KWAME NKRUMAH UNIVERSITY OF SCIENCE AND TECHNOLOGY, KUMASI

COLLEGE OF SCIENCE

FACULTY OF PHYSICAL AND COMPUTATIONAL SCIENCES

DEPARTMENT OF MATHEMATICS

OPTIMAL ROUTING OF ACCIDENT VICTIMS BASED ON PATIENT PRIORITY AND HOSPITAL RANKS USING ANT COLONY OPTIMIZATION

By

AMA DARKOAH QUASHIE

AND

SYLVESTER ARHIN MENSAH

A PROJECT SUBMITTED TO THE DEPARTMENT OF MATHEMATICS IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE AWARD OF THE DEGREE OF BACHELOR OF SCIENCE IN MATHEMATICS

September 2023

Declaration

We hereby declare that this project is the result of our own and original research and that no part of it has been submitted to any institution or organization for the award of a degree. All inclusions from the work of others have been duly acknowledged.

QUASHIE AMA DARKOAH	..... 	.....
Student (9336519)	Signature	Date
MENSAH ARHIN SYLVESTER	..... 	.....
Student (9334019)	Signature	Date

Certified by:

PROF. PETER AMOAKO-YIRENKYI	.....	.....
Supervisor	Signature	Date

Certified by:

VERY REV. (PROF.) WILLIAM OBENG-DENTEH	.....	.....
Head of Department	Signature	Date

.

ABSTRACT

Accidents, often unexpected, call for a quick emergency response to transport victims to appropriate medical institutions. This research focuses on optimizing emergency medical service (EMS) vehicle routes to transport accident victims to the most suitable medical facilities. Traditional manual routing methods, still widely used in many regions, lack scientific validation and may not produce the best routes. This study aims to develop a more effective EMS route system to transport accident victims to the most suitable medical facilities efficiently. The objectives include developing a routing model to reduce ambulance travel distances, integrating hospital capability data to match trauma assessments, and testing the model's performance. Distances between locations were obtained using Google Roads data, and an Ant Colony algorithm was used to find optimal routes. The study's findings promise to improve patient transportation, ensuring immediate access to treatment for injuries of different severity. By optimizing EMS routes and response times, this research has the potential to save lives.

DEDICATION

We dedicate this study to the Almighty God for His protection and all the good things and to our parents for their advice, encouragement and financial support during our 4-year study.

ACKNOWLEDGEMENT

We gratefully thank the significant contribution of everyone who directly or indirectly influenced the development and writing of this work. We want to acknowledge our supervisor, Professor Peter Amoako-Yirenkyi, for his advice and inspiration and Evander Nana Bosomefi Eghan's frequent comments and suggestions on how to carry out and document this work.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background Of Study

According to Hussein et al. (2021a), accidents are a sudden occurrence that causes property loss, personal injury, or both. There are various classes of accidents, such as fatal accidents, serious accidents, minor accidents, property damage accidents, and others.

With these classes of accidents, it is essential to know which healthcare facilities best suit which class of accident concerning emergency response. Ignoring the choice of healthcare facilities will inevitably lead to patient transfer between healthcare facilities, thereby increasing the chances of death of the patients, Zeng et al. (2021).

With a time of the essence in emergencies, getting the injured quickly to healthcare facilities can be crucial to their survival. The efficacy of response actions would mean the difference between life and death when a life-threatening incident occurs, Berkoune et al. (2012).

Ambulances and other emergency vehicles must navigate roadways to transport patients to the nearest medical facility. In Sánchez-Mangas et al. (2010), reported that a 10-minute reduction in treatment waiting time reduces the chances of death by one-third.

Choosing the best facility to handle the patient's injuries and the closest proximity is crucial in multiple medical facilities to minimise treatment time. The focus of our study is to determine the best routes for emergency medical vehicles to take from accident scenes to the medical facilities that are best equipped to treat patients with injuries following trauma assessments.

1.2 Problem Statement

Transporting accident victims from the incident site to a hospital with adequate resources to handle the victim's injury has been highly recognised as one of the critical factors that ensure the victim's survival or prevent further deterioration of the extent of their injury Ningwa et al. (2020); therefore its relevance cannot be underestimated.

In ambulance routing problems, especially in disaster response scenarios, hospital capability is frequently disregarded. Following triage, in which patients are examined and classified depending on the severity of their injuries, the nearest hospital is often chosen for patient transfer without regard to the hospital's ability to treat the patient's particular medical condition Talarico et al. (2015).

This disregard for hospital capabilities in ambulance routing might result in inefficiencies and poor patient outcomes. For example, a patient with severe injuries may be taken to a nearby hospital that lacks the necessary resources or experience to offer the level of care required. This can cause treatment delays and potentially worsen the patient's condition Aringhieri et al. (2022).

Furthermore, the current Ambulance routing system used in Ghana, especially the Kumasi Ambulance unit, is typically based on the manual decision-making of the driver. Years of navigation of a particular road network gives these drivers knowl-

edge of the various short routes of the road system and how to elude traffic at a specific time of the day to get to their destination. According to Eftekhari et al. (2019), a participant in an interview stated how EMS drivers take shortcuts to get to accident sites based on their expertise on the road networks.

Though this traditional method has been in operation since the Kumasi Ambulance unit's inception, the drivers' various shortest routes are not scientifically proven. They might not be the optimal shortest routes to their various destinations.

We would want to incorporate hospital capabilities into the ambulance routing model so that while making routing decisions, we can consider the suitable hospitals for patient transport and look for optimal routes to transport accident victims to the nearest appropriate hospitals.

1.3 Objectives

General Objective:
Our general objective is to create an optimised emergency medical service (EMS) route system for efficiently transporting accident victims to appropriate medical facilities.

Specific Objectives:

1. To Formulate a routing model to minimise the travel distance of ambulances to transport victims to the nearest appropriate hospital.

2. To Integrate hospital capability data into the model to determine which hospital can handle the trauma assessment.

3. To run simulations with various scenarios to evaluate the performance of the proposed model

1.4 Methodology

Using "QGIS" and "Google Roads", the location and distances between each node were obtained. The vehicle routing problem was used to formulate the model, and the Ant Colony algorithm was adopted to find the optimal routes for this study.

1.5 Justification

This study would help us improve the process of transporting patients from the incident site to the appropriate hospitals that meet the standard of treatment of their injury severity. Using advanced algorithms, this study would optimise the routes taken by ambulances so that patients can receive the required treatment on time.

1.6 Significance

1. It will ensure the efficient transportation of patients from incident sites to the appropriate hospital by minimising the travel distance and reducing the time it takes to get a patient the required treatment.

2. It will ensure that patients are transported to hospitals that meet the standard of treatment for their injury severity to prevent the transfer of patients from one hospital to another.

1.6.1 Organization

This thesis is organised into five chapters:
Chapter 1 comprises the background of the study, problem statement, objectives, methodology, justification, and the structure of this study. Chapter 2 gives a comprehensive survey of various literature related to this study. The mathematical model of the problem with its constraint and assumptions are presented in Chapter 3. Simulations of the algorithm used are shown in Chapter 4. Chapter 5 shows the findings and recommendations of this study.

Chapter 2

Literature Review

2.1 Background

Much research has been conducted on finding the optimal routes in ambulance routing. The study applied methods such as Ant Colony Optimization, Particle Swarm Optimization, Bat Algorithm, Genetic Algorithm, and others to find the optimal routes. This study focused entirely on finding the optimal route from randomly selected incident sites to a list of selected hospitals in Kumasi with their respective ranks based on resource availability at these hospitals and the type of injury severity they can handle using the Ant Colony Optimization Algorithm.

2.1.1 Definition

1. Graph theory: It studies graphs and mathematical structures to model pairwise relations between objects Wikipedia (2023). A graph (V, E) consists of vertices and edges.

2. Vertices (V): These are points linked by two or more lines. Vertices, also called Nodes in this study, represent specific points in the map's road network, such as hospital locations, incident locations, intersections, etc.

3. Edges (E): This graph line links vertices or nodes. It is also referred to as arcs. It denotes the distance between two or more nodes or locations on a map.

4. Path (P): A path is a series of nodes or vertices connected by edges travelled by a vehicle. In graph theory, a simple or basic path is a route with no recurring vertices.

2.2 Vehicle Routing Problem (VRP)

The vehicle routing problem (VRP) is a well-known combinatorial optimisation problem in which optimal vehicle routes are determined to serve a set of clients. The goal is to reduce the total distance travelled or cost while satisfying constraints such as vehicle capacity, time windows, and client needs. Toth and Vigo (2002) presents a comprehensive overview of VRP formulations, solution methods, and applications. They categorise VRP into numerous groups based on various factors such as the number of depots, the number of vehicles, and the presence of time windows. The authors also examine the problem's complexity and present a summary of exact and approximation algorithms created to address different versions of the problem.

2.2.1 Ambulance Routing Problem (ARP)

The ambulance routing problem is a complex problem that includes determining the most efficient routes for ambulances to take to serve a specific group of patients. This problem involves careful evaluation of variables such as traffic conditions, infrastructure damage, patient severity, and multi-stakeholder collaboration. Recent research has looked into various methodologies, such as machine learning-based risk scores Spangler et al. (2019), location allocation-routing models reward memari2020air, and the usage of advanced technology. These strategies seek to minimise response times, optimise ambulance routes, and ensure patients receive timely and adequate medical care.

2.2.2 Patient Prioritization System

Allocating EMS resources in a mass casualty incident (MCI) is a complex task that involves patient prioritising and medical facility selection. The number of

casualties in an MCI usually exceeds the available medical resources, necessitating the optimal utilisation of limited resources to maximise the number of survivors. Research has looked into various elements of resource allocation in MCIs. In MCIs, patient prioritisation is a critical part of resource allocation. The goal is to identify patients who have potentially fatal injuries and distribute resources accordingly. The Simple Triage and Rapid Treatment (START) system prioritises patients based on their physiological status and is the most widely used standard for mass casualty triage Mills et al. (2013).In conclusion, EMS resource allocation in MCIs necessitates the integration of patient prioritising and hospital selection.

2.2.3 Quantum Geographic Information System (QGIS) and Google Roads

QGIS (Quantum GIS) is an open-source geographic information system (GIS) program that allows users to create, modify, view, analyse, and publish geographical data. It is frequently applied in various sectors, including urban planning, environmental science, natural resource management, etc. QGIS has an easy-to-use interface and supports a variety of data types, making it a valuable tool for spatial analysis and mapping.

Google Roads, on the other hand, refers to Google Maps' road network data. It contains data on road segments, intersections, and other key properties that define the road network. Google Roads is an important component of Google Maps used for various applications, including route planning, navigation, and ETA prediction Derrow-Pinion et al. (2021)

.

QGIS does not incorporate Google Roads directly into its software. QGIS, on the other hand, allows users to import and view road network data in their projects, including Google Roads data. Users can access Google Roads data by connecting to web services that provide road network data Derrow-Pinion et al. (2021).

After importing the Google Roads data into QGIS, users may use the mapping and analysis capabilities of the software to build customised maps and do spatial analysis on the road network data.

2.3 Literature Review

In Tikani and Setak (2019), ambulance routing problems in disaster response scenarios where patients are categorised into different groups based on injury severity and different ambulances with other features also incorporated to transport the various patients to the hospital. Using a semi-soft time window, a survival function is used to determine threshold values that reflect the corresponding emergencies in relief operations.

According to Javidaneh et al. (2010), This paper investigates the use of Ant Colony Optimization (ACO) to solve the Ambulance Routing Problem (ARP) in the context of emergency medical services. The ARP involves finding optimal routes for ambulances to reach patients and convey them to hospitals quickly. ACO is a meta-heuristics algorithm inspired by ant foraging behaviour that optimises this challenging task. The study, undertaken by researchers from renowned Iranian academic institutions, used real-world data to provide a unique ARP problem-solving method. While providing a unique perspective, little was highlighted in explaining methods and experimental outcomes. Overall, it advances the investigation of ACO as a viable tool for optimising ambulance routing in emergency medical care.

In Zeng et al. (2021), focused on ambulance routing problems where emergency service providers, traffic operators, and hospitals are involved in trauma assessment and road clearing to provide efficient first aid. They also implemented a mixed integer model for vehicle routing based on a high-spatial-resolution network to reduce travel time, dispatching cost, and late arrival penalty. Patients are grouped based on the severity of the injury and assigned to hospitals with adequate expertise.

In Talarico et al. (2015), investigated ambulance routing problems where patients were classified into two groups, with some patients treated on the incident sites and others sent to the hospital.

According to Boutilier and Chan (2020), this research explored the optimisation of ambulance emergency response in developing nations. The article dived into the unique issues that resource-constrained environments provided and presented optimisation strategies that can be adapted for these situations. The authors also stressed the importance of addressing traffic conditions and inadequate healthcare infrastructure when optimising ambulance routing and response times. They proposed novel strategies that have the potential to significantly improve emergency care delivery in nations that are still developing. This work contributed significant knowledge into addressing crucial healthcare requirements in resource-constrained settings and encouraged more research in this vital study area.

In Silva and Pinto (2010), proposed a methodology for analysing emergency medical systems (EMS) using simulation and optimisation techniques. They developed a discrete-event simulation model that can be used to generate realistic scenarios of EMS operations. They then used optimisation techniques to find the optimal deployment of ambulances and other resources to minimise response times and maximise system efficiency.

According to Bettinelli et al. (2014), they investigated how optimisation methods could be used to increase the efficiency of Emergency Medical Services (EMS). Using simulation and optimisation models, the authors studied the impact of several factors on EMS efficiency, such as ambulance routing and patient prioritising. They discovered that optimising resource allocation can increase response times and save lives. The study emphasised the significance of integrating data and deploying dynamic systems for decision assistance to improve EMS planning.

In Tlili et al. (2017), an exciting approach to the ambulance routing problem (ARP) is described an exciting approach to the ambulance routing problem (ARP) that employs a swarm-based algorithm based on particle swarm optimisation (PSO). The efficiency of this system was extensively examined using real-world ambulance call data from Tunis, Tunisia, thanks to a collaborative effort between institutions in Tunisia and France, reflecting a worldwide perspective on ambulance routing difficulties. Their research included a thorough comparison with many advanced algorithms, proving that their suggested PSO-based technique outperforms them regarding solution quality and computing efficiency.

In Hussein et al. (2021b), This article proposes an innovative Bat Algorithm (BA) for improving ambulance routes in the Ambulance Routing Problem (ARP). BAT iteratively refines solutions based on the echolocation behaviour of bats. This paper compares BA's performance to existing algorithms using real-world ambulance call data from Sfax, Tunisia. This study highlighted an effective strategy with practical dataset usage, giving a feasible ARP solution. However, it primarily focuses on a single city, with the possibility for more comprehensive urban application and modification to multiple kinds of ambulances.

In Spangler et al. (2019), proposed a machine learning-based risk score strategy to improve ambulance routing and triage decisions. They conducted a study to validate machine learning-based risk assessments in pre-hospital circumstances. They discovered that these risk scores consistently outperformed traditional triage algorithms and human prioritisation decisions in predicting hospital outcomes.

According to Derrow-Pinion et al. (2021), QGIS is an open-source GIS for creating, modifying, viewing, analysing, and publishing geographical data. It is used in urban planning, environmental science, natural resource management, etc. Google Roads is a component of Google Maps used for route planning, navigation, and ETA prediction. QGIS does not directly incorporate Google Roads into its software but

allows users to import and view road network data in their projects and do spatial analysis on the road network data.

Chapter 3

Methodology

3.1 Introduction

This chapter explains the Vehicle Routing Problem, the mathematical model, parameters, methods and algorithm used to solve the problem.

3.2 Terms and Concepts

3.2.1 Optimization Problem

An optimisation problem seeks to find the set of inputs to an objective function that results in a maximum or minimum function Sadrehaghighi (2022). Optimisation problems can be divided into two classes. These are the Continuous Optimization Problem and Combinatorial Optimization Problem.

3.2.2 Combinatorial Optimization Problem

A combinatorial optimisation problem involves discrete decision variables and their elements. Mostly, the solution of a combinatorial optimisation problem has a finite number of solutions. They are NP-Hard problems because there is no deterministic method to solve in polynomial time.

3.2.3 Meta-Heuristics

Meta-heuristics are higher-level heuristic algorithms used to find the approximate optimal solution of a complex combinatorial problem that the exact methods cannot solve. The meta-heuristics techniques are instrumental in solving real-life problems with high dimensions and complex solution space. Some examples of these meta-heuristics are simulating annealing, ant colony optimisation, particle swarm optimisation, tabu search and others.

3.3 Parameters

Parameters	Description
P	set of all patients
P_q	set of high-priority patients.
P_m	set of medium-priority patient
P_l	set of low-priority patient
H	set of all hospitals
H_a	set of grade A hospitals
H_b	set of grade B hospitals
K	set of ambulances
w_q	priority score for high-priority patient
w_m	priority score for medium-priority patient
d_ij	travel distance from incident node(i) to hospital node(j)
w	total priority score
i	incident node
j	hospital node

Table 3.1: Parameters and Description

3.3.1 Decision Variable

$$x_{ijkp} = \begin{cases} 1 & \text{if an ambulance (k) transports patient (p) from the incident node(i)} \\ & \text{to hospital node (j)} \\ 0 & \text{Otherwise} \end{cases}$$

3.3.2 Binary Variable

$$y_{ijp} = \begin{cases} 1 & \text{if a patient (p) is assigned to hospital (h) from the incident node(i)} \\ 0 & \text{Otherwise} \end{cases}$$

3.4 Mathematical Model

$$Minimize \sum_{i} \sum_{j} \sum_{k} \sum_{p} d_{ij} \cdot w \cdot x_{ijkp} \tag{3.1}$$

subject to

$$\sum_{p} x_{ijkp} \leq 1 \tag{3.2}$$

$$\sum_{j} y_{ijp} = 1 \tag{3.3}$$

$$w_q \cdot y_{ijp_q} > w_m \cdot y_{ijp_m} \tag{3.4}$$

$$\sum_{p} y_{ij_a p_q} = \sum_{p} \sum_{k} x_{ij_a k p_q} \tag{3.5}$$

$$\sum_{p} y_{ijp_m} = \sum_{p} \sum_{k} x_{ijkp_m} \tag{3.6}$$

$$\sum_{i} \sum_{j} \sum_{k} \sum_{p} x_{ijkp} \leq \text{total ambulances} \tag{3.7}$$

The objective function (3.1) minimises the total distance travelled, prioritising patient transportation. Constraint (3.2) ensures an ambulance carries a maximum of one patient. Constraint (3.3) ensures a patient is assigned to precisely one hospital. Constraint (3.4) provides priority to high-priority patients before medium-priority patients. Constraint(3.5) ensures that the number of assignments of high-priority patients to a grade-A hospital equals the number of transportation of high-priority patients to a grade-A hospital. Constraint (3.6) ensures that the number of assign-

ments of medium-priority patients to all hospitals equals the number of transportation of medium-priority patients to all hospitals. Constraint(3.7) provides that the total number of ambulances available bounds the number of transport of all patients to all hospitals.

3.5 Ant Colony Optimization

The Ant Colony Optimization, or simply ACO, was first proposed by Marco Dorigo in 1991 based on the foraging behaviour of ants. It is a meta-heuristics method to solve complex combinatorial problems.Akhtar (2019). The ACO deploy artificial ants to find optimal paths among a set of path when locating food sources. These ants leave pheromone trails on the path they traverse. The Ant Colony Optimization can be applied to solve routing, scheduling, and other complex combinatorial problems.

3.6 Procedure

3.6.1 ACO algorithm

Algorithm 1 Template of ACO Talbi (2006)

```
Initialize the pheromone trails;
repeat
    for each ant do
        Solution construction using the pheromone trail;
        Update the pheromone trails;
        Evaporation;
        Reinforcement;
        Stopping criteria
        Best solution found or set of solutions
    end for
```

3.6.2 Solution Process

- Initialisation: Create a colony of artificial ants and get the nested node as the start node and the food source node as the end node. Initialise the amount of pheromone on each of the paths to be taken by the ants.

- Ant Movement: Each ant randomly chooses an initial solution. While the stopping criteria are not met. Each ant selects the next solution based on the pheromone levels and heuristics information. These ants construct the solution iteratively until the stopping criteria are met or the solution is complete.

- Pheromone Level Update: After all the solutions are constructed, we determine the quality of the solutions. The pheromone deposited on the path determines the quality of the solutions. Optionally, apply the evaporation of pheromone on the path to prevent stagnation. The formula calculates the evaporation.

$$\tau_{ij}^{k} = (1 - \rho)\tau_{ij} + \sum_{k=1}^{m} \delta\tau_{ij}^{k} \tag{3.8}$$

- Termination Condition: Check if the stopping criteria are met, which is the number of iterations.

- Solution Tracking: As the algorithm keeps running, keep track of the best solution

- Convergence: If the stopping criteria are not met, repeat the process

- Results: Display the solution after the optimisation process.

Chapter 4

Data Collection, Implementation, Results and Discussion

4.1 Introduction

This chapter focuses on determining the optimal route from randomly selected incident sites to a list of designated hospitals. Furthermore, it describes how data for this study was collected and utilised.

4.2 Data Collection

There are three incident sites, eight hospitals, and some road junctions, making 25 nodes on the network graph. The locations of these incident sites, hospitals, and junctions were marked on a Quantum GIS (QGIS); we then imported Google Roads data to help us map the hospitals, incident sites and junctions and determine distances between the nodes. The hospitals' respective ranks' data was obtained from Agencies Ministry Of Health. (2020, March 5). Ministry of Health. QGIS with a Google road-based map was used to create a map representation of the connected nodes. We used Python 3.7 to run the Ant Colony algorithm and matplotlib and networkX tools to generate an estimated road network graph. The following are the names of the incident sites and hospitals and their respective node number on the network graph.

Node Number	Incident Site
4	Martyrs of Uganda
5	Children Park, Amakom
24	Ashanti Regional School Feeding Farm

Table 4.1: Incident sites and their respective node number

Node Number	Hospital	rank
1	SDA Hospital, Kwadaso	Medium
8	Komfo Anokye Teaching Hospital	High
13	Suame Hospital	Medium
14	Tafo Hospital	Medium
21	Kumasi South Hospital	Medium
22	KNUST Hospital	High
25	Manhyia Hospital	Medium

Table 4.2: Hospitals with their respective node number and rank

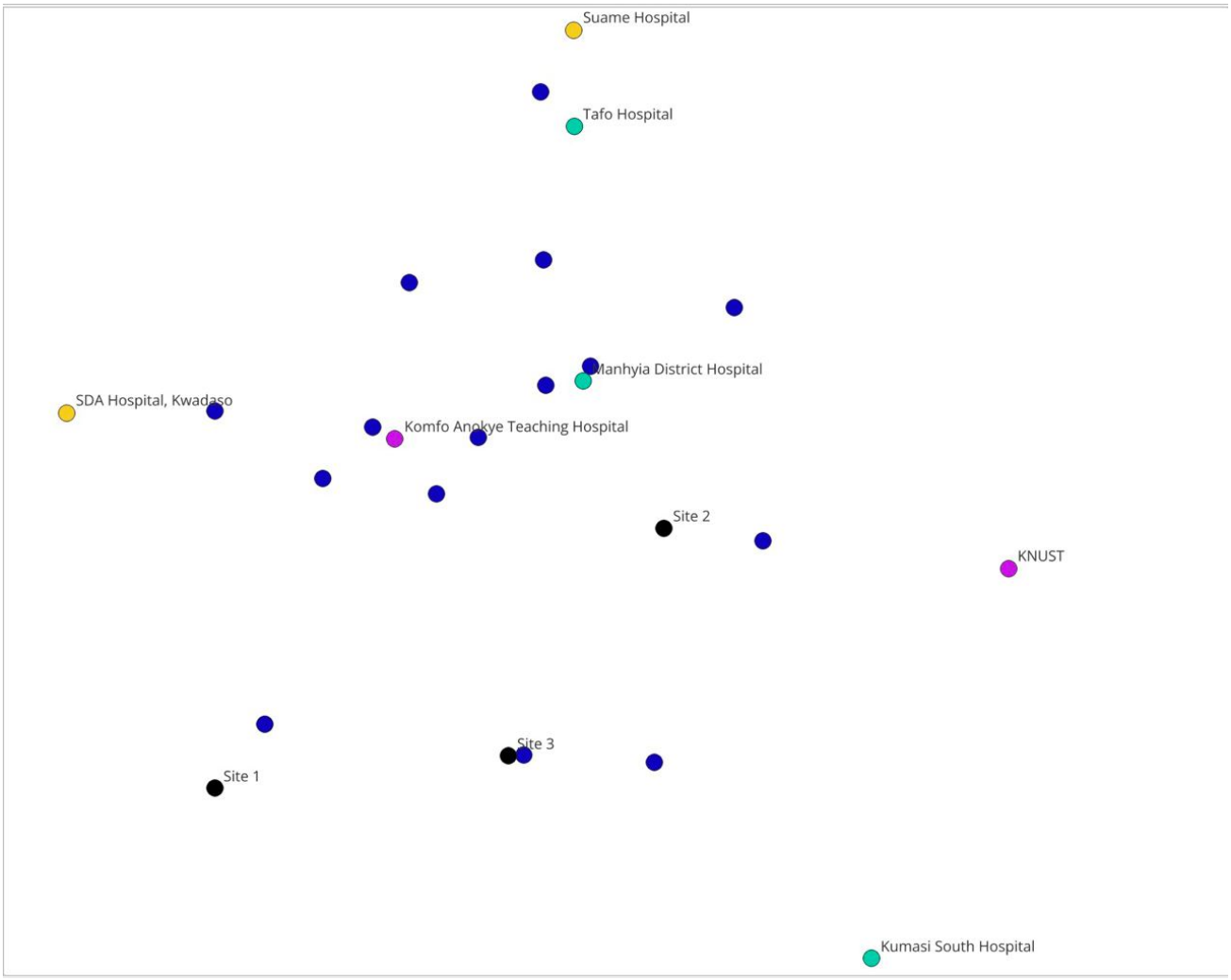

Figure 4.1: This is a representation of Colored Nodes Representing Real-World Locations on a simplified backdrop

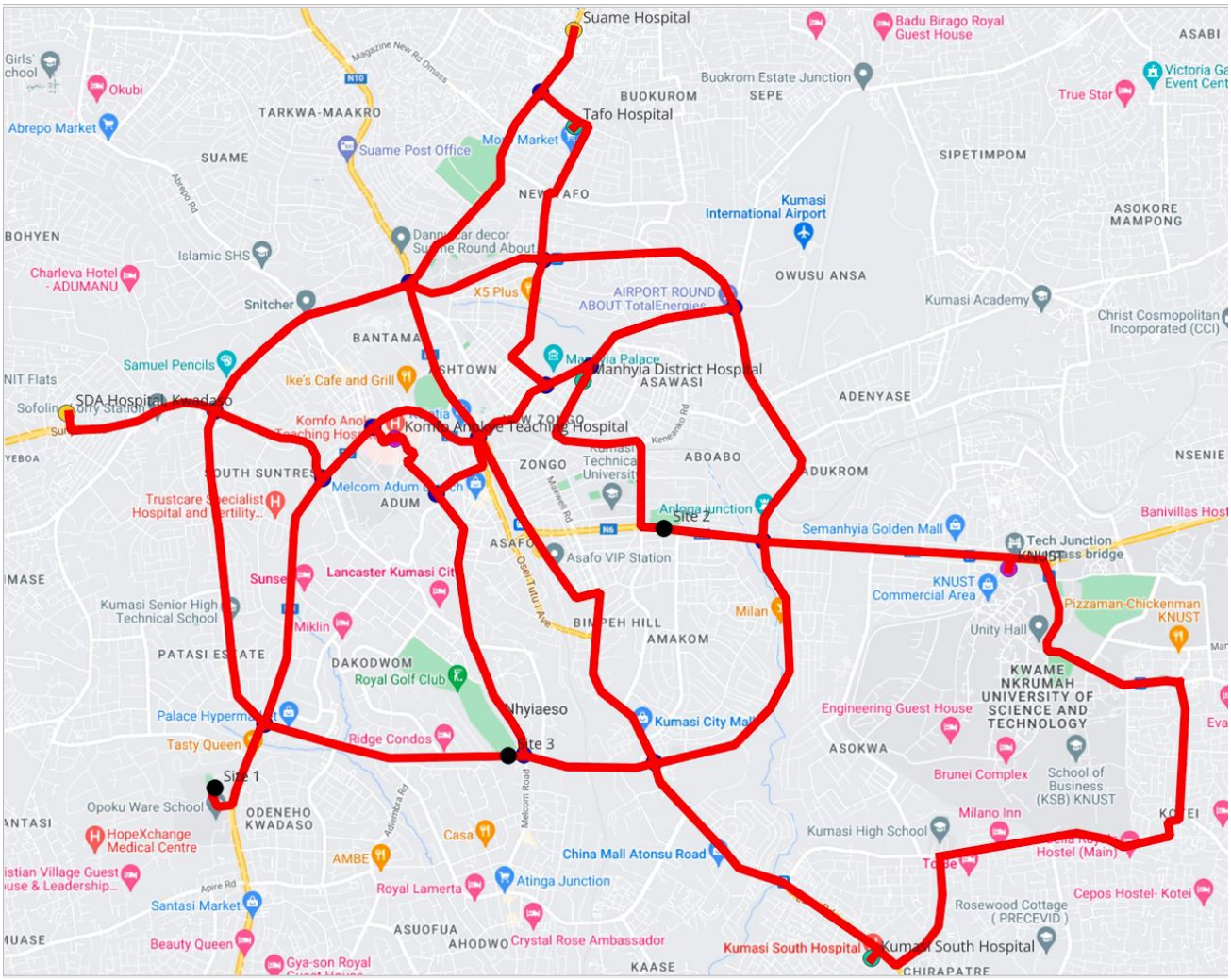

Figure 4.2: This is a map representation of the incident sites and hospitals linked by major roadways on a Google Road-based Map Backdrop

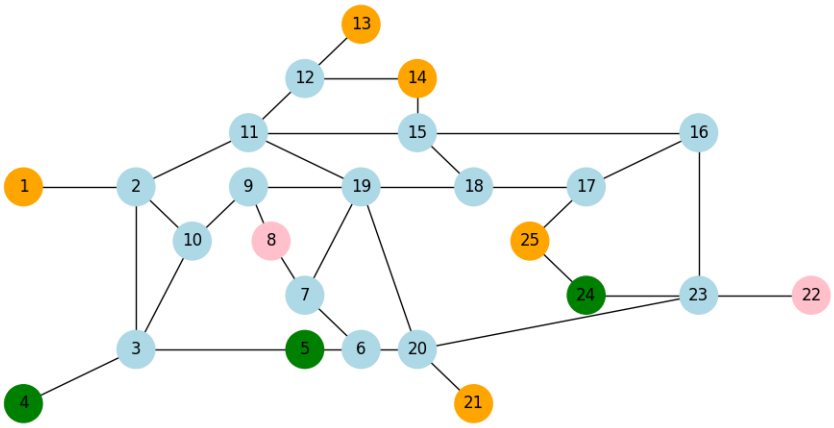

Figure 4.3: This is a sketch network of the Map representation

Figure 4.3 illustrates a directed network, with nodes represented as points and edges shown as connecting lines. This graph has two types of nodes: hospital nodes and incident nodes.

- Hospital nodes (shown as pink and orange points) are meant to be travelled solely in the hospital's direction, representing patients' movement to medical facilities. Moving away from a hospital node, on the other hand, is not allowed.
- Incident nodes (shown as green points) function similarly but in the opposite direction. Only movement away from an incident node is permitted, indicating the dispatch of ambulances away from the incident site.
- All other nodes in the graph (shown as light blue points) are bidirectional, allowing free movement in both directions. This design allows for routing and transit between multiple places to be flexible.

Source	**Destination**	**Distance(m)**
2	1	1655
2	3	3230
2	11	2367
2	10	1368
3	2	3230
3	10	2541
4	3	1213
5	3	2472
5	6	152
6	7	2801
6	20	1319
7	6	2801
7	8	867
7	19	834
9	8	201
9	10	857
9	19	1203
10	2	1368
10	3	2541
10	9	857
11	2	2367
11	12	2368
11	15	1409
11	19	1653
12	13	711
12	14	541

Table 4.3: Weight on the edges of the sketch graph

Source	Destination	Distance(m)
15	11	1409
15	14	1678
15	16	2144
15	18	1487
16	15	2144
16	17	1585
16	23	2557
17	16	1585
17	18	477
17	25	156
18	15	1487
18	17	477
18	19	870
19	20	3977
19	18	870
19	11	1653
19	9	1203
19	7	834
20	21	2945
20	23	3054
20	19	3977
20	6	1319
23	20	3054
23	22	2561
23	16	2557
24	23	984
24	25	2519

Table 4.4: Weight on the edges of the sketch graph

4.3 Results and Discussion from Ant Colony Algorithm

The graphs and tables below represent approximate simulations of optimal routes from incident sites to assigned hospitals and alternate routes from incident sites to assigned hospitals with their respective cost based on the results generated from the Ant Colony algorithm.

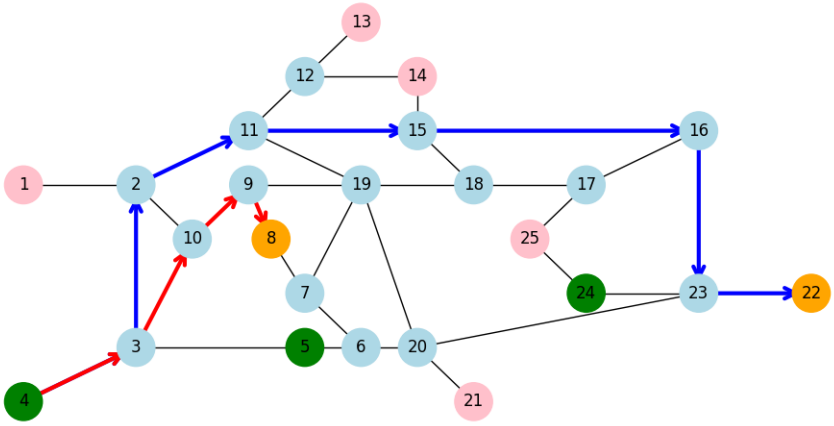

Figure 4.4: Optimal routes for transporting a high-priority patient from the first incident site to available Grade A hospitals

Route	**Cost**
4-3-2-11-15-16-23-22	15481.0
4-3-10-9-8	4812.0

Table 4.5: Routes and cost for Figure 4.4

Figure 4.4 illustrates the optimