodes have not been visited.
4. Repeat this process until all nodes are visited and the MST is complete.

FLOWCHART



JAVA CODE

```
import java.util.*;

public class ReliefDistribution {

    Run | Debug
    public static void main(String[] args) {
        CityGraph cityGraph = new CityGraph();
        cityGraph.addEdge(source:"Warehouse", destination:"A", weight:5);
        cityGraph.addEdge(source:"Warehouse", destination:"B", weight:10);
        cityGraph.addEdge(source:"Warehouse", destination:"C", weight:8);
        cityGraph.addEdge(source:"A", destination:"B", weight:3);
        cityGraph.addEdge(source:"B", destination:"C", weight:2);
        cityGraph.addEdge(source:"C", destination:"D", weight:4);
        cityGraph.addEdge(source:"C", destination:"E", weight:5);
        cityGraph.addEdge(source:"D", destination:"E", weight:1);

        String startNode = "Warehouse";
        List<Route> mst = getMinimumSpanningTree(cityGraph, startNode);

        System.out.println(x:"Minimum Spanning Tree (MST) edges:");
        for (Route route : mst) {
            System.out.println(route.source + " - " + route.destination + " with weight " + route.weight);
        }

        int totalWeight = mst.stream().mapToInt(route -> route.weight).sum();
        System.out.println("Total distance (time) to supply all zones: " + totalWeight);
    }

    private static List<Route> getMinimumSpanningTree(CityGraph cityGraph, String start) {
        List<Route> mst = new ArrayList<>();
        Set<String> visited = new HashSet<>();
        PriorityQueue<Route> priorityQueue = new PriorityQueue<>(Comparator.comparingInt(e -> e.weight));

        visited.add(start);
        priorityQueue.addAll(cityGraph.getNeighbors(start));
    }
}
```

```

        while (!priorityQueue.isEmpty()) {
            Route route = priorityQueue.poll();
            if (visited.contains(route.destination)) {
                continue;
            }
            visited.add(route.destination);
            mst.add(route);

            for (Route neighbor : cityGraph.getNeighbors(route.destination)) {
                if (!visited.contains(neighbor.destination)) {
                    priorityQueue.add(neighbor);
                }
            }
        }

        return mst;
    }
}

```

```

class CityGraph {
    private final Map<String, List<Route>> adjList = new HashMap<>();

    public void addEdge(String source, String destination, int weight) {
        adjList.putIfAbsent(source, new ArrayList<>());
        adjList.putIfAbsent(destination, new ArrayList<>());
        adjList.get(source).add(new Route(source, destination, weight));
        adjList.get(destination).add(new Route(destination, source, weight));
    }

    public List<Route> getNeighbors(String zone) {
        return adjList.get(zone);
    }

    public Set<String> getZones() {
        return adjList.keySet();
    }
}

