Project Report

On

OPTIMIZATION OF EMERGENCY RESPONSE LOCATIONS USING LOCATION ALLOCATION MODELS

Submitted

in partial fulfilment of the requirements for the degree of

BACHELOR OF SCIENCE

in

MATHEMATICS

By

LEIRA JAISON (AB23AMAT018)
Under the Supervision of

Ms. ANNA SHEETHAL V S



DEPARTMENT OF MATHEMATICS AND STATISTICS

ST. TERESA'S COLLEGE (AUTONOMOUS)

ERNAKULAM, KOCHI - 682011

MARCH 2026

ST. TERESA'S COLLEGE (AUTONOMOUS), ERNAKULAM

CERTIFICATE

This is to certify that the dissertation entitled, **OPTIMIZATION OF EMERGENCY RESPONSE LOCATIONS USING LOCATION ALLOCATION MODELS** is a bonafide record of the work done by **LEIRA JAISON** under my guidance as partial fulfillment of the award of the degree of **Bachelor of Science in Mathematics** at St. Teresa's College (Autonomous), Ernakulam affiliated to Mahatma Gandhi University, Kottayam. No part of this work has been submitted for any other degree elsewhere.

Date: 3/10/2025
Place: Ernakulam

Ms. Anna Sheethal V S
Assistant Professor,
Department of Mathematics and Statistics,
St.Teresa's college (Autonomous),
Ernakulam

ST. TERESA'S COLLEGE (AUTONOMOUS) DEPARTMENT OF MATHEMATICS AND STATISTICS ERNAKULAM

Dr. Elizabeth Reshma M T

Assistant Professor and Head,
Department of Mathematics and Statistics,
St.Teresa's college (Autonomous),
Ernakulam

20/01/26

External Examiners

1: ..... **2: …….....**

Dr. SREEJA K.U. (PEN:716628)
Assistant Professor
Department of Mathematics
Maharaja's College
Ernakulam - 682 011

DECLARATION

I hereby declare that the work presented in the project is based on the original work done by me under the guidance of Ms. Anna Sheethal V S, Assistant Professor, Department of Mathematics and Statistics, St. Teresa's College (Autonomous), Ernakulam and has not been included in any other project submitted previously for the award of any degree.

Place: Ernakulam.

Date: 3/10/2025

LEIRA JAISON (AB23AMAT018)

ACKNOWLEDGEMENT

I am profoundly thankful to my mentor and supervisor, Ms. Anna Sheethal V S for her unwavering guidance, insightful suggestions, and constant encouragement. I extend my thanks to the faculty members of the Department of Mathematics and Statistics for their suggestions and encouragement during the preparation of this work. I am also immensely grateful to my peers and colleagues for their constructive feedback, collaboration, and thought-provoking discussions. This has been a remarkable learning experience, and I am humbled and grateful for the opportunity to grow both academically and personally through this journey.

Place: Ernakulam.

Date: 3/10/2025

LEIRA JAISON (AB23AMAT018)

CONTENTS

CERTIFICATE……..……...ii

DECLARATION…….…...iii

ACKNOWLEDGEMENT……..…….…...…iv

CONTENTS……...……...…….....…v

INTRODUCTION……..1

CHAPTER 1: BACKGROUND, MOTIVATION, AND STUDY FRAMEWORK

1.1 Background and Motivation…….……..……..…..2

1.2 Project objectives

1.2.1 Optimize Resource Allocation using Linear Programming…..……...……3

1.2.2 Optimization of Routes using Graph Theory…….....5

1.3 Scope and Limitations

1.3.1 Scope and Limitations of SCLP, MCLP, and MILP for Resource Allocation…...6

1.3.2 Scope and Limitations of using Graph Theory for Routes Optimization:

Dijkstra's Algorithm …….....7

CHAPTER 2: RESOURCE ALLOCATION USING LINEAR PROGRAMMING

2.1 The Set Covering Location Problem (SCLP)…….....9

2.2 Maximum Covering Location Problem (MCLP)…….....……..13

2.3 Mixed-Integer Linear Programming (MILP)……...……18

CHAPTER 3: ROUTE OPTIMIZATION USING GRAPH THEORY

3.1 Graph Theory and Road Network……..……23

3.2 Shortest Path Algorithms

3.2.1 Dijkstra's Algorithm……..……24

3.2.2 Bellman Ford Algorithm……...…….…..35

3.2.3 A* Search Algorithm…….....43

3.2.4 Floyd–Warshall Algorithm……...…….....47

CHAPTER 4: **CASE STUDY**

4.1 Resource Allocation using Linear Programming

4.1.1 The Set Covering Location Problem (SCLP)……..……..…..54

4.1.2 Maximum Covering Location Problem (MCLP)…….....56

4.1.3 Mixed-Integer Linear Programming (MILP)…….....59

4.2: Route Optimization using Graph Theory

4.2.1 Finding Shortest Distance Path using Dijkstra's Algorithm……..…62

4.2.2 Finding Shortest Time Path using Dijkstra's Algorithm……....67

CONCLUSION.....71

REFERENCE……..……72

INTRODUCTION

The effectiveness of emergency response systems—spanning fire, medical, and rescue services—is fundamentally dependent on the strategic placement of resources and the operational efficiency of their deployment. In time-sensitive crises, delays measured in mere minutes can translate directly into catastrophic outcomes, underscoring the necessity for optimized planning. This project addresses this crucial challenge by employing a rigorous mathematical optimization framework to enhance the logistics of emergency preparedness and response.

The study is structured around the two core challenges of emergency management: strategic location and dynamic routing.

First, the strategic phase utilizes **linear programming (LP)** by analyzing and implementing three key Location Allocation Models: the **Set Covering Location Problem (SCLP)**, the **Maximal Covering Location Problem (MCLP)**, and the **Mixed-Integer Linear Program (MILP)**. These models are applied to determine the optimal, fixed positions for emergency facilities to maximize service coverage and minimize base-to-demand distance under resource constraints.

Second, the dynamic phase focuses on efficient real-time travel using **Graph Theory**. Road networks are modeled as weighted graphs, where the **shortest path** is identified primarily through **Dijkstra's Algorithm**. The operational suitability and performance of Dijkstra's method are then benchmarked against comparison algorithms, including the $A*$ Algorithm, Bellman-Ford, and Floyd-Warshall, to confirm the most efficient routing strategy.

This thesis is organized into four main chapters. **Chapter 1** provides the foundational context, motivation and project objectives. **Chapter 2** presents the theoretical framework and application of the LP Location Allocation Models. **Chapter 3** details the graph theory approach and the comparative analysis of the shortest path algorithms. Finally, **Chapter 4** is case study.

CHAPTER: 1

BACKGROUND, MOTIVATION, AND STUDY FRAMEWORK

1.1 BACKGROUND AND MOTIVATION

Emergency response services are paramount in the vital task of saving lives, reducing injuries, and lessening damage during unexpected emergencies. The importance of a quick response cannot be overemphasized since delay can result in disastrous effects that impact the individuals and communities as a whole. This section highlights the importance of timely emergency response and the challenges associated with it.

EFFECT OF TIMELY REACTIONS EMERGENCY SITUATIONS

Timely reactions in emergency situations have several critical benefits.

• Boosts survival chances: Quick responses in medical emergencies significantly boost survival chances as timely interventions can be the deciding factor between life and death.

• Reducing injury severity: Immediate access to medical treatment not only eases the severity of injuries but also minimizes the risk of long-term health consequences that may result from untreated conditions.

• Limiting property damage: swift action effectively limits property damage reducing financial losses and preventing further harm to community infrastructure.

• Building community resilience: Effective and coordinated responses promote faster recovery periods, enabling communities to recover more speedily from crises with improved stability and coherence.

CONSEQUENCES OF DELAYED RESPONSE

Delayed emergency responses can have severe consequences. Such delays may lead to increased mortality rates as it may result in higher death rates, as precious seconds are wasted when every second matters. Longer response times can also worsen health outcomes, leading to higher illness rates and added strain on healthcare facilities. Furthermore, delayed responses can result in more widespread property damage, leading to significant long-term economic

costs. Ultimately, slow emergency responses can erode the resilience of a community, making it more vulnerable to future crises and undermining confidence in emergency services.

BARRIERS AND CALL FOR IMPROVEMENT

Response teams face several barriers that can impact their effectiveness. Resource constraints, including limitations in staff, equipment, and finance, can compromise the efficiency of emergency responses. Ineffective distribution of resources can lead to severe coverage gaps, leaving some areas without adequate support during emergencies. Additionally, the unpredictability of emergency scenarios creates special difficulties, necessitating dynamic and creative solutions to respond effectively. Improving emergency response systems requires attention to strategies that can speed up response times, mitigate the effect of emergencies, and enhance community resilience. This can be done by:

• Strategic Locating of Facilities: Optimizing the location of emergency response facilities to reduce response times and improve coverage.

• Optimized Distribution of Resources: Resource allocation to maximize efficiency can increase the overall effectiveness of emergency services, making sure they are adequately ready to respond fast and efficiently.

• The Integrating Modern Technology can enhance communication, enhance data analysis, and enable faster decision-making in times of emergency.

By optimizing and upgrading emergency response systems, communities can not only save lives and mitigate damage but also develop a culture of resilience that equips them to face future adversity with confidence and strength.

1.2 PROJECT OBJECTIVES

1.2.1 OPTIMIZE RESOURCE ALLOCATION USING LINEAR PROGRAMMING

Introduction to Resource Allocation in Linear Programming

Resource allocation is a critical aspect of decision-making in various fields, including business, economics, and engineering. It involves distributing limited resources among activities or projects to achieve a specific objective. Linear programming is a powerful tool used to optimise resource allocation by maximising or minimising a linear objective function subject to a set of linear constraints.

Resource allocation refers to the process of assigning available resources to different tasks or projects to achieve a specific goal.

Our goal is to determine the most effective placement of emergency resources (e.g., ambulances) to maximize coverage and minimize a system's overall response time. This phase employs linear programming models, specifically the Set Covering Location Problem (SCLP), Maximum Covering Location Problem (MCLP), and Mixed-Integer Linear Programming (MILP), to solve the facility location problem independently of any real-time event. The output of this phase will be a fixed, optimal configuration of facilities.

SCLP, MCLP, and MILP are widely used linear programming models for optimizing resource allocation in emergency response—especially for siting facilities and allocating rescue resources in a way that best serves communities during disasters.

SCLP IN EMERGENCY RESPONSE

The Set Covering Location Problem (SCLP) seeks the minimum number of emergency facilities (like shelters, ambulance stations, or fire stations) required so every demand point (e.g., accident sites or neighbourhoods) is covered within a specified service radius.

In emergencies, SCLP helps ensure basic accessibility: that each incident location has at least one rapid response possibility, using as few resources or sites as possible.

MCLP IN EMERGENCY RESPONSE

The Maximal Coverage Location Problem (MCLP) aims to maximize the number of demand points that can be covered by a limited number of facilities or resources.

In situations where not all locations can be fully covered due to limited vehicles or responders, MCLP optimizes placement to serve as much of the affected population as possible, increasing the overall effectiveness of the emergency response.

MILP IN EMERGENCY RESPONSE

Mixed-Integer Linear Programming (MILP) enables more detailed modeling by allowing a mix of integer (binary) and continuous decision variables, which can represent assignments (yes/no decisions), quantities of resources, and allocations across multiple facilities or types of incidents.

MILP is used for multi-objective or multi-resource allocation (e.g., for deploying ambulances, fire trucks, and rescue teams simultaneously), optimizing cost, response time, and resource constraints in a realistic way.

PRACTICAL APPLICATION

These models help emergency managers decide where to place limited resources (sites, vehicles, teams) and how to allocate them during large-scale incidents—including natural disasters and multi-site accidents—balancing cost, coverage, and responsiveness.

They are often implemented using linear programming solvers and can incorporate geographic, temporal, and capacity constraints to represent real-world emergency networks.

Using SCLP, MCLP, and MILP provides a robust, quantitative approach to strategic resource allocation for emergency response, allowing for evidence-based, cost-effective, and rapid deployment planning.

1.2.2 OPTIMIZATION OF ROUTES USING GRAPH THEORY

The objective of optimizing routes using Graph Theory is to determine the most efficient paths for emergency responders to reach emergency locations quickly and safely. This involves finding the shortest or minimum-weight paths in a graph, where nodes represent locations, and edges represent roads or paths between them.

DEFINITION AND EQUATION

In graph theory, the shortest path/Minimum cost path between nodes s (start) and t (target) can be defined as:

$$d(s, t) = \min \sum w(i, j) \times x(i, j) \text{ , where}$$

- d(s, t) represents the shortest distance/cost from node s to node t
- x(i, j) is a binary variable indicating whether edge (i, j) is included in the shortest path
- w(i, j) is the weight of edge (i, j)

While the equation captures the essence of the shortest path problem, in practice, the shortest path can be found using Dijkstra's algorithm, which updates the distance to each node as:

$$d(i) = \min(d(j) + w(j, i)) \text{ for all } j \in V$$

Where d(i) is the current shortest distance from the source node to node i.

To implement the algorithm, programming languages (e.g., Python, Java, or C++) will use to implement the chosen algorithm. Then proceeding to test implementation with sample data and validate the results.

APPLICATION TO EMERGENCY RESPONSE

1. Emergency Vehicle Routing: Determine the most efficient routes for emergency vehicles, such as ambulances or fire trucks, to reach emergency locations.

2. Resource Allocation: Optimize the allocation of emergency resources, such as personnel or equipment, to emergency locations.

3. Evacuation Planning: Determine the most efficient evacuation routes for people in danger, minimizing risk and response time.

BENEFITS

1. Reduced Response Time: Optimized routes can reduce response times, improving outcomes in emergency situations.

2. Increased Efficiency: Optimized routes can reduce fuel consumption, lower emissions, and decrease wear on emergency vehicles.

3. Improved Safety: Optimized routes can minimize risk and reduce the likelihood of accidents or injuries during emergency responses.

1.3 SCOPE AND LIMITATIONS

1.3.1 SCOPE AND LIMITATIONS OF SCLP, MCLP, AND MILP FOR RESOURCE ALLOCATION

SCOPE

- This study demonstrates the application of SCLP, MCLP, and MILP for resource allocation in emergency response using hypothetical problem scenarios.
- The focus is on optimizing locations and allocation for facilities and resources (such as ambulances, shelters) via linear programming techniques in various emergency contexts.
- The models used help illustrate objective-driven allocation—minimizing cost, maximizing coverage, and balancing multiple constraints.
- Scenarios include planning for disaster relief, emergency medical response, and facility placement within controlled, imagined environments.
- Problem size, complexity, and parameters (number of demand points, service radii, resource limits) are defined based on hypothetical, not real-world data, to showcase model capabilities and typical outputs.

LIMITATIONS

- **Generality vs. Realism:** Results are not based on actual field data, so findings only indicate potential applications, not validated effectiveness in real emergencies.
- **Model Assumptions:** Linear programming models simplify real-world complexities—such as variable traffic, time-dependent demand, and infrastructure failures—which may reduce accuracy in actual emergency situations.
- **Data and Computational Limits:** As problem size and the number of constraints grow, MILP computational performance can degrade significantly, making large or highly detailed problems slow or impractical to solve without advanced solvers.
- **Hypothetical Scenarios:** Insights may not generalize; unique challenges in real emergencies (e.g., multi-objective trade-offs, coordination issues) might require modifications or more sophisticated approaches.
- **Excluded Factors:** Some models prioritize a single goal (coverage, cost) and may ignore secondary impacts, robustness, or multi-perspective needs relevant in real crisis resource distribution.
- **Solver Limitations:** Solution quality and computational time may vary widely depending on scenario parameters and solver algorithms, especially for models with "big M" constraints or weak LP relaxations.

1.3.2 SCOPE AND LIMITATIONS OF USING GRAPH THEORY FOR ROUTES OPTIMIZATION: DIJKSTRA'S ALGORITHM

SCOPE

- Graph theory, and specifically Dijkstra's algorithm, is employed to optimize routes for emergency vehicles and responders by identifying the shortest or fastest paths in road networks modelled as graphs.
- The algorithm streamlines emergency logistics by quickly calculating optimal routes from a start to a destination node, thus minimizing travel time and aiding rapid resource allocation or rescue operations.
- It can integrate real-time data (like GPS and live traffic) for dynamic adjustment of paths, improving situational responsiveness and operational safety.
- Dijkstra's algorithm provides a foundation for intelligent command centers and decision support systems in emergency management—facilitating coordination, resource deployment, and adaptation to evolving scenarios.

LIMITATIONS

- **Static Environment Assumption:** Dijkstra's algorithm traditionally operates on a static graph; sudden changes, such as real-time road closures or extreme congestion,

require frequent recalculation or alternative algorithms better suited for dynamic routing.

- **Single Objective:** It finds the shortest path by distance or cost, but may not account for multi-objective needs (e.g., risk minimization, multiple vehicle types, or resource constraints) unless extensively modified.
- **Computational Complexity:** For very large networks, repeated real-time calculation can be resource-intensive and slow, potentially impacting immediate decision-making when many routes need optimization quickly.
- **Simplifications:** The method relies on accurate, complete data—missing or incorrect information about road conditions, disaster impacts, or travel speeds can reduce solution reliability.
- **Lack of Predictive Adaptation:** Without integration with traffic prediction or learning models, Dijkstra's algorithm cannot anticipate future changes, limiting its applicability for proactive planning in evolving emergencies.

Finding the shortest path in a graph is a fundamental problem with numerous applications in fields like networking, transportation, and logistics. Dijkstra's algorithm is a popular choice for solving this problem, but it has limitations, particularly when dealing with negative weights or requiring all-pairs shortest paths. In this discussion, we'll also explore other algorithms like Bellman-Ford, Floyd-Warshall, and A* that can be used in specific scenarios. We'll delve deeper into these alternative algorithms in the upcoming chapter, examining their applications and trade-offs.

CHAPTER: 2

RESOURCE ALLOCATION USING LINEAR PROGRAMMING

2.1 THE SET COVERING LOCATION PROBLEM (SCLP)

In many public service and emergency response settings, resources (such as fire stations or hospitals) must be deployed so that every community or demand point is served within an acceptable response time. The Set Covering Location Problem (SCLP) is a classic mathematical model designed to minimize the number of facilities needed while guaranteeing that every demand point is covered. Its applications range from urban planning to disaster response and logistics.

OBJECTIVE FUNCTION

Our aim is to minimize the total number of facilities built. This can be expressed as the sum of all our decision variables. Since X_j is 1 for each facility built and 0 otherwise, the sum of all X_j will give us the total number of facilities.

The objective function is:

Minimize $\sum_{j=1}^{n} x_j$

CONSTRAINT ensures that every demand point is covered by at least one facility located within the maximum allowable response time. For each demand point.

$$\sum_{j \in J} a_{ij}\, x_j \geq 1 \quad \forall i \in I$$

• I: The set of all demand points requiring coverage, indexed by i

• J: The set of all potential facility locations, indexed by j

• a_{ij}: A parameter that is 1 if a facility at potential site j can cover demand point i within the maximum response time, and 0 otherwise.

BINARY VARIABLE CONSTRAINT

$$X_j \in \{0,1\} \quad \forall j \in J$$

EXAMPLE:

F = Facilities

These are potential locations where you could build service centers like fire stations, hospitals, or emergency response bases. In our model, F1 through F8 represent 8 possible locations you could choose from.

D = Demand Points

These are locations that need service and must be covered, such as neighborhoods, schools, or business districts. In our model, D1 through D8 represent 8 areas that require service coverage.

The optimization problem is:

Select the minimum number of facilities (F) needed to cover all demand points (D) within the response time limit. [How many centers are to be operated and in which locations?]

	F1	F2	F3	F4	F5	F6	F7	F8
D1	5.49	14.79	28.27	10.41	22.77	4.85	26.13	19.6
D2	9.99	4.71	16.71	12.45	16.66	9.42	13.74	17.3
D3	17.94	4.71	11.33	11.33	7.45	13.33	12.01	10.19
D4	25.87	12.16	4.21	19.81	9.61	22.03	8.89	15.55
D5	15.6	13.86	25.14	2.98	14.72	8.24	25.44	9.54
D6	23.55	10.41	10.17	14.79	3.59	18.29	13.47	9.42
D7	3.59	13.72	25.74	16.22	25.01	10.19	21.99	24.02
D8	18.7	7.08	8.43	17.93	14.72	14.72	5.07	18.43

Figure 2.1.a: Distance Matrix

This table shows the initial distance (in km) from each facility locations (F1 – F8) to each demand points (D1-D8)

Service Parameters:

Speed: 40 km/h (typical emergency vehicle speed in urban areas)

Max Response Time: 8 minutes (industry standard for emergency services)

Max Coverage Distance = 40 km/h × (8 min ÷ 60 min/h) = 5.33 km

[Max distance allowed = speed × travel time]

If (distance <= 5.33), then 1, otherwise 0

Coverage Radius: 5.33 km

Coverage Area per Facility: $\pi \times (5.33)^2 \approx 89.3$ km²

EXCEL IMPLEMENTATION

Applied the formula =IF(Cell<=5.33,1,0) to distance matrix to determine which facilities can cover which demand points within the limit.

	F1	F2	F3	F4	F5	F6	F7	F8
D1	0	0	0	0	0	1	0	0
D2	0	1	0	0	0	0	0	0
D3	0	1	0	0	0	0	0	0
D4	0	0	1	0	0	0	0	0
D5	0	0	0	1	0	0	0	0
D6	0	0	0	0	1	0	0	0
D7	1	0	0	0	0	0	0	0
D8	0	0	0	0	0	0	1	0

Figure 2.1.b: Coverage Matrix (Binary a_{ij} values)

This matrix converts the raw distance data into binary parameter a_{ij}, which is 1 if the distance is $\leq 5.33\ km$ (covered) and 0 otherwise (not covered)

Set covering constraints:

F6 ≥ 1 (Cover D1)

F2 ≥ 1 (Cover D2)

F2 ≥ 1 (Cover D3)

F3 ≥ 1 (Cover D4)

F4 ≥ 1 (Cover D5)

F5 ≥ 1 (Cover D6)

F1 ≥ 1 (Cover D7)

F7 ≥ 1 (Cover D8)

SOLVER IMPLEMENTATION:

Set the Objective to minimize the sum of facility decision variables. Set the Variable Cells to binary. Added Set Covering Constraints to ensure every demand point is covered.

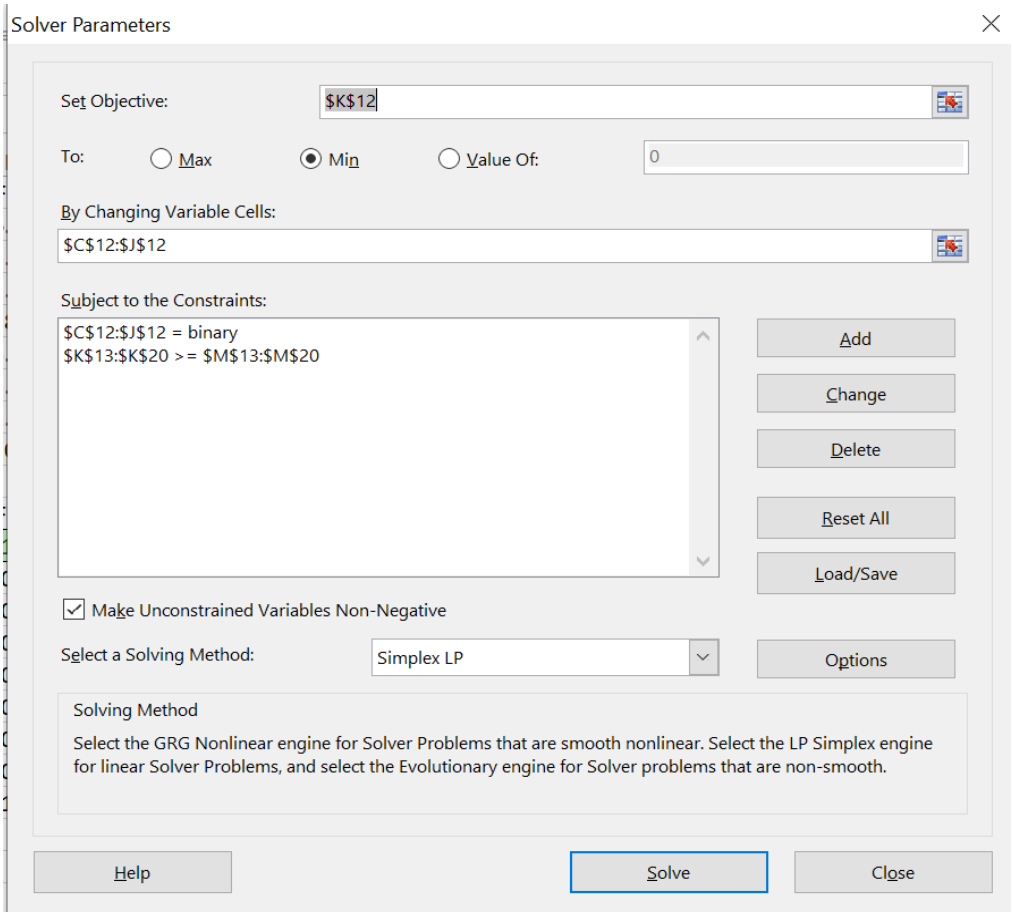


Figure 2.1.c: SCLP Solver parameters

Solver parameters are used to configure the optimization problem in Excel's Solver tool.

SOLUTION AND ANALYSIS

	F1	F2	F3	F4	F5	F6	F7	F8			
Select	1	1	1	1	1	1	1	0	7		
D1	0	0	0	0	0	1	0	0	1	>=	1
D2	0	1	0	0	0	0	0	0	1	>=	1
D3	0	1	0	0	0	0	0	0	1	>=	1
D4	0	0	1	0	0	0	0	0	1	>=	1
D5	0	0	0	1	0	0	0	0	1	>=	1
D6	0	0	0	0	1	0	0	0	1	>=	1
D7	1	0	0	0	0	0	0	0	1	>=	1
D8	0	0	0	0	0	0	1	0	1	>=	1

Figure 2.1.d: Solution

The above *Figure 2.1.d* displays the optimal selection of facilities that satisfy all constraints.

Optimal Facility Location Selected: F1, F2, F3, F4, F5, F6, F7

Total Facilities: 7

Coverage: 8/8 demand points (100.0%)

Efficiency: This solution minimizes the number of facilities while ensuring complete coverage.

This solution demonstrates the mathematical model's effectiveness: every model component maps directly to a real operational rule.

2.2 MAXIMUM COVERING LOCATION PROBLEM (MCLP)

Resources are often limited, especially in emergencies or budget-constrained scenarios. The Maximum Covering Location Problem (MCLP) addresses the challenge of maximizing coverage (by population or demand) when only a restricted number of facilities can be established. This model is vital for planning ambulance stations, retail outlets, and disaster response bases in environments with hard limits on resource expansion.

OBJECTIVE FUNCTION:

The objective is to maximize the total population covered by the limited facilities.

Maximize $\boldsymbol{z = \sum_{j \in J} w_j y_j}$

Z is the total covered population.

J: Demand neighborhoods

I: Candidate facility locations

W_j is the population of neighborhood j

Y_j is a binary decision variable that equals 1 if neighborhood j is covered, and O if it is not.

FACILITY LOCATION CONSTRAINT

$$\sum_{i \in I} X_i = p$$

I is the set of all candidate locations where a facility can be built.

X_i is a binary decision variable that equals 1 if a facility is built at location i, and O otherwise.

p is the total number of facilities to be located.

COVERAGE CONSTRAINS

This constraint links the facility location decisions (X_i) to the coverage outcomes (Y_j). A neighborhood j can only be covered ($Y_j = 1$) if at least one facility is built at a location i that can cover it within the specified maximum travel time, S.

$$Y_j \leq \sum_{i \in N_j} X_i \quad \forall j \epsilon J$$

N_j is the set of all candidate locations i that can cover neighborhood j within the maximum travel time. This set is defined based on the travel time matrix (d_{ij}) and the maximum travel time threshold (S).

BINARY VARIABLE CONSTRAINS

$$X_{i,} Y_j \in \{0,1\} \ \forall i \in j \in J$$

EXAMPLE PROBLEM STATEMENT

Let D1, D2, D3, D4, D5, D6 be the demand points and F1, F2, F3, F4, F4, F5, F6 are the facilities.

Maximize the coverage of demand points by assuming that the number of ambulance stations to be opened is limited to 3 due to budget constraints and ensuring that emergency response time does not exceed 5 minutes to any covered neighborhood. The maximal covering location problem model is used to achieve 100% fulfilment.

GIVEN DATA:

Demand Points	Population
D1	35000
D2	60000
D3	55000
D4	40000
D5	50000
D6	70000

Figure 2.2.a: Population at each demand points

Travel Time Matrix (minutes)						
From/To	F1	F2	F3	F4	F5	F6
D1	6	8	15	10	7	5
D2	9	15	22	7	5	8
D3	6	12	20	8	6	7
D4	12	18	25	5	8	10
D5	10	7	18	13	6	9
D6	15	20	8	12	14	16

Figure 2.2.b: Travel Time Matrix (minutes)

The above *Figure 2.2.b* shows the travel time (in minutes) from each facility (F1−F6) to each demand point (D1−D6). We used this data to identify all facilities that could cover a neighbourhood within the Max Response Time of 5 minutes.

A neighborhood is considered "covered" if at least one ambulance station can reach it within the 5-minute response time limit.

SOLVER IMPLEMENTATION

Use Excel Solver to maximize total covered population.

Set the Objective to maximize the total covered population.

Added the Budget Constraint requiring the total number of facilities open to be exactly 3.

Set both facility and coverage decision variables to binary.

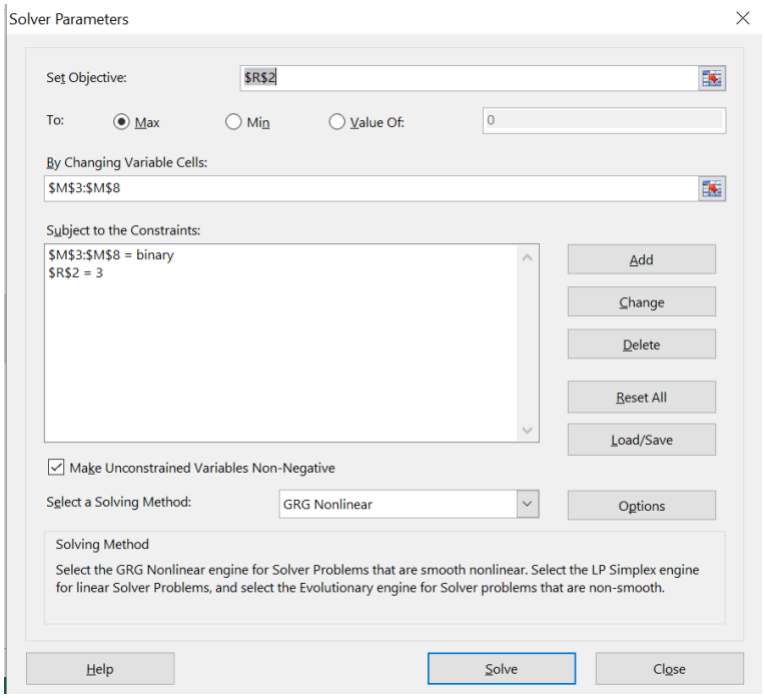


Figure 2.2.c: MCLP Solver Parameters

Purpose of the Solver Parameters is to configure the MCLP optimization in Solver.

SOLUTION

Decision Variables		coverage status	MAX RESPONSE TIME	5
Potential Station Location	Open (1) / Not Open (0)	NBD	COVERED (1)/NOT COVERED(0)	
F1	0	D1	1	
F2	0	D2	1	
F3	0	D3	0	
F4	1	D4	1	
F5	1	D5	0	
F6	1	D6	0	

Figure 2.2.d: MCLP Solution (Open/Not Open and Coverage Status)

The above figure presents the decision variables and the resulting coverage status after the maximization. The Solver selected facilities F4, F5, and F6 to open. Details which demand points are covered by the selected facilities.

TOTAL STATIONS OPEN	3
TOTAL POPULATION COVERED	135000
COVERAGE PERCENTAGE	0.435483871

Figure 2.2.d: Result

RESULTS AND DISCUSSION

Optimal Station Locations:

F4, F5, F6

Coverage with F4 + F5 + F6:

Table 2.1.a

DEMAND	F4	F5	F6	Min Time	Covered	Population
D1	10	7	5	5≤5	yes	35000
D2	7	5	8	5≤5	yes	60000
D3	8	6	7	6>5	NO	55000
D4	5	8	10	5≤5	yes	40000
D5	13	6	9	6>5	NO	50000
D6	12	14	16	12>15	NO	70000

Table 2.1.a, Confirmed that the minimum travel time to D1, D2, and D4 was ≤5 minutes, resulting in a Total Population Covered of 135,000.

TOTAL POPULATION COVERED = 135000

COVERAGE PERCENTAGE = 0.435483871

2.3 MIXED-INTEGER LINEAR PROGRAMMING (MILP)

Facility location is often intertwined with the operational logistics of satisfying demand from multiple sources with capacity, cost, and selection constraints. Mixed-Integer Linear Programming (MILP) models extend previous models to simultaneously decide facility locations and resource flows in complex networks. These are widely used in supply chain design, transport logistics, and multi-service resource allocation.

EXAMPLE PROBLEM:

There are 5 facilities and 5 demand points. Data for production and transportation from facilities to demand points, demand for each of the demand points and fixed cost for running each of the facilities are given. Capacity of each of the supply points = 20.

Find quantities to shipped from the facility to the demand region and as well as which facilities to be used.

FACILITIES	DEMAND POINTS -PROD & TRANSPORT COST					FIXED COST	CAPACITY
	D1	D2	D3	D4	D5		
F1	81	92	101	13	115	9000	20
F2	117	77	108	98	100	6750	20
F3	102	105	95	119	111	9750	20
F4	115	125	90	59	74	6150	20
F5	142	100	103	105	71	6000	20
DEMAND	12	8	14	16	7	57	

Figure 2.3.a: MILP Data Table

Figure 2.3.a provides all parameters for the problem, including production/transportation costs, fixed facility costs, capacity and demand. We are using this data to formulate the objective function: Minimize Total Cost = Fixed Costs + Variable Costs

i: Facilities (1 to n)

j: Demand points (1 to m)

Parameters**:**

F_i : Fixed cost for facility i

c_{ij} **:** Transportation cost from i to j

d_j : Demand at location j

k_i : Capacity at i

Variables**:**

Y_i : 1 if facility i is used; 0 otherwise

x_{ij} : Quantity of emergency material/patients/personnel transported from facility j to demand point i.

MINIMIZE TOTAL COST = FIXED COST + VARIABLE COST

$$= \sum_{i=1}^{n} F_i Y_i + \sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij}\, x_{ij}$$

DEMAND CONSTRAINT:

$$\sum_{i=1}^{n} x_{ij} = d_j \Longrightarrow d_j - \sum_{i=1}^{n} x_{ij} = 0$$

CAPACITY CONSTRAINT:

$$\sum_{j=1}^{m} x_{ij} \leq k_i y_i \Longrightarrow k_i y_i - \sum_{j=1}^{m} x_{ij} \geq 0$$

$$y_i = 0 \; or \; 1$$

$$x_{ij} \geq 0$$

SOLVER IMPLEMENTATION

Solver Parameters is used to configure the integrated facility location and resource flow problem.

Set the Objective to Minimize the Total Cost cell.

Set the facility use variables to binary and the flow variables to non-negative.

Enforced Demand Constraints and Capacity Constraints.

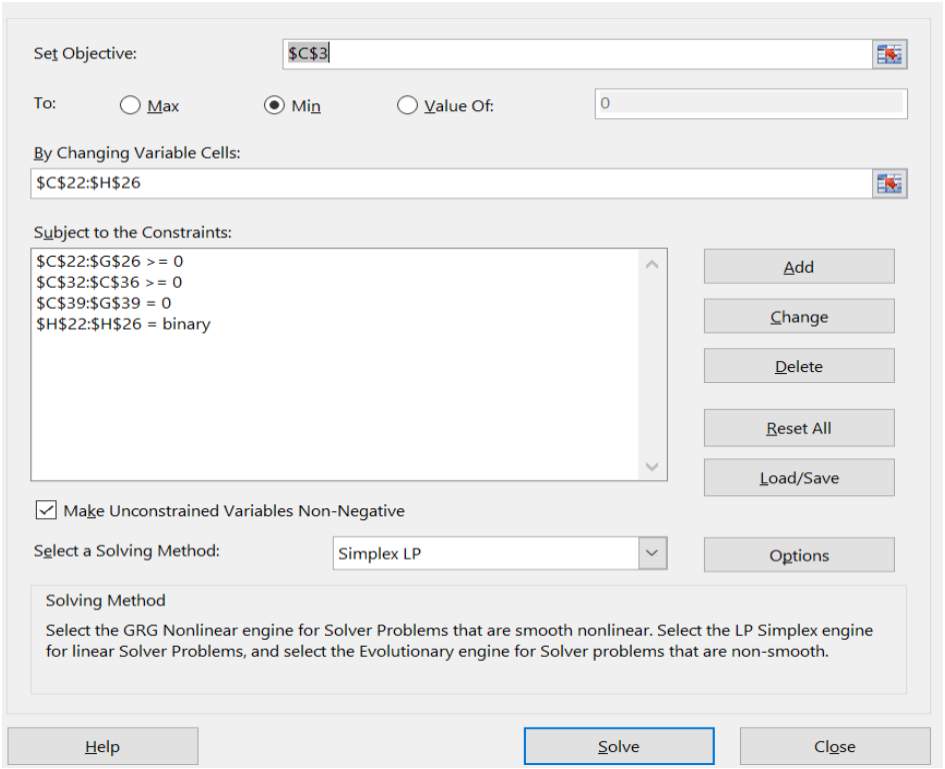


Figure 2.3.b: MILP Solver Parameters

SOLUTION

DECISION VARIABLES

FACILITIES	DEMAND POINTS					FACILITY OPEN/NOT	
	D1	D2	D3	D4	D5		
F1	0	0	0	0	0	0	
F2	12	8	0	0	0	1	20
F3	0	0	0	0	0	0	
F4	1.78E-15	0	4	16	0	1	20
F5	0	0	10	0	7	1	20

Figure 2.3.c: MILP Solution (Decision Variables)

Shows which facilities were selected (Facility Open/Not) and the optimal flow quantities from the chosen facilities to the demand points. The Solver selected facilities F2, F4, and F5 to open and determined the specific quantities to be shipped to meet all demand.

Facilities used: F2, F4, F5

D1 with demand 12 is satisfied by F2

D2 with demand 8 is satisfied by F2

D3 with demand 14 is partially satisfied by F4 (4) and partially satisfied by F5 (10)

D2 with demand 16 is satisfied by F4

D2 with demand 7 is satisfied by F5

CONSTRAINTS

FACILITIES	EXCESS CAPCITY
F1	0
F2	0
F3	0
F4	0
F5	3

	D1	D2	D3	D4	D5
UNMET DEMAND	0	0	0	0	0

Figure 2.3.d: shows excess capacity and unmet demands

F5 has Excess capacity of 3 unit. All 5 demand points are met.

OBJECTIVE FUNCTION

Z =	23751

Figure 2.3.e: MILP Objective Function

Figure 2.3.e displays the resulting minimized total cost.

The Total Cost (Z) was determined to be 23751.

CONCLUSION for chapter 2: Resource Allocation using Linear Programming

This chapter successfully demonstrated the application of Linear Programming (LP) and Mixed-Integer Linear Programming (MILP) models to critical resource allocation and facility location problems in public service and logistics. By exploring the Set Covering Location Problem (SCLP), the Maximum Covering Location Problem (MCLP), and a Mixed-Integer Facility Location Problem, we showcased how mathematical optimization provides actionable, data-driven solutions for complex decision-making.

The SCLP model focused on efficiency by minimizing the number of facilities required to achieve 100% demand coverage within a set response time. In contrast, the MCLP addressed resource scarcity by maximizing the population covered when the number of facilities was strictly limited by a budget constraint. Finally, the MILP example integrated location decisions with operational logistics, minimizing the total fixed and variable costs while satisfying demand and adhering to capacity constraints.

The consistent use of Excel's Solver across all problems validated its utility as a practical tool for implementing and solving these powerful optimization models. The solutions derived—selecting a minimum of 7 facilities (F1-F7) for SCLP, maximizing coverage by selecting F4, F5, and F6 for MCLP, and minimizing cost by selecting F2, F4, and F5 for the MILP—directly answer real-world questions regarding where to locate resources and how to operate them optimally. This framework is essential for urban planning, emergency services, and supply chain management, ensuring maximum service delivery within fiscal and physical constraints.

CHAPTER: 3
ROUTE OPTIMIZATION USING GRAPH THEORY

3.1 GRAPH THEORY AND ROAD NETWORKS

In graph theory, the shortest path/Minimum cost path between nodes s (start) and t (target) can be defined as:

$$d(s, t) = min \sum w(i, j) \times x(i, j) \text{ where,}$$

- $d(s, t)$ represents the shortest distance/cost from node s to node t
- $x(i, j)$ is a binary variable indicating whether edge (i, j) is included in the shortest path
- $w(i, j)$ is the weight of edge (i, j)

Applications to emergency response involve several critical aspects. Firstly, **Emergency Vehicle Routing** determines the most efficient routes for emergency vehicles, such as ambulances or fire trucks, to reach emergency locations. This can be achieved by using advanced algorithms like Dijkstra's algorithm or A* algorithm, which take into account real-time traffic data, road conditions, and other factors to optimize routes. Secondly, **Resource Allocation** optimizes the allocation of emergency resources, such as personnel or equipment, to emergency locations, using data analytics and machine learning to predict the likelihood of emergencies in different areas and allocate resources accordingly. Thirdly, **Evacuation Planning** determines the most efficient evacuation routes for people in danger, minimizing risk and response time, by utilizing geographic information systems (GIS) to identify the safest and most efficient evacuation routes, considering factors like road conditions, weather, and potential hazards. These applications can significantly enhance emergency response operations.

Optimized routing in graph theory has numerous practical applications in emergency response, including emergency services like ambulance, fire, and police departments, which can leverage graph algorithms to improve response times and reduce costs. Disaster response teams can also benefit from graph-based optimized routing, using it to quickly and safely evacuate people from affected areas. Furthermore, hospitals and healthcare organizations can utilize graph-based optimized routing to rapidly transport patients, medical supplies, and personnel, ultimately enhancing the efficiency and effectiveness of emergency response operations.

By applying graph theory and advanced algorithms to emergency response operations, organizations can improve response times, increase efficiency, and enhance safety, ultimately saving lives and reducing the impact of emergencies.

3.2 SHORTEST PATH ALGORITHMS

Shortest path algorithms are crucial in finding the most efficient routes in emergency response situations. These algorithms can be categorized into single-source shortest path algorithms, such as Dijkstra's and Bellman-Ford, which find the shortest path from a single source node to all other nodes; all-pairs shortest path algorithms, like Floyd-Warshall, which determine the shortest path between all pairs of nodes; and informed search algorithms, such as A*, which use heuristics to guide the search towards the destination node. By understanding and applying these algorithms, emergency responders can optimize routes, reduce response times, and ultimately save lives. Each algorithm has its strengths and limitations, and selecting the most suitable one depends on the specific requirements of the emergency response scenario.

3.2.1 DIJKSTRA'S ALGORITHM

Dijkstra's algorithm is a popular method for finding the shortest path from a single source node to all other nodes in a weighted graph. It is a Single Sourced Shortest Path (SSSP) algorithm for graphs with non-negative edge weights. Dijkstra's algorithm requires the graph to have non-negative edge weights, ensuring that once a node is visited, its optimal distance cannot be improved. This constraint is crucial because the algorithm relies on the property that the shortest path to a node is composed of shortest paths to its predecessors; with negative edge weights, this property no longer holds, potentially leading to incorrect results. This property is especially important because it enables Dijkstra's algorithm to act in a greedy manner by always selecting the next most promising node with the minimum distance at each step.

Let $G = (V, E)$ be a weighted directed graph, where V is a finite set of vertices and $E \subseteq V \times V$ is a set of directed edges. Each edge $e = (u, v) \in E$ has a non-negative weight $w(e) = w(u, v) \in \mathbb{R} \geq 0$.

Given a source vertex $s \in V$, the shortest path problem is to find a path P from s to every other vertex $v \in V$ that minimizes the total weight $w(P) = \sum_{e \in p} w(e)$.

Dijkstra's algorithm is a combinatorial optimization algorithm that solves the shortest path problem. It can be formulated as a dynamic programming algorithm that iteratively updates the shortest distance labels d(v) for each vertex v ∈ V.

Let $d(v)$ denote the shortest distance from s to v. The algorithm maintains a priority queue Q of vertices, where the priority of each vertex v is its current shortest distance label $d(v)$

MATHEMATICAL FORMULATION

The algorithm can be defined as follows:

1. Initialize $d(s) = 0$ and $d(v) = \infty$ for all $v \in V \setminus \{s\}$

2. Create a priority queue $Q = V$ with priorities $d(v)$

3. While $Q \neq 0$;

Extract the vertex $u \in Q$ with the minimum priority $d(u)$. For each neighbor v of u, update the shortest distance label $d(v)$ as follows:

$d(v) = min(d(v), d(u) + w(u, v))$

Where u is the current node, v is the neighboring node, and $w(u, v)$ is the weight of the edge between u and v.

OPTIMALITY

Dijkstra's algorithm has two primary time complexities:

$O(V^2)$ using an array to find the minimum distance vertex and $O((V + E) \log V)$ using a binary heap as a priority queue, where V is the number of vertices and E is the number of edges. With a more advanced Fibonacci heap, the complexity can be improved to $O(E + V \log V)$.

ILLUSTRATIVE EXAMPLE

Suppose a city has five major intersections labeled A, B, C, D, and E. These intersections are connected by roads with the following travel times (in minutes):

STARTING POINT [FROM]	ENDING POINT [TO]	TIME TAKEN [IN MINUTES]
A	B	4
A	C	2
B	C	1
B	D	5
C	D	8
C	E	10
D	E	2

Table 3.2.1.a Intersection and Travel Times

An ambulance is stationed at intersection A and receives an emergency call from intersection E. The goal is to determine the quickest route for the ambulance to reach E.

Applying Dijkstra's Algorithm**:**

Initializing Distances

Setting the starting node (A) distance to 0.

Setting all other node distances to ∞.

Marking all nodes as unvisited.

Visiting the Current Node,

Begin at node A (distance 0).

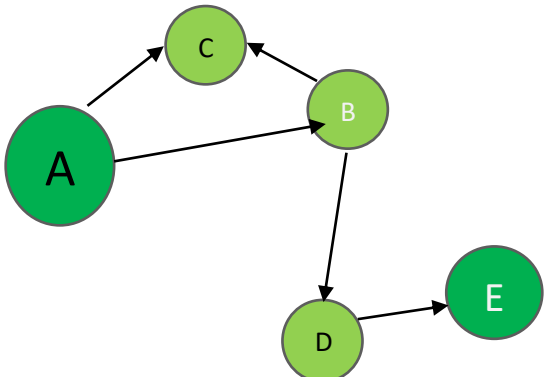


Figure 3.2.1.a Graph

Updating Neighboring Nodes

For each unvisited neighbor of the current node, calculate the tentative distance from A.

If this distance is less than the currently recorded distance, update it.

From A:

To B: 0 + 4 = 4

To C: 0 + 2 = 2

Updated distances:

A: 0 (visited)

B: 4

C: 2

D: ∞

E: ∞

Mark the Current Node as Visited

Mark A as visited. The next node to visit is the unvisited node with the smallest tentative distance (C, with distance 2).

Repeat the Process

From C:

To B: 2 + 1 = 3 (update B from 4 to 3)

To D: 2 + 8 = 10

To E: 2 + 10 = 12

Updated distances:

A: 0 (visited)

B: 3

C: 2 (visited)

D: 10

E: 12

Mark C as visited. Next, visit B (distance 3).

From B:

To D: 3 + 5 = 8 (update D from 10 to 8)

To C: already visited

Updated distances:

A: 0 (visited)

B: 3 (visited)

C: 2 (visited)

D: 8

E: 12

Mark B as visited. Next, visit D (distance 8).

From D:

To E: 8 + 2 = 10 (update E from 12 to 10)

Updated distances:

A: 0 (visited)

B: 3 (visited)

C: 2 (visited)

D: 8 (visited)

E: 10

Mark D as visited. Next, visit E (distance 10).

Finish

All nodes are now visited. The shortest distance from A to E is 10 minutes.

This step-by-step application of Dijkstra's algorithm determines the minimum travel time from the ambulance's starting point (A) to the emergency location (E).

HYPOTHETICAL EXAMPLE: APPLYING DIJKSTRA'S ALGORITHM TO EMERGENCY RESPONSE

EXAMPLE 1

Let's consider an emergency response network with 30 nodes, representing hospitals, emergency response centers, and intersections. We'll use Dijkstra's algorithm to find the shortest path from each emergency response center to the nearest hospital. Each node will have a unique ID and a type (hospital, emergency response center, or intersection). We'll use the following node types:

H: Hospital

E: Emergency Response Center

I: Intersection

The graph has the following edges:

1. E1 to I1 with weight 2
2. E1 to I2 with weight 3
3. E2 to I3 with weight 1
4. E2 to I4 with weight 4
5. I1 to H1 with weight 2
6. I1 to I5 with weight 1
7. I2 to H2 with weight 3
8. I2 to I6 with weight 2
9. I3 to H3 with weight 1
10. I3 to I7 with weight 3
11. I4 to H4 with weight 2
12. I4 to I8 with weight 1
13. I5 to H1 with weight 1
14. I6 to H2 with weight 2
15. I7 to H3 with weight 2
16. I8 to H4 with weight 1
17. I9 to E1 with weight 2
18. I9 to I10 with weight 3
19. I10 to E2 with weight 1
20. I10 to I11 with weight 2
21. I11 to I12 with weight 1
22. I12 to I13 with weight 2
23. I13 to I14 with weight 3
24. I14 to I15 with weight 1
25. I15 to I16 with weight 2
26. I16 to I17 with weight 1
27. I17 to I18 with weight 3
28. I18 to I19 with weight 2

29. I19 to I20 with weight 1
30. I20 to I21 with weight 2
31. I21 to I22 with weight 3
32. I22 to I23 with weight 1
33. I23 to I24 with weight 2
34. I24 to I25 with weight 3
35. I25 to I26 with weight 1
36. I26 to I27 with weight 2
37. I27 to I28 with weight 3
38. I28 to I29 with weight 1
39. I29 to I30 with weight 2
40.

These edges represent the connections between the nodes in the graph, with weights representing the distance or time between each pair of nodes in kilometre.

CODE IMPLEMENTATION

```
import sys
import heapq

def dijkstra(graph, start):
    distances = {node: sys.maxsize for node in graph}
    distances[start] = 0
    queue = [(0, start)]

    while queue:
        current_distance, current_node = heapq.heappop(queue)

        if current_distance > distances[current_node]:
            continue

        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(queue, (distance, neighbor))

    return distances

# Define the graph
graph = {
    'E1': {'I1': 2, 'I2': 3},
    'E2': {'I3': 1, 'I4': 4},
    'I1': {'H1': 2, 'I5': 1},
    'I2': {'H2': 3, 'I6': 2},
    'I3': {'H3': 1, 'I7': 3},
```

```
    'I4': {'H4': 2, 'I8': 1},
    'I5': {'H1': 1},
    'I6': {'H2': 2},
    'I7': {'H3': 2},
    'I8': {'H4': 1},
    'H1': {},
    'H2': {},
    'H3': {},
    'H4': {},
    'I9': {'E1': 2, 'I10': 3},
    'I10': {'E2': 1, 'I11': 2},
    'I11': {'I12': 1},
    'I12': {'I13': 2},
    'I13': {'I14': 3},
    'I14': {'I15': 1},
    'I15': {'I16': 2},
    'I16': {'I17': 1},
    'I17': {'I18': 3},
    'I18': {'I19': 2},
    'I19': {'I20': 1},
    'I20': {'I21': 2},
    'I21': {'I22': 3},
    'I22': {'I23': 1},
    'I23': {'I24': 2},
    'I24': {'I25': 3},
    'I25': {'I26': 1},
    'I26': {'I27': 2},
    'I27': {'I28': 3},
    'I28': {'I29': 1},
    'I29': {'I30': 2},
    'I30': {}
}

emergency_response_centers = ['E1', 'E2']
hospitals = ['H1', 'H2', 'H3', 'H4']


for emergency_response_center in emergency_response_centers:
    distances = dijkstra(graph, emergency_response_center)
    nearest_hospital = None
    min_distance = sys.maxsize

    for hospital in hospitals:
        if distances[hospital] < min_distance:
            min_distance = distances[hospital]
            nearest_hospital = hospital

print(f"Nearest hospital to {emergency_response_center} is {nearest_hospital} with a
distance of {min_distance}")
```

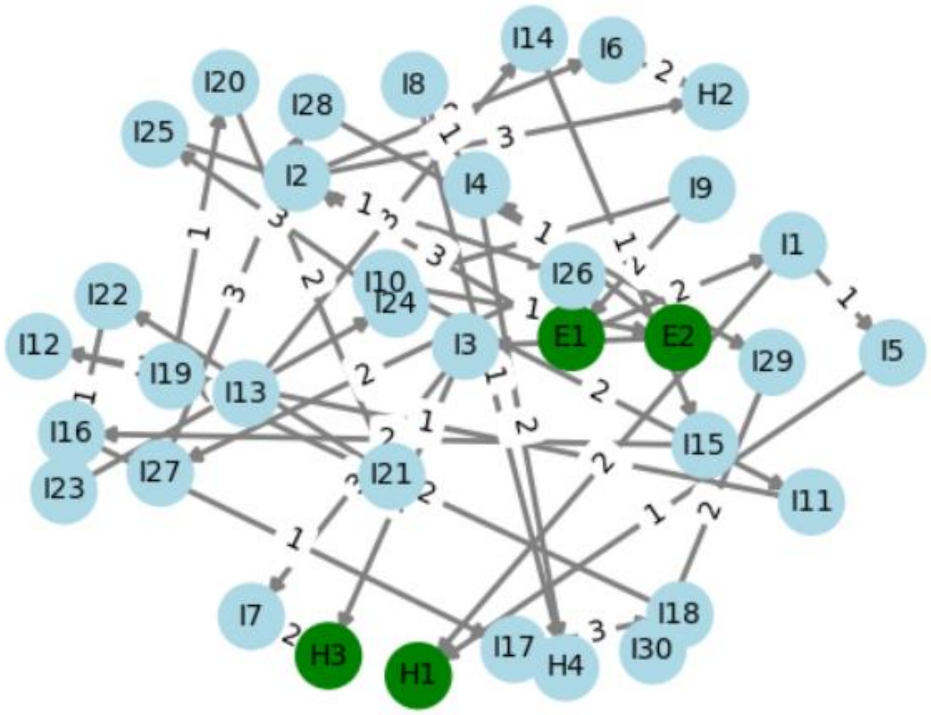


Figure 3.2.1.b: Representation of nodes and edges

RESULT:

Nearest hospital to E1 is H1 with a distance of 4km

Nearest hospital to E2 is H3 with a distance of 2km.

EXAMPLE 2: TRAFFIC-AWARE SHORTEST PATH PROBLEM

We need to find shortest path from node A to node T in a graph with 20 nodes, where the edge weights represent travel times that vary according to traffic conditions. The graph is as follows:

Nodes: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T

Edg**es** with weights (travel times in minutes) that vary according to traffic conditions:

Table 3.2.1.b Traffic Aware Edge Weights

EDGE	Weight (Low Traffic)	Weight (Moderate Traffic)	Weight (high Traffic)
A - B	5	10	20
A - C	3	6	12
B - D	4	8	16
B - E	2	4	8
C - F	6	12	24

D - G	3	6	12
E - H	5	10	20
F- I	4	8	16
G - J	2	4	8
H - K	6	12	24
I - L	3	6	12
J - M	5	10	20
K - N	4	8	16
L - O	2	4	8
M - P	6	12	24
N - Q	3	6	12
O - R	5	10	20
P - S	4	8	16
Q - T	2	4	8
R - S	3	6	12
S - T	5	10	20

The traffic conditions are as follows:

Low traffic: 00:00 - 07:00 and 19:00 - 23:59

Moderate traffic: 07:00 - 10:00 and 16:00 - 19:00

Heavy traffic: 10:00 - 16:00

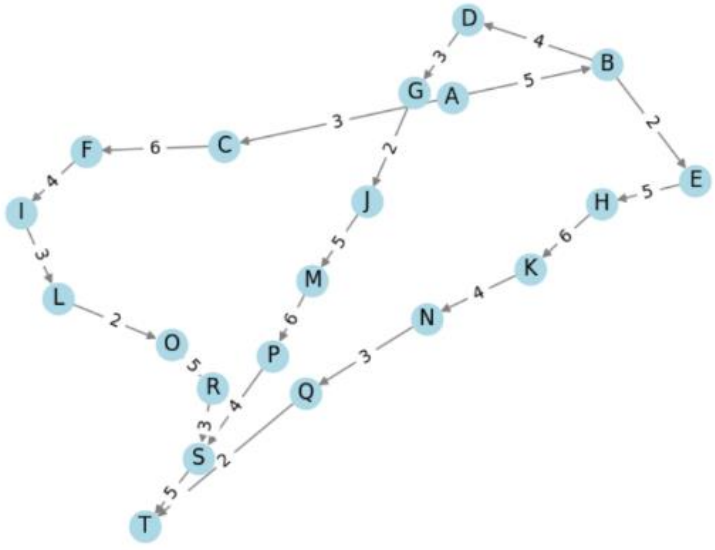


Figure 3.2.1.c: representation of nodes and edges

CODE IMPLEMENTATION

```
import sys
import heapq

def dijkstra(graph, start):
    distances = {node: sys.maxsize for node in graph}
    distances[start] = 0
    queue = [(0, start)]

    while queue:
        current_distance, current_node = heapq.heappop(queue)

        if current_distance > distances[current_node]:
            continue

        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(queue, (distance, neighbor))

    return distances

# Define the graphs for different traffic conditions
graph_low_traffic = {
    'A': {'B': 5, 'C': 3},
    'B': {'D': 4, 'E': 2},
    'C': {'F': 6},
    'D': {'G': 3},
    'E': {'H': 5},
    'F': {'I': 4},
    'G': {'J': 2},
    'H': {'K': 6},
    'I': {'L': 3},
    'J': {'M': 5},
    'K': {'N': 4},
    'L': {'O': 2},
    'M': {'P': 6},
    'N': {'Q': 3},
    'O': {'R': 5},
    'P': {'S': 4},
    'Q': {'T': 2},
    'R': {'S': 3},
    'S': {'T': 5},
    'T': {}
}

graph_moderate_traffic = {
```

```
    'A': {'B': 10, 'C': 6},
    'B': {'D': 8, 'E': 4},
    'C': {'F': 12},
    'D': {'G': 6},
    'E': {'H': 10},
    'F': {'I': 8},
    'G': {'J': 4},
    'H': {'K': 12},
    'I': {'L': 6},
    'J': {'M': 10},
    'K': {'N': 8},
    'L': {'O': 4},
    'M': {'P': 12},
    'N': {'Q': 6},
    'O': {'R': 10},
    'P': {'S': 8},
    'Q': {'T': 4},
    'R': {'S': 6},
    'S': {'T': 10},
    'T': {}
}

graph_heavy_traffic = {
    'A': {'B': 20, 'C': 12},
    'B': {'D': 16, 'E': 8},
    'C': {'F': 24},
    'D': {'G': 12},
    'E': {'H': 20},
    'F': {'I': 16},
    'G': {'J': 8},
    'H': {'K': 24},
    'I': {'L': 12},
    'J': {'M': 20},
    'K': {'N': 16},
    'L': {'O': 8},
    'M': {'P': 24},
    'N': {'Q': 12},
    'O': {'R': 20},
    'P': {'S': 16},
    'Q': {'T': 8},
    'R': {'S': 12},
    'S': {'T': 20},
    'T': {}
}

start_node = 'A'

distances_low_traffic = dijkstra(graph_low_traffic, start_node)
distances_moderate_traffic = dijkstra(graph_moderate_traffic, start_node)
distances_heavy_traffic = dijkstra(graph_heavy_traffic, start_node)
```

```
print("Shortest distance from node A to node T in low traffic:", distances_low_traffic['T'])
print("Shortest distance from node A to node T in moderate traffic:",
distances_moderate_traffic['T'])
print("Shortest distance from node A to node T in heavy traffic:",
distances_heavy_traffic['T'])
```

RESULT:

Shortest distance from node A to node T in low traffic: 27 Minutes

Shortest distance from node A to node T in moderate traffic: 54 Minutes

Shortest distance from node A to node T in heavy traffic: 108 Minutes

3.2.2. BELLMAN FORD ALGORITHM

The Bellman-Ford algorithm is a graph search algorithm that finds the shortest path from a source vertex to all other vertices in a weighted graph. It can handle negative weight edges and detect negative cycles, making it suitable for specific emergency response scenarios.

METHODOLOGY

These are the following steps taken to apply the Bellman -Ford algorithm in emergency response location:

1. Graph Construction: Create a weighted graph representing the emergency response network.

2. Edge Weight Assignment: Assign weights to edges based on distance, time, or other relevant factors.

3. Algorithm Implementation: Implement the Bellman-Ford algorithm to find the shortest path from each emergency location to the nearest response center.

4. Analysis and Optimization: Analyze the results and optimize the emergency location allocation model using the Bellman-Ford algorithm's output.

BELLMAN-FORD ALGORITHM EQUATION

$$d(v) = \min(d(v), d(u) + w(u, v)) \text{ where:}$$

- d(v) is the distance from the source vertex to vertex v
- d(u) is the distance from the source vertex to vertex u
- w (u, v) is the weight of the edge from vertex u to vertex v

By applying the Bellman-Ford algorithm, emergency response systems can be optimized to reduce response times and improve outcomes.

OPTIMALITY

The Bellman-Ford algorithm has a time complexity of O(V·E), where V is the number of vertices and E is the number of edges in the graph. This complexity arises because the algorithm performs edge relaxations V-1 times (where V is the number of vertices), and in each iteration, it checks all E edges

ROUTE OPTIMIZATION USING BELLMAN-FORD ALGORITHM [TO DETERMINE THE SHORTEST PATH:

HYPOTHETICAL EXAMPLE 1

During a natural disaster, emergency responders need to navigate through a network of roads to reach affected areas. The road network is represented as a weighted graph, where each edge represents a road segment, and the weight represents the time it takes to traverse that segment. However, some roads have been designated as "priority routes" and emergency responders can gain bonus time credits for using these routes.

The graph has the following structure:

Nodes: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y

Edges:

A -> B: weight = 5

A -> C: weight = -3 (priority route)

B -> D: weight = 2

C -> D: weight = 4

C -> E: weight = -2 (priority route)

D -> F: weight = 1

E -> F: weight = 3

F -> G: weight = 2

G -> H: weight = 1

H -> I: weight = 4

I -> J: weight = -1

J -> K: weight = 3

K -> L: weight = 2

L -> M: weight = 1

M -> N: weight = -2

N -> O: weight = 4

O -> P: weight = 3

P -> Q: weight = 2

Q -> R: weight = 1

R -> S: weight = -3

S -> T: weight = 2

T -> U: weight = 1

U -> V: weight = 4

V -> W: weight = 3

W -> X: weight = 2

X -> Y: weight = 1

The emergency responders start at node A and need to reach node Y. we use the Bellman-Ford algorithm to find the shortest path from A to F, taking into account the negative weights.

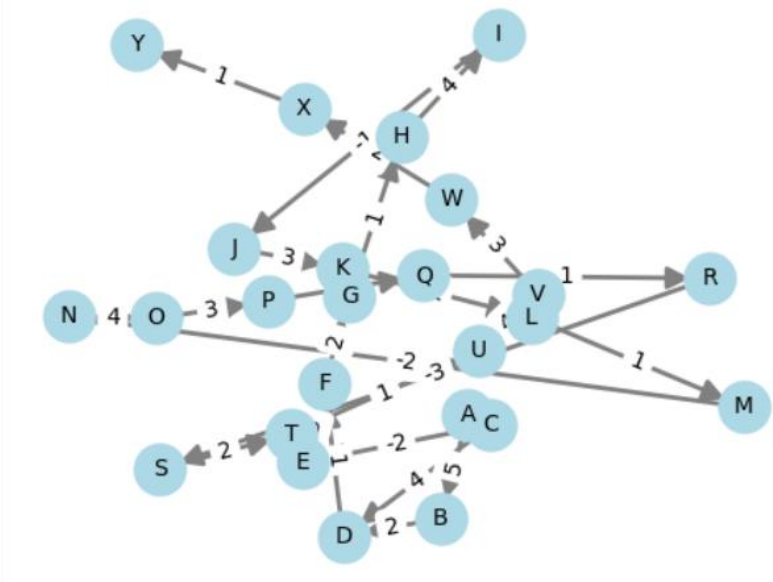


Figure 3.2.2.a Graph(A-Y)

Apply the Bellman-Ford algorithm to find the shortest path from A to Y.

- Determine the minimum time required to reach node Y from node A.
- Identify the nodes in the shortest path.

This graph contains negative weight edges, but it does not contain any negative cycles.

CODE IMPLEMENTATION

```
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = []

    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])

    def bellman_ford(self, src):
        dist = [float("Inf")] * self.V
        dist[src] = 0
        parent = [-1] * self.V

        for _ in range(self.V - 1):
            for u, v, w in self.graph:
                if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                    dist[v] = dist[u] + w
                    parent[v] = u

        for u, v, w in self.graph:
            if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                print("Graph contains negative weight cycle")
                return

        return dist, parent

    def print_path(self, parent, j):
        if parent[j] == -1:
            print(chr(j + 65), end=" ")
            return
        self.print_path(parent, parent[j])
        print(chr(j + 65), end=" ")

# Create a graph with 25 vertices (A to Y)
g = Graph(25)

# Add edges to the graph
edges = [
    (0, 1, 5), (0, 2, -3), (1, 3, 2), (2, 3, 4), (2, 4, -2),
    (3, 5, 1), (4, 5, 3), (5, 6, 2), (6, 7, 1), (7, 8, 4),
    (8, 9, -1), (9, 10, 3), (10, 11, 2), (11, 12, 1), (12, 13, -2),
```

```
    (13, 14, 4), (14, 15, 3), (15, 16, 2), (16, 17, 1), (17, 18, -3),
    (18, 19, 2), (19, 20, 1), (20, 21, 4), (21, 22, 3), (22, 23, 2),
    (23, 24, 1)
]

for u, v, w in edges:
    g.add_edge(u, v, w)

# Apply Bellman-Ford algorithm
dist, parent = g.bellman_ford(0)

# Print shortest distance and path from A to Y
print("Shortest distance from A to Y:", dist[24])
print("Shortest path from A to Y: ", end="")
g.print_path(parent, 24)
```

Shortest distance from A to Y: 28 km

Shortest path from A to Y: A C E F G H I J K L M N O P Q R S T U V W X Y

Process finished with exit code 0.

HYPOTHETICAL EXAMPLE 2: TRAFFIC-AWARE SHORTEST PATH PROBLEM

The find shortest path from node A to node T in a graph with 20 nodes, where the edge weights represent travel times that vary according to traffic conditions. The graph is as follows:

Nodes: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T

Edges with weights (travel times in minutes) that vary according to traffic conditions:

Table 3.2.2.a. Traffic Aware Edge Weights

EDGE	Weight (Low Traffic)	Weight (Moderate Traffic)	Weight (high Traffic)
A - B	5	10	20
A - C	3	6	12
B - D	4	8	16
B - E	2	4	8
C - F	6	12	24
D - G	3	6	12
E - H	5	10	20
F- I	4	8	16
G - J	2	4	8
H - K	6	12	24
I - L	3	6	12
J - M	5	10	20
K - N	4	8	16

L - O	2	4	8
M - P	6	12	24
N - Q	3	6	12
O - R	5	10	20
P - S	4	8	16
Q - T	2	4	8
R - S	3	6	12
S - T	5	10	20

The traffic conditions are as follows:

Low traffic: 00:00 - 07:00 and 19:00 - 23:59

Moderate traffic: 07:00 - 10:00 and 16:00 - 19:00

Heavy traffic: 10:00 - 16:00

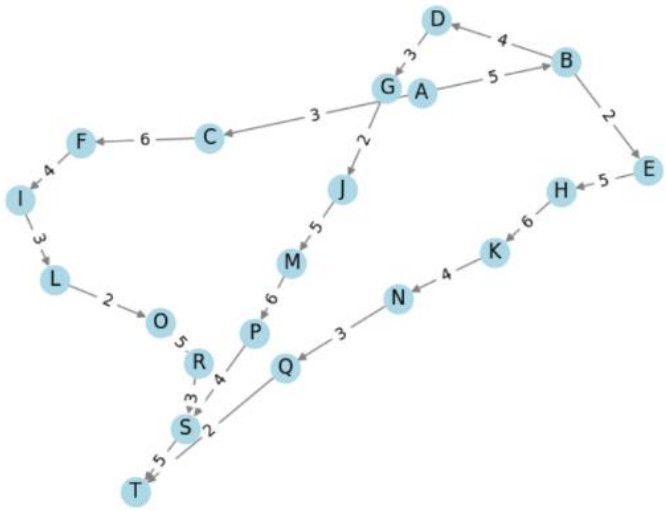


Figure 3.2.2.b Network Graph(A-T)

To find the shortest path for all three traffic conditions, we'll need to run the following algorithm for each traffic condition.

```
def bellman_ford(graph, source):
    distance = {node: float('inf') for node in graph}
    distance[source] = 0
    predecessor = {node: None for node in graph}

    for _ in range(len(graph) - 1):
        for node in graph:
            for neighbor, weight in graph[node].items():
                if distance[node] != float('inf') and distance[node] + weight < distance[neighbor]:
                    distance[neighbor] = distance[node] + weight
                    predecessor[neighbor] = node
```

```
    for node in graph:
        for neighbor, weight in graph[node].items():
            if distance[node] != float('inf') and distance[node] + weight < distance[neighbor]:
                print("Negative cycle detected")
                return None

    return distance

# Define the graphs for different traffic conditions
graph_low_traffic = {
    'A': {'B': 5, 'C': 3},
    'B': {'D': 4, 'E': 2},
    'C': {'F': 6},
    'D': {'G': 3},
    'E': {'H': 5},
    'F': {'I': 4},
    'G': {'J': 2},
    'H': {'K': 6},
    'I': {'L': 3},
    'J': {'M': 5},
    'K': {'N': 4},
    'L': {'O': 2},
    'M': {'P': 6},
    'N': {'Q': 3},
    'O': {'R': 5},
    'P': {'S': 4},
    'Q': {'T': 2},
    'R': {'S': 3},
    'S': {'T': 5},
    'T': {}
}

graph_moderate_traffic = {
    'A': {'B': 10, 'C': 6},
    'B': {'D': 8, 'E': 4},
    'C': {'F': 12},
    'D': {'G': 6},
    'E': {'H': 10},
    'F': {'I': 8},
    'G': {'J': 4},
    'H': {'K': 12},
    'I': {'L': 6},
    'J': {'M': 10},
    'K': {'N': 8},
    'L': {'O': 4},
    'M': {'P': 12},
    'N': {'Q': 6},
    'O': {'R': 10},
    'P': {'S': 8},
    'Q': {'T': 4},
```

```
    'R': {'S': 6},
    'S': {'T': 10},
    'T': {}
}

graph_heavy_traffic = {
    'A': {'B': 20, 'C': 12},
    'B': {'D': 16, 'E': 8},
    'C': {'F': 24},
    'D': {'G': 12},
    'E': {'H': 20},
    'F': {'I': 16},
    'G': {'J': 8},
    'H': {'K': 24},
    'I': {'L': 12},
    'J': {'M': 20},
    'K': {'N': 16},
    'L': {'O': 8},
    'M': {'P': 24},
    'N': {'Q': 12},
    'O': {'R': 20},
    'P': {'S': 16},
    'Q': {'T': 8},
    'R': {'S': 12},
    'S': {'T': 20},
    'T': {}
}

source_node = 'A'

distances_low_traffic = bellman_ford(graph_low_traffic, source_node)
distances_moderate_traffic = bellman_ford(graph_moderate_traffic, source_node)
distances_heavy_traffic = bellman_ford(graph_heavy_traffic, source_node)

print("Shortest distance from node A to node T in low traffic:", distances_low_traffic['T'])
print("Shortest distance from node A to node T in moderate traffic:",
distances_moderate_traffic['T'])
print("Shortest distance from node A to node T in heavy traffic:",
distances_heavy_traffic['T'])
```

```
Shortest distance from node A to node T in low traffic: 27
Shortest distance from node A to node T in moderate traffic: 54
Shortest distance from node A to node T in heavy traffic: 108

Process finished with exit code 0
```

Figure 3.2.2.c Result

RESULT:

Shortest distance from node A to node T in low traffic: 27 MINUTES

Shortest distance from node A to node T in moderate traffic: 54 MINUTES

Shortest distance from node A to node T in heavy traffic: 108 MINUTES

CONCLUSION

In conclusion, the Bellman-Ford algorithm to determine the optimal ambulance route to the hospital, with A4 being the best choice with a total distance of 1.35 km. The graph visualizations created throughout the project effectively showcased the connections between ambulances, College C, hospitals, and other nodes, highlighting the distances between them. These findings demonstrate the potential for route optimization in emergency response scenarios, enabling informed decisions for efficient ambulance deployment. The codes developed in this project can be further enhanced and applied to real-world emergency response systems.

3.2.3 A* SEARCH ALGORITHM

Efficient placement and routing of emergency response facilities such as hospitals, fire stations, and ambulance dispatch units is critical in reducing response times and saving lives. This project explores the application of the A* (A-star) Search Algorithm for optimizing emergency response locations. A* combines heuristic-driven search with exact path cost estimation, making it well-suited for navigating urban road networks where speed and accuracy are essential. Through a city grid case study, this report demonstrates how A* can minimize travel times between emergency service stations and demand points, ensuring better accessibility and resource utilization.

The *A* algorithm* is a widely used pathfinding algorithm, designed to find the most efficient route between two points. It improves on earlier algorithms like **Dijkstra's Algorithm** by incorporating a **heuristic** that estimates the cost of reaching the goal from any given node, making it more efficient in terms of computational time.

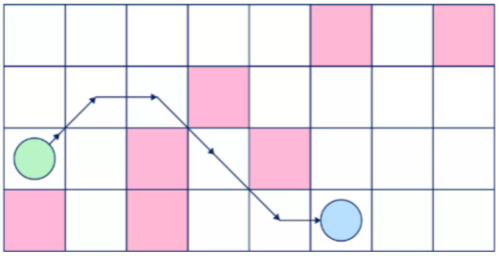


*Figure 3.2.3.a A*Search Grid Example*

Methodology

A* prioritizes nodes that are likely to lead to the goal by balancing two factors:

1. **g(n)**: The actual cost of reaching node **n** from the start.
2. **h(n)**: The heuristic estimate of the cost to reach the goal from node **n**.
3. **f(n)**: The estimated total cost from the start node to the end goal via node n.

This combination allows A* to be both optimal and efficient, making it the go-to algorithm for applications that require rapid pathfinding with high accuracy.

The A* algorithm is one of the most widely used pathfinding and graph traversal techniques due to its efficiency and accuracy in finding the shortest path. It begins with initialization, where the start and goal nodes are defined. Two lists are maintained: the open list, which contains nodes to be evaluated, and the closed list, which holds nodes already evaluated. The start node is placed in the open list, with its cost from the start (g(n)) set to zero and its heuristic estimate (h(n)) calculated toward the goal.

The algorithm then enters its main loop, where it repeatedly selects the node from the open list with the lowest total cost, f(n) = g(n) + h(n). This choice ensures that the search is guided not only by the path already taken but also by an estimate of the remaining distance. Once a node is selected, it is moved to the closed list, and its neighbours are evaluated. For each neighbour, a tentative g(n) is computed, representing the cost to reach that node via the current path. If the neighbour is not in the open list, it is added with updated cost values. If it already exists in the open list but a better (lower-cost) path is found, its values and parent reference are updated.

At every step, the algorithm carefully maintains and updates three cost functions: g(n), the exact cost from the start to a node; h(n), the heuristic estimate to the goal; and f(n), the sum of the two. This evaluation process continues until the goal node is reached, at which point the algorithm reconstructs the optimal path by tracing back through the parent nodes. If the open list becomes empty before reaching the goal, it means no valid path exists.

A critical element of A* is the heuristic function h(n), which guides the search process. A well-designed heuristic allows the algorithm to prioritize nodes that are more likely to lead toward the goal, making it significantly faster than uninformed searches. Importantly, the heuristic must be admissible, meaning it should never overestimate the actual cost to the goal, ensuring that A* always finds the shortest path. Without heuristics, A* reduces to Dijkstra's Algorithm, which explores paths exhaustively.

The A* algorithm combines the strengths of Dijkstra's uniform cost search and heuristic-guided greedy search, achieving both optimality and efficiency. Its ability to balance real path costs with intelligent estimates makes it highly suitable for applications such as navigation, robotics, and emergency response optimization.

Common Heuristics in A*

Several heuristics can be used in A*, depending on the structure of the problem:

1. Manhattan Distance:

Used in grid-based problems, where movements are restricted to horizontal and vertical directions.

Formula:

$$h(n) = |x_{goal} - x_{current}| + |y_{goal} - y_{current}|$$

2. Euclidean Distance:

Used when movement is allowed in any direction, accounting for the straight-line distance between two points.

Formula:

$$h(n) = \sqrt{(x_{goal} - x_{current})^2 + (y_{goal} - y_{current})^2}$$

3. Diagonal Distance:

Suitable for grid-based systems that allow diagonal movements.

Formula:

$$h(n) = \max(|x_{goal} - x_{current}|, |y_{goal} - y_{current}|)$$

Choosing the Right Heuristic

Manhattan Distance is ideal for problems with grid-based layouts and no diagonal movements.
Euclidean Distance is suitable when diagonal or any-direction movement is allowed.
Diagonal Distance is used for grid problems with diagonal movements allowed.
Choosing the right heuristic depends on the **nature of the environment** and the **problem** at hand.

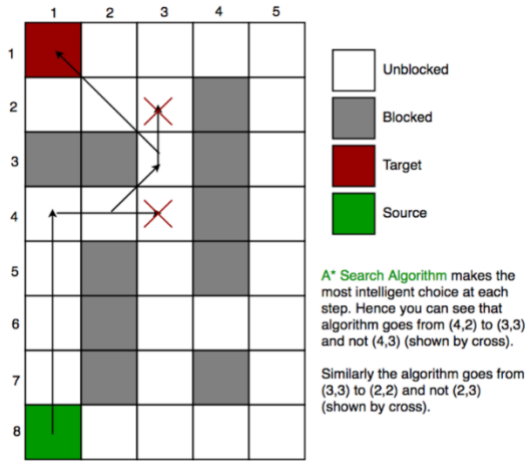


Figure 3.2.3.b Heuristics Grid Example

HYPOTHETICAL EXAMPLE:

A natural disaster has occurred in a city, and emergency responders need to reach different affected areas. The city has a complex road network with multiple intersections, varying road conditions, and traffic congestion. The emergency responders are located at node R, and they need to reach nodes A, B, C, D, E, F, G, H, and I. The road network is represented by a weighted graph, where the weights represent the time, it takes to travel between nodes. The weighted

graph is represented by the following adjacency list:

- R -> A (weight: 5)
- R -> B (weight: 3)
- R -> C (weight: 2)
- A -> D (weight: 4)
- A -> E (weight: 2)
- B -> D (weight: 1)
- B -> F (weight: 5)
- C -> E (weight: 3)
- C -> G (weight: 2)
- D -> H (weight: 2)
- E -> H (weight: 4)

- E -> I (weight: 1)

- F -> I (weight: 3)

- G -> I (weight: 5)

- H -> I (weight: 2)

Additionally, some roads have traffic congestion, which increases the travel time. The traffic congestion is represented by the following: Road A -> D has a 50% chance of traffic congestion, which increases the travel time by 2 units. Road B -> F has a 30% chance of traffic congestion, which increases the travel time by 1 unit. Road E -> I have a 20% chance of traffic congestion, which increases the travel time by 1 unit. Using python, we can find the shortest expected path from node R to nodes A, B, C, D, E, F, G, H, and I, considering the traffic congestion.

RESULT

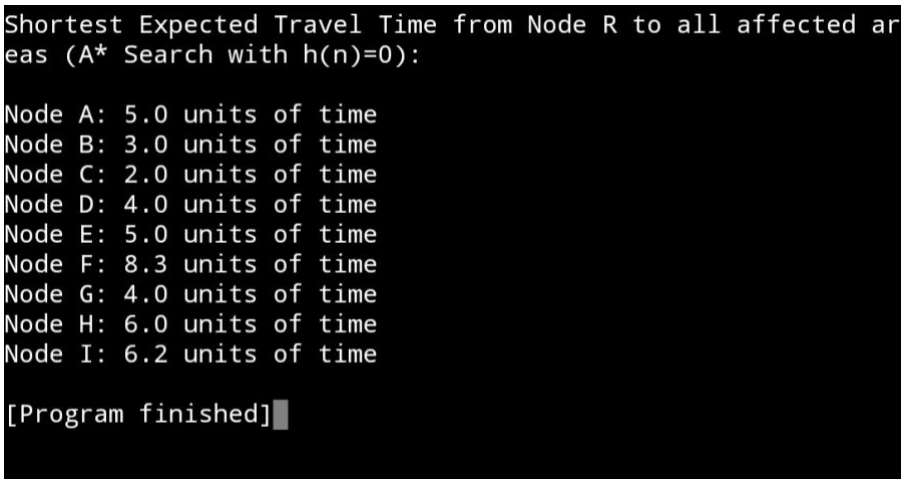


Figure 3.2.3.c Result

CONCLUSION AND FUTURE SCOPE

The A* Search Algorithm proves to be a powerful tool in optimizing emergency response locations and routing. Its heuristic-driven approach ensures both accuracy and efficiency in pathfinding. In real-world applications, A* can be integrated with Geographic Information Systems (GIS) and real-time traffic data to dynamically update routes and facility allocations. Future research may extend this model to multi-agent coordination, robust optimization under uncertainty, and integration with machine learning for demand prediction.

3.2.4 FLOYD–WARSHALL ALGORITHM

The Floyd–Warshall algorithm is a dynamic programming algorithm designed to find shortest paths between all pairs of vertices in a weighted graph. It is particularly effective for scenarios where positive and negative edge weights might be present, but no negative cycles exist. It

works on both directed and undirected graphs. It systematically updates shortest paths by checking whether an intermediate vertex provides a shorter route between two vertices.

METHODOLOGY

The following steps are applied to implement the Floyd–Warshall algorithm for optimizing emergency response routes:

1. Graph Construction: Represent the emergency response network (ambulances, college, hospitals) as a weighted graph, where nodes represent locations and edges represent distances.

2. Adjacency Matrix Initialization: Create a distance matrix where each entry dist[i][j] represents the distance between node i and j. If no direct path exists, initialize it as infinity.

3. Path Reconstruction: Maintain a 'next' matrix to reconstruct the exact sequence of nodes in the shortest path.

5. Optimization and Analysis: Use the results to identify the nearest hospital for each ambulance via the college, ensuring the fastest response route in the network.

FLOYD-WARSHALL ALGORITHM EQUATION

$$D_{[i][j]} = min(D_{[i][j]}, D_{[i][k]} + D_{[k][j]})$$

Where:

$D_{[i][j]}$ is the current shortest path distance from vertex i to vertex j

$D_{[i][k]}$ is the current shortest path distance from vertex i to intermediate vertex k

$D_{[k][j]}$ is the current shortest path distance from intermediate vertex k to vertex j

HYPOTHETICAL EXAMPLE TRAFFIC-AWARE SHORTEST PATH PROBLEM

The find shortest path from node A to node T in a graph with 20 nodes, where the edge weights represent travel times that vary according to traffic conditions. The graph is as follows:

Nodes: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T

Edges with weights (travel times in minutes) that vary according to traffic conditions:

Table 3.2.4.a Traffic Aware Edge Weights

EDGE	Weight (Low Traffic)	Weight (Moderate Traffic)	Weight (high Traffic)
A - B	5	10	20
A - C	3	6	12
B - D	4	8	16

B - E	2	4	8
C - F	6	12	24
D - G	3	6	12
E - H	5	10	20
F- I	4	8	16
G - J	2	4	8
H - K	6	12	24
I - L	3	6	12
J - M	5	10	20
K - N	4	8	16
L - O	2	4	8
M - P	6	12	24
N - Q	3	6	12
O - R	5	10	20
P - S	4	8	16
Q - T	2	4	8
R - S	3	6	12
S - T	5	10	20

The traffic conditions are as follows:

Low traffic: 00:00 - 07:00 and 19:00 - 23:59

Moderate traffic: 07:00 - 10:00 and 16:00 - 19:00

Heavy traffic: 10:00 - 16:00

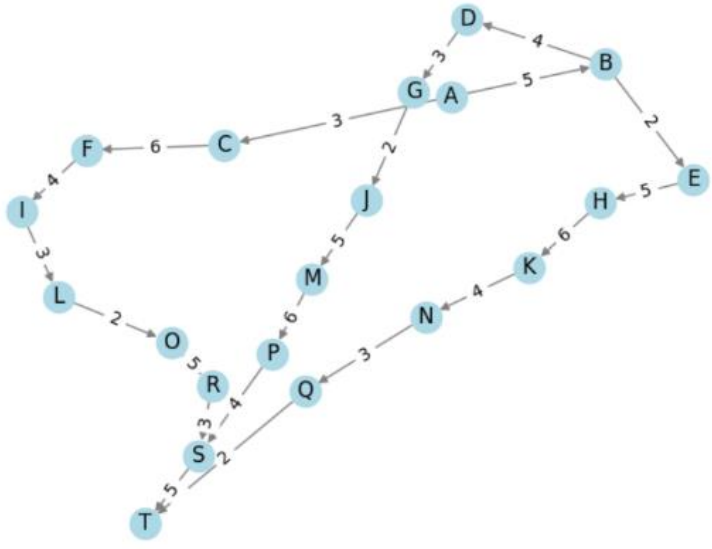


Figure 3.2.4.a Network Graph (A-T)

To find the shortest path for all three traffic conditions, we'll need to run the following algorithm for each traffic condition.

CODE

```
def floyd_warshall(graph):
    nodes = list(graph.keys())
    distance_matrix = {node: {neighbor: float('inf') for neighbor in nodes} for node in nodes}

    for node in nodes:
        distance_matrix[node][node] = 0
        for neighbor, weight in graph[node].items():
            distance_matrix[node][neighbor] = weight

    for intermediate in nodes:
        for start in nodes:
            for end in nodes:
                distance_matrix[start][end] = min(distance_matrix[start][end],
distance_matrix[start][intermediate] + distance_matrix[intermediate][end])

    return distance_matrix

# Define the graphs for different traffic conditions
graph_low_traffic = {
    'A': {'B': 5, 'C': 3},
    'B': {'D': 4, 'E': 2},
    'C': {'F': 6},
    'D': {'G': 3},
    'E': {'H': 5},
    'F': {'I': 4},
    'G': {'J': 2},
    'H': {'K': 6},
    'I': {'L': 3},
    'J': {'M': 5},
    'K': {'N': 4},
    'L': {'O': 2},
    'M': {'P': 6},
    'N': {'Q': 3},
    'O': {'R': 5},
    'P': {'S': 4},
    'Q': {'T': 2},
    'R': {'S': 3},
    'S': {'T': 5},
    'T': {}
}

graph_moderate_traffic = {
    'A': {'B': 10, 'C': 6},
    'B': {'D': 8, 'E': 4},
    'C': {'F': 12},
    'D': {'G': 6},
    'E': {'H': 10},
    'F': {'I': 8},
```

```
    'G': {'J': 4},
    'H': {'K': 12},
    'I': {'L': 6},
    'J': {'M': 10},
    'K': {'N': 8},
    'L': {'O': 4},
    'M': {'P': 12},
    'N': {'Q': 6},
    'O': {'R': 10},
    'P': {'S': 8},
    'Q': {'T': 4},
    'R': {'S': 6},
    'S': {'T': 10},
    'T': {}
}

graph_heavy_traffic = {
    'A': {'B': 20, 'C': 12},
    'B': {'D': 16, 'E': 8},
    'C': {'F': 24},
    'D': {'G': 12},
    'E': {'H': 20},
    'F': {'I': 16},
    'G': {'J': 8},
    'H': {'K': 24},
    'I': {'L': 12},
    'J': {'M': 20},
    'K': {'N': 16},
    'L': {'O': 8},
    'M': {'P': 24},
    'N': {'Q': 12},
    'O': {'R': 20},
    'P': {'S': 16},
    'Q': {'T': 8},
    'R': {'S': 12},
    'S': {'T': 20},
    'T': {}
}

# Add missing edges with infinity weight
nodes = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T']
for graph in [graph_low_traffic, graph_moderate_traffic, graph_heavy_traffic]:
    for node in nodes:
        if node not in graph:
            graph[node] = {}
        for neighbor in nodes:
            if neighbor not in graph[node]:
                graph[node][neighbor] = float('inf')

distance_matrix_low_traffic = floyd_warshall(graph_low_traffic)
```

```
distance_matrix_moderate_traffic = floyd_warshall(graph_moderate_traffic)
distance_matrix_heavy_traffic = floyd_warshall(graph_heavy_traffic)

print("Shortest distance from node A to node T in low traffic:",
distance_matrix_low_traffic['A']['T'])
print("Shortest distance from node A to node T in moderate traffic:",
distance_matrix_moderate_traffic['A']['T'])
print("Shortest distance from node A to node T in heavy traffic:",
distance_matrix_heavy_traffic['A']['T'])
```

RESULT:

Shortest distance from node A to node T in low traffic: 27

Shortest distance from node A to node T in moderate traffic: 54

Shortest distance from node A to node T in heavy traffic: 108

CONCLUSION for chapter 3: ROUTE OPTIMIZATION USING GRAPH THEORY

Dijkstra's algorithm is a well-known graph algorithm that can be used to find the shortest path between emergency response units and incident locations. By considering factors like traffic congestion, road closures, and terrain, Dijkstra's algorithm can provide the quickest route to a hospital or a fire scene. For instance, emergency responders can use Dijkstra's algorithm to determine the optimal route to a medical emergency, reducing response times and improving patient outcomes. The Bellman-Ford algorithm is particularly useful in scenarios where some roads have negative weights, such as roads that are closed or have penalties. By using Bellman-Ford, emergency responders can find the shortest path to an incident location while avoiding roads that are closed due to construction or natural disasters. This algorithm can be applied to various emergency response scenarios, including ambulance routing and fire department routing. The A* algorithm is well-suited for emergency response route optimization due to its informed search nature and ability to guide the search towards the goal node efficiently. By using a heuristic function that estimates the distance from each node to the incident location, A* can find the shortest path quickly and efficiently. A* can be applied to various emergency response scenarios, including search and rescue operations, where the goal is to find the most efficient route to a search area. The application of graph algorithms to emergency response route optimization has several potential benefits, including:

Ambulance routing: Emergency medical services (EMS) can use these algorithms to determine the quickest route to a medical emergency, considering factors like traffic congestion and road conditions.

Fire department routing: Fire departments can use these algorithms to determine the quickest route to a fire scene, considering factors like traffic congestion, road closures, and terrain.

Search and rescue operations: Search and rescue teams can use these algorithms to determine the most efficient route to a search area, considering factors like terrain, weather conditions, and obstacles.

By applying graph algorithms and leveraging real-time data and analytics, emergency responders can optimize their routes and respond to emergencies more effectively. This can potentially save lives and reduce response times. Further research is needed to explore the

application of these algorithms in various emergency response scenarios and to develop more efficient and effective route optimization strategies.

CHAPTER 4:
CASE STUDY

4.1: RESOURCE ALLOCATION USING LINEAR PROGRAMMING

4.1.1 THE SET COVERING LOCATION PROBLEM (SCLP)

Demand Points

- ST Teresa's college
- Vytilla junction
- Panampilly nagar
- Kadavanthra
- Ernakulam south
- MG road

Facilities

- Medical trust, Ernakulam
- Lourdes hospital, Ernakulam
- Aster medicity, Ernakulam
- Lisie hospital, Ernakulam
- Ernakulam general hospital
- Ernakulam medical centre

The optimization problem is:

Select the minimum number of facilities needed to cover all demand points within the response time limit.

DISTANCE [KM]	MEDICAL TRUST	LOURDES HOSPITAL	ASTER MEDICITY	LISIE HOSPITAL	ERN GENERAL HOSPITAL	ERN MEDICAL CENTRE
ST. TERESA'S	3.1	4.3	13	3.2	1.3	7.7
VYTTILA JUNCTION	4	8.7	14	6.3	6.5	4.6
PANAMPILLY NAGAR	1.6	6.9	17	5.3	3.5	7.9
KADAVANTHRA	3	7.7	18	5.7	5.5	8.1
ERNAKULAM SOUTH	0.9	5.1	10	3.6	1.7	8.4
MG ROAD	2.3	3.4	8.3	2.4	2.3	6

Figure 4.1.1.a: Distance [km] Table

Figure 4.1.1.a show the travel distances from potential facilities (hospitals) to demand points in kilometers. This table serves as the initial data for the SCLP model, listing the distances between 6 demand points and 6 facilities

Service Parameters:

Speed: 40 km/h (typical emergency vehicle speed in urban areas)

Max Response Time: 8 minutes (industry standard for emergency services)

Max Coverage Distance = 40 km/h × (8 min ÷ 60 min/h) = 5.33 km

If (distance <= 5.33), then 1, otherwise 0

SOLUTION AND ANALYSIS

	MEDICAL TRUST	LOURDES HOSPITAL	ASTER MEDICITY	LISIE HOSPITAL	ERN GENERAL HOSPITAL	ERN MEDICAL CENTRE			
Select	1	0	0	0	0	0	1		
ST. TERESA'S	1	1	0	1	1	0	1	>=	1
VYTTILA JUNCTION	1	0	0	0	0	1	1	>=	1
PANAMPILLY NAGAR	1	0	0	1	1	0	1	>=	1
KADAVANTHRA	1	0	0	0	0	0	1	>=	1
ERNAKULAM SOUTH	1	1	0	1	1	0	1	>=	1
MG ROAD	1	1	0	1	1	0	1	>=	1

Figure 4.1.1.b: Solution Table

Figure 4.1.1.b: shows the binary coverage status (1 if covered, 0 if not) based on the calculated maximum coverage distance (5.33 km). Binary Coverage Matrix: A '1' indicates the distance is ≤5.33 km (covered), and '0' indicates it's >5.33 km (not covered). The solution row shows that Medical Trust, Ernakulam is selected.

Optimal Solution Found:

Facilities Selected: MEDICAL TRUST, ERNAKULAM

Total Facilities: 6

Coverage: 6/6 demand points (100.0%)

Efficiency: This solution minimizes the number of facilities while ensuring complete coverage.

4.1.2 MAXIMUM COVERING LOCATION PROBLEM (MCLP)

The Kochi Municipal Corporation is planning to establish a network of ambulance stations to provide emergency medical services to the city's residents. Due to budget constraints, they can only establish 2 ambulance stations from 6 potential locations. The goal is to maximize the population coverage while ensuring that emergency response time does not exceed 10 minutes to any covered neighborhood.

Kochi has been divided into 6 major residential neighborhoods with varying population densities. The city has identified 6 potential locations for ambulance stations, each strategically positioned near major hospitals or medical facilities.

GIVEN DATA:

Potential Ambulance Station Locations:

1. Medical Trust Hospital, Ernakulam
2. Lourdes Hospital, Ernakulam
3. Aster Medcity, Ernakulam
4. Akshaya hospital kadavanthara, Ernakulam
5. Welcare hospital Panampilly Nagar, Ernakulam
6. DG medical centre MG Road, Ernakulam

Neighborhoods (Demand Points)	Population
MG Road (Junction)	9992
Panampilly Nagar	15973
Ernakulam South	12249
Marine Drive	15790
Kadavanthra	28752
Vyttila Junction	38134

Figure 4.1.2.a: Neighbourhoods (Demand Points)-Population Table

Above figure list the residential neighborhoods, which are the demand points, and their associated population weights. It lists 6 neighborhoods with populations ranging from 9,992 to 38,134, totaling 120,890. This is the value to be maximized.

Travel Time Matrix (minutes)						
From/To	Medical Trust	Lourdes Hospital	Aster Medcity	Akshaya hospital kadavanthara	Welcare hospital Panampilly Nagar	DG medical centre MG Road
MG ROAD	6	7	20	6	9	5
PANAMPILLY NAGAR	4	14	27	4	6	5
ERNAKULAM SOUTH	3	10	23	5	7	2
MARINE DRIVE	5	8	21	8	10	5
KADAVANTHARA	7	15	28	4	6	6
VYTTILA JUNCTION	8	16	26	6	2	8

Figure 4.1.2.b: Travel Time Matrix (minutes) Table

Figure 4.1.2.b shows the travel time from each potential ambulance station location to each neighborhood (demand point). It provides the travel time in minutes between 6 neighborhoods and 6 potential station locations and is used to check the ≤10 minutes constraint.

SOLUTION

Decision Variables		coverage status	MAX RESPONSE TIME	10
Potential Station Location	Open (1) / Not Open (0)	NBD	COVERED (1)/NOT COVERED(0)	
Medical Trust	0	MG ROAD	1	
Lourdes Hospital	0	PANAMPILLY NAGAR	1	
Aster Medcity	0	ERN SOUTH	1	
Akshaya hospital kadavanthara	0	MARINE DRIVE	1	
Welcare hospital Panampilly Nagar	1	KADAVANTHRA	1	
DG medical centre MG Road	1	VYTTILA JUNC	1	

Figure 4.1.2.c: Solution Table

Figure 4.1.2.c shows the final decision variables (open/not open) for each potential station and the coverage status of the neighborhoods.

Decision Variables: Indicates that Welcare hospital Panampilly Nagar (1) and DG medical centre MG Road (1) were selected as the two open stations.

TOTAL STATIONS OPEN	2
TOTAL POPULATION COVERED	120890
COVERAGE PERCENTAGE	1

Figure 4.1.2.d: Result

The above figure confirms that 2 total stations open and 120,890 total population covered (100% coverage)

RESULTS AND DISCUSSION

Optimal Station Locations:

- Welcare hospital Panampilly Nagar
- DG medical centre MG Road

COVERAGE WITH WELCARE HOSPITAL PANAMPILLY NAGAR + DG MEDICAL CENTRE MG ROAD:

Neighborhood	Welcare hospital Panampilly Nagar Time	DG medical centre MG Road Time	Min Time	Covered	Population
MG Road Junction	9	5	5≤10	yes	9992
Panampilly Nagar	4	5	5≤10	yes	15973

Ernakulam South	7	2	2≤10	yes	12249
Marine Drive	10	5	5≤10	yes	15790
Kadavanthra	6	6	6≤10	yes	28752
Vyttila Junction	2	8	2≤10	yes	38134

Table 4.1.2.a: Coverage with Selected Hospitals Table

Total Coverage = 120890 [100%]

4.1.3 MIXED-INTEGER LINEAR PROGRAMMING (MILP)

There are 5 facilities and 5 demand points. Data of demand for each of the demand points are given. Fixed cost for ambulance services up to 20 km of any trip is 2500 INR and extra cost per additional km is 50 INR. The calculated extra cost from demand point to facility is given. Capacity of each of the supply points = 20. Minimize the total cost of satisfying all service demand. Determine which facilities should be used and the optimal allocation of service units from the used facilities to demand points. All demand must be met and facility capacity is not exceeded.

Demand points

- ALUVA
- CHERANALLOOR
- KUMBALANGI
- THRIPPUNITHURA
- CHERAI

Facilities

- MEDICAL TRUST, ERNAKULAM
- LOURDES HOSPITAL, ERNAKULAM
- ASTER MEDICITY, ERNAKULAM
- LISIE HOSPITAL, ERNAKULAM
- ERN MED CENTRE, ERNAKULAM

FACILITIES	EXTRA COST PER ADDITIONAL KM					FIXED COST (INR/20KM)	CAPACITY
	ALUVA	CHERANALLOOR	KUMBALANGI	THRIPPUNITHURA	CHERAI		
MEDICAL TRUST	100	1250	0	0	250	2500	20
LOURDES HOSPITAL	700	1050	0	0	150	2500	20
ASTER MEDICITY	150	850	850	950	50	2500	20
LISIE HOSPITAL	750	950	850	0	250	2500	20
ERN MED CENTRE	200	850	0	0	300	2500	20
DEMAND	12	8	14	16	7	57	

Figure 4.1.3.a: Demand Points Transport Cost Table

Figure 4.1.3.a, contains the input parameters: 5×5 extra transport cost per additional km matrix, INR 2500 fixed cost for ambulance services up to 20 km per facility, 20 unit capacity per facility, and the demand for each point (total demand: 57).

SOLUTION

DECISION VARIABLES

FACILITIES	DEMAND POINTS					FACILITY OPEN/NOT OPEN	
	ALUVA	CHERANALLOOR	KUMBALANGI	THRIPPUNITHURA	CHERAI		
MEDICAL TRUST	10	0	10	0	0	1	20
LOURDES HOSPITAL	0	0	4	16	0	1	20
ASTER MEDICITY	2	8	0	0	7	1	20
LISIE HOSPITAL	0	0	0	0	0	0	
ERN MED CENTRE	0	0	0	0	0	0	

Figure 4.1.3.b: Decision Variables Table

Figure 4.1.3.b shows the final facility utilization (how much demand is shipped from which facility to which demand point) and the facility open/not open decision.

FACILITIES USED: MEDICAL TRUST, LOURDES HOSPITAL, ASTER MEDICITY

- ALUVA with demand 12 is partially satisfied by MEDICAL TRUST (10) and partially satisfied by ASTER MEDICITY (2).
- CHERANALLOOR with demand 8 is satisfied by ASTER MEDICITY.
- KUMBALANGI with demand 14 is partially satisfied by MEDICAL TRUST (10) and partially satisfied by LOURDES HOSPITAL (4).
- THRIPPUNITHURA with demand 16 is satisfied by LOURDES HOSPITAL.
- CHERAI with demand 7 is satisfied by ASTER MEDICITY.

CONSTRAINTS

FACILITIES	EXCESS CAPCITY
MEDICAL TRUST	0
LOURDES HOSPITAL	0
ASTER MEDICITY	3
LISIE HOSPITAL	0
ERN MED CENTRE	0

	ST TERESA'S	VYTTILA JUNCTION	PANAMPILLY NAGAR	KADAVANTHRA	MG ROAD
UNMET DEMAND	0	0	0	0	0

Figure 4.1.3.c: Constraints Table

ASTER MEDICITY has Excess capacity of 3 unit. All 5 demand points are met.

OBJECTIVE FUNCTION

Z =	**15950**

Figure 4.1.3.d: Objective Function (Z) Result

Figure 4.1.3.d states the final minimized total cost (Objective Function value).The optimal total cost (Z) is 15,950.

4.2: ROUTE OPTIMIZATION USING GRAPH THEORY

4.2.1: FINDING SHORTEST DISTANCE PATH USING DIJKSTRA'S ALGORITHM

This section presents a case study on route optimization using Dijkstra's algorithm. The objective is to find the shortest path from St Teresa's College- science block, to the nearest emergency services, including hospitals, fire stations, ambulance stations, and police stations, in the event of a disaster.

Location: St Teresa's College is located in Ernakulam, Kerala.

Nearest Emergency Services: The following emergency services are located near St Teresa's College:

- Hospitals:
    1. General hospital, Ernakulam
    2. Medical Trust Hospital, Pallimuk, Ernakulam
    3. Lisie Hospital, North, Ernakulam
    4. Sree Sudheendra Medical Mission Hospital, Ernakulam
    5. Lakshmi Hospital, Pallimuk, Ernakulam
    6. Amrita Hospital, Edappally, Ernakulam

- Fire stations:
    1. Fire and rescue station, Gandhinagar
    2. Fire force, Kerala, marine drive
    3. Fire station, cochin shipyard

- Ambulance services:
    1. Cochin Emergency Ambulance Service, South, Ernakulam
    2. Medon Health Care Ambulance Service, South Ernakulam,
    3. Ambulance Service On Time, South, Ernakulam
    4. We One Ambulance Service, Shenoys, Ernakulam

    (St. Teresa's College has an on-campus clinic with 24/7 emergency services)

- Police stations:
    1. Central Police Station, High Court, Ernakulam
    2. Commissioner Of Police Kochi City, Marine Drive
    3. Ernakulam Town North Police Station, Chittoor
    4. Ernakulam Town South Police Station, Thevara

A disaster has occurred at St Teresa's College science block [C], and ambulances from A1, A2, A3, and A4 need to respond to the emergency. The ambulances are located at different positions and need to reach College C first, and then proceed to the nearest hospital (H1, H2, H3, H4, or H5). Dijkstra's algorithm can be used to optimize the routes for emergency responders to reach in the event of the disaster.

NODES	LOCATION	LOCATION TYPE
C	ST TERESAS COLLEGE, ERNAKULAM - SCIENCE BLOCK	INSTITUTION
H1	GENERAL HOSPITAL, ERNAKULAM	HOSPITAL
H2	MEDICAL TRUST HOSPITAL, PALLIMUK	HOSPITAL
H3	LISIE HOSPITAL, NORTH	HOSPITAL
H4	SREE SUDHEENDRA MEDICAL MISSION HOSPITAL, EKM	HOSPITAL
H5	LAKSHMI HOSPITAL, PALLIMUK	HOSPITAL
H6	AMRITA HOSPITAL, EDAPPALLY, ERNAKULAM	HOSPITAL
F1	FIRE AND RESCUE STATION, GANDHINAGAR	FIRE STATION
F2	FIRE FORCE, KERALA, MARINE DRIVE	FIRE STATION
F3	FIRE STATION, COCHIN SHIPPIYARD	FIRE STATION
A1	COCHIN EMERGENCY AMBULANCE SERVICE,EKM SOUTH	AMBULANCE STATION
A2	MEDON HEALTHCARE EMERGENCY AMBULANCE SERVICE,EKM SOUTH	AMBULANCE STATION
A3	AMBULANCE SERVICE ON TIME,EKM SOUTH	AMBULANCE STATION
A4	WE ONE AMBULANCE SERVICE,SHENOYS EKM	AMBULANCE STATION
P1	CENTRAL POLICE STATION,HIGHCOURT EKM	POLICE STATION
P2	COMMISSIONER OF POLICE KOCHI CITY , MARINE DRIVE	POLICE STATION
P3	ERANAKULAM TOWN NORTH POLICE STATION,CHITTOOR-	POLICE STATION
P4	ERANAKULAM TOWN SOUTH POLICE STATION,THEVARA	POLICE STATION

Table 4.2.1.a Nodes, Location and Location Type Table

STARTING POINT	ENDING POINT	DISTANCE (KM)
C	H1	0.55
C	H2	1.7
C	H3	3
C	H4	1.3
C	H5	1.3
C	F1	3
C	F2	0.9
C	F3	3.7
C	A1	3
C	A2	2.7
C	A3	3
C	P1	1.8
C	P2	1
A1	H1	2.6
A2	H1	3.1
A3	H1	1.892
A4	H1	1
A1	C	2.2
A2	C	2.9
A3	C	1.6
A4	C	0.8
P1	C	1.8
P2	C	1
P3	C	2
P4	C	3.4
F1	C	3.1
F2	C	0.9
F3	C	2.9

Table 4.2.1.b Distance Table

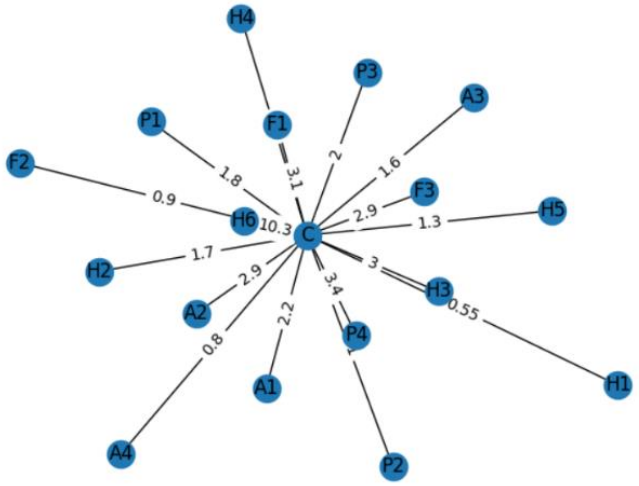


Figure 4.2.1.a: illustration of nodes and edges

Code Implementation

```
import sys
import heapq

def dijkstra(graph, start):
    distances = {node: sys.maxsize for node in graph}
    distances[start] = 0
    queue = [(0, start)]

    while queue:
        current_distance, current_node = heapq.heappop(queue)

        if current_distance > distances[current_node]:
            continue

        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(queue, (distance, neighbor))

    return distances

# Define the graph
graph = {
    'A1': {'C': 2.2},
    'A2': {'C': 2.9},
    'A3': {'C': 1.6},
    'A4': {'C': 0.8},
    'C': {'H1': 0.55, 'H2': 1.7, 'H3': 3, 'H4': 1.3, 'H5': 1.3},
    'H1': {},
    'H2': {},
    'H3': {},
    'H4': {},
    'H5': {}
}

ambulances = ['A1', 'A2', 'A3', 'A4']
```

```
hospitals = ['H1', 'H2', 'H3', 'H4', 'H5']

shortest_distance = float('inf')
best_ambulance = None
best_hospital = None

for ambulance in ambulances:
    distances = dijkstra(graph, ambulance)

    for hospital in hospitals:
        if distances[hospital] < shortest_distance:
            shortest_distance = distances[hospital]
            best_ambulance = ambulance
            best_hospital = hospital

print(f"The best ambulance to take is {best_ambulance} to hospital
{best_hospital} with a total distance of {shortest_distance} km.")
```

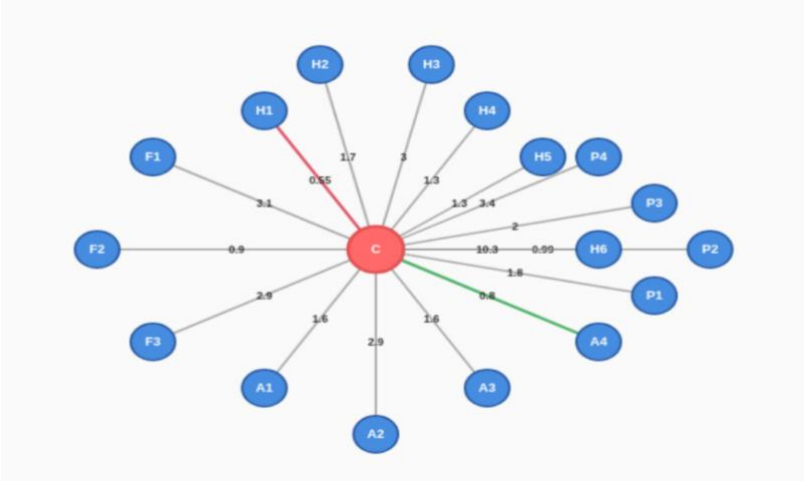


Fig4.2.1.b: Representation of Shortest Path

Best ambulance to take is A4 to hospital H1 with a total distance of 1.35 km.

FIRE STATION TO COLLEGE

To find the shortest route from Fire Force Kerala, Marine Drive (F1) to St. Teresa's College Science Block (C), we'll consider possible route nodes and apply Dijkstra's algorithm.

- A: Fire Force Kerala, Marine Drive
- B: Kadavanthara Junction
- C: Palarivattom Junction
- D: MG Road
- E: Park Avenue Road (St. Teresa's College)
- F: St. Teresa's College Science Block

Edges with weights (distances in km):

- A -> B: 4.5
- B -> C: 2.5
- C -> D: 2.5
- D -> E: 1.5
- E -> F: 0
- A -> E: 0.47 (direct route)

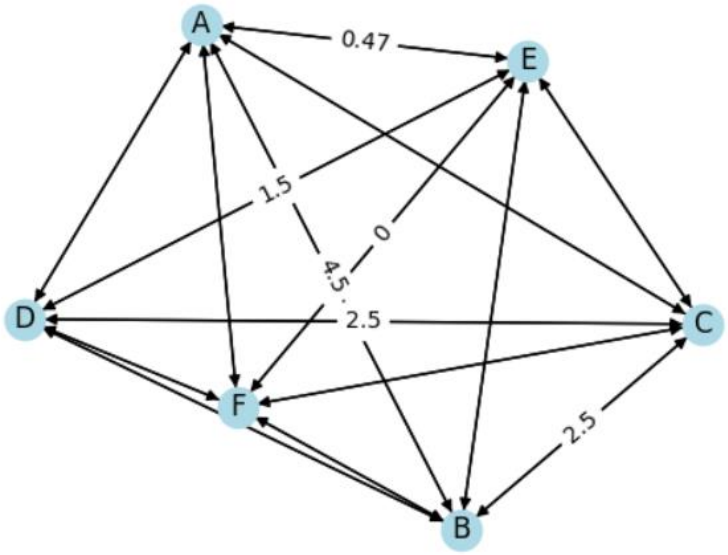


Figure 4.2.1.c: representation of nodes and edges

Code Implementation:

```
import sys
import heapq

graph = {
   'A': {'B': 4.5, 'E': 0.47},
   'B': {'C': 2.5},
   'C': {'D': 2.5},
   'D': {'E': 1.5},
   'E': {'F': 0},
   'F': {}
}

def dijkstra(graph, start_node):
   distances = {node: sys.maxsize for node in graph}
   distances[start_node] = 0
   priority_queue = [(0, start_node)]

   while priority_queue:
      current_distance, current_node = heapq.heappop(priority_queue)
```

```
        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances

start_node = 'A'
distances = dijkstra(graph, start_node)
print(f'Shortest distance from {start_node} to F: {distances["F"]} km')

path = ['F']
current_node = 'F'
while current_node != start_node:
    for node, neighbors in graph.items():
        if current_node in neighbors:
            if distances[current_node] - neighbors[current_node] == distances[node]:
                path.append(node)
                current_node = node
                break
path.reverse()
print(f'Shortest path: {" -> ".join(path)}')
```

RESULT:

Shortest distance from A to F: 0.47 km

Shortest path: A -> E -> F

4.2.2: FINDING SHORTEST TIME PATH USING DIJKSTRA'S ALGORITHM

To find the shortest route that takes less time from A1 to C to H1:

The following data consists of the road network in Ernakulam and Kochi. The nodes represent intersections or landmarks, and the edges represent roads with their corresponding travel times.

Node	Description
A	We One Ambulance Service, Shenoys, Ernakulam

B	St. Teresa's College Science Block, Ernakulam
C	General Hospital, Ernakulam
D	Ernakulam Junction
E	Pallimukku Junction

Table 4.2.2a Road Network Data

Edges with weights (travel times in minutes):

- A -> D: 5 minutes
- A -> B: 8 minutes
- D -> B: 3 minutes
- B -> C: 12 minutes
- D -> C: 10 minutes
- B -> E: 5 minutes
- E -> C: 8 minutes

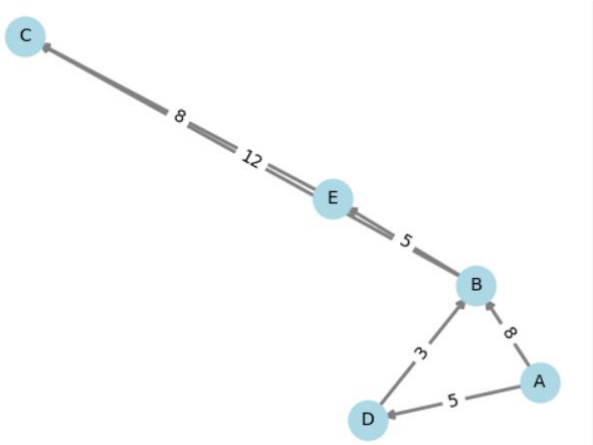


Figure 4.2.2.a: illustration of nodes and edges

To find the shortest path from A to B to C, we'll apply Dijkstra's algorithm in two stages:

1. Find the shortest path from A to B.

2. Find the shortest path from B to C. to start, Initialize the distance to node A as *0* and all other nodes as ∞.

Code Implementation:

```
import heapq

graph = {
    'A': {'D': 5, 'B': 8},
    'D': {'B': 3},
    'B': {'C': 12, 'E': 5},
```

```
    'E': {'C': 8},
    'C': {},  # Add node 'C' to the graph
}

def dijkstra(graph, start_node, end_node):
    distances = {node: float('inf') for node in graph}
    distances[start_node] = 0
    queue = [(0, start_node)]

    while queue:
        current_distance, current_node = heapq.heappop(queue)

        if current_distance > distances[current_node]:
            continue

        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(queue, (distance, neighbor))

    return distances[end_node]

# Find shortest path from A to B
time_ab = dijkstra(graph, 'A', 'B')
print(f"The shortest time from A to B is {time_ab} minutes.")

# Find shortest path from B to C
time_bc = dijkstra(graph, 'B', 'C')
print(f"The shortest time from B to C is {time_bc} minutes.")

# Calculate total time
total_time = time_ab + time_bc
print(f"The total time from A to B to C is {total_time} minutes.")
```

```
The shortest time from A to B is 8 minutes.
The shortest time from B to C is 12 minutes.
The total time from A to B to C is 20 minutes.

Process finished with exit code 0
```

The shortest path would be A -> D -> B -> C with a total travel time of 20 minutes [walking]. In terms of ambulance, total travel time = 5 minute.

CONCLUSION for chapter 4: case study on resource and route optimization

This chapter presented a series of case studies demonstrating the application of Linear Programming and Graph Theory to solve real-world problems in emergency service logistics and resource allocation within the Kochi region.

Key Findings from Location Problems

The first section focused on optimizing the placement of facilities using two Linear Programming models:

Set Covering Location Problem (SCLP): The objective was to select the minimum number of facilities required to cover all demand points within a specific response time limit. Using a maximum coverage distance of 5.33 km (calculated based on a 40 km/h speed and an 8-minute maximum response time), the optimal solution was to select only Medical Trust to achieve 100.0% coverage of the six demand points.

Maximum Covering Location Problem (MCLP): This study addressed a budget constraint, allowing for the establishment of only two ambulance stations from six potential locations. The goal was to maximize the population coverage while ensuring the emergency response time did not exceed 10 minutes. The optimal station locations identified were Welcare hospital Panampilly Nagar and DG medical centre MG Road. This combination achieved 100% coverage, covering a total population of 120,890 within the 10-minute limit.

The chapter also explored a Mixed-Integer Linear Programming (MILP) problem to determine optimal facility utilization and activation.

Out of five potential facilities, the optimal solution activated Medical Trust, Lourdes Hospital, and Aster Medicity. All five demand points were met, with a total cost minimized at INR 15,950.

Key Findings from Route Optimization

The second part of the chapter applied Dijkstra's algorithm from Graph Theory to optimize emergency response routes:

Shortest Distance Path: The objective was to find the shortest route for an ambulance, from its station to a disaster site (St Teresa's College science block [C]) and then to the nearest hospital. The analysis found that the best ambulance is A4 to hospital H1 (General Hospital, Ernakulam) with a total distance of 1.35 km.

Shortest Time Path: This case study calculated the shortest travel time from an ambulance service (Node A: We One Ambulance Service) to the disaster site (Node B: St. Teresa's College) and then to the nearest hospital (Node C: General Hospital, Ernakulam).Applying Dijkstra's algorithm, the shortest time from A to B is 8 minutes and from B to C is 12 minutes, resulting in a total travel time of 20 minutes. However, the case study concludes that for an ambulance, the total travel time is only 5 minutes.

In summary, the case studies demonstrate the power of optimization techniques in making informed, data-driven decisions for critical emergency services, enabling decision-makers to effectively minimize costs, maximize population coverage, and reduce response times.

CONCLUSION

This project successfully demonstrated the power of mathematical optimization techniques in transforming emergency response logistics, achieving the dual objectives of strategic facility placement and dynamic route planning.

SUMMARY OF FINDINGS

- Location Allocation Models (SCLP, MCLP, MILP) provided a definitive, quantitative approach to resource allocation. By applying SCLP, we found the most cost-effective facility count necessary for complete coverage. Conversely, MCLP maximized the covered population under tight budget and resource constraints, revealing the best placement strategy when full coverage is unattainable. Finally, the MILP model showcased the ability to integrate fixed establishment costs, transportation costs, and facility capacity constraints to minimize overall system cost and optimize resource flow.
- Graph Theory and Shortest Path Algorithms delivered the solution for operational efficiency.

  Dijkstra's Algorithm was confirmed as the ideal tool for real-time shortest path calculation in emergency networks, effectively minimizing travel time from a station to an incident (reduced response time). Comparing it with A*, Bellman-Ford, and Floyd-Warshall established the appropriate context for each algorithm—demonstrating that Dijkstra's and A* are best suited for non-negative weight, single-source problems typical of emergency routing, while Bellman-Ford or Floyd-Warshall would be necessary for more complex scenarios involving negative edge weights or all-pairs shortest paths.

REFERENCES

[1] Constantine Toregas, Ralph Swain, Charles ReVelle, Lawrence Bergman, (1971) The Location of Emergency Service Facilities. Operations Research 19(6):1363-1373.

[2] A 0-1 Model for fire and emergency service facility location selection: A case study in Nigeria. Arogundade O.T., Akinwale A.T.; Adekoya A.F. and Awe Oludare G. Source: Journal of Theoretical and Applied Information Technology (JATIT)

[3] IEOR 151 – Lecture 15 Set Covering Problem. Source: University of California, Berkeley

[4] Idayani, D. Puspitasari, Y., & Sari, L. D. K. (2021). Maximal covering problem model for determination of fire station location (A case study). Source: Atlantis Press

[5] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

[6] GeeksforGeeks: https://www.geeksforgeeks.org/

[7] JATINDER N. D. GUPTA. A search algorithm for the generalized flowshop scheduling problem. Computers & Operations Research, 2(2), 83-90.

[8] Franz Josef Och, Nicola Ueffing, Hermann Ney. An efficient A* search algorithm for statistical machine translation.

[9] Warshall, S. (1962). A theorem on Boolean matrices. Journal of the ACM, 9(1), 11–12.

[10] Floyd, R. W. (1962). Algorithm 97: Shortest Path. Communications of the ACM, 5(6), 345.

[11] A note on two problems in connexion with graphs by Edsger W. Dijkstra (1959).