```

```

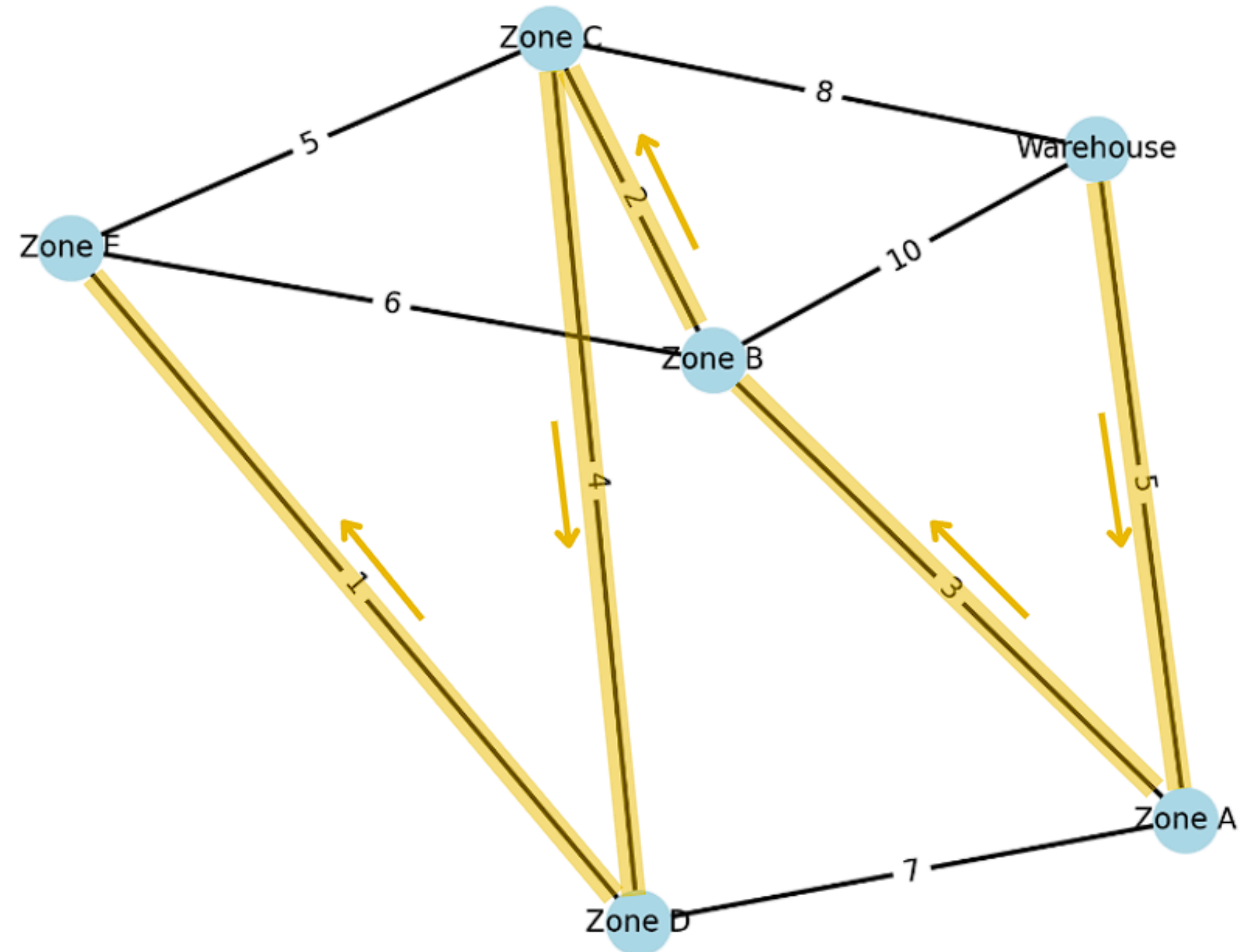
class Route {
    String source;
    String destination;
    int weight;

    Route(String source, String destination, int weight) {
        this.source = source;
        this.destination = destination;
        this.weight = weight;
    }
}

```

Output & Minimum Spanning Tree

```
Minimum Spanning Tree (MST) edges:  
Warehouse - A with weight 5  
A - B with weight 3  
B - C with weight 2  
C - D with weight 4  
D - E with weight 1  
Total distance (time) to supply all zones: 15
```



TIME COMPLEXITY

BEST CASE



$$O(E \log V)$$

The graph is already a MST. Each edge added to the MST is the smallest among all available edges. The algorithm selects the minimum-weight edge at each step, and the priority queue operations are optimized.

AVERAGE



$$O((V + E) \log V)$$

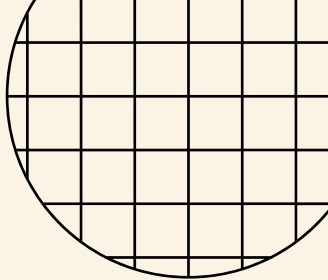
The edges of the graph are randomly distributed. Each vertex has a constant number of edges adjacent to it. Priority queue operations and updates may vary, leading to an average logarithmic time complexity for each operation.

WORST CASE



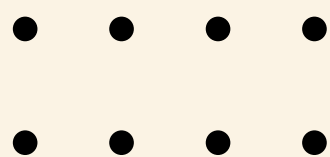
$$O((V + E) \log V)$$

The graph is densely connected, and each edge added to the MST results in multiple updates in the priority queue.



Conclusion

In conclusion, Prim's algorithm is an ideal choice for efficiently distributing relief supplies in a post-earthquake scenario. By constructing the MST, it connects all zones with minimal delivery times, addressing critical needs such as timeliness, resource management, and cost efficiency. The algorithm's simplicity and effectiveness ensure optimal relief distribution, making it highly suitable for disaster response logistics. Its manageable time complexity further confirms its practical application for large-scale operations, providing a robust solution for coordinating and optimizing relief efforts.



THANK YOU !

UNIVERSITI PUTRA MALAYSIA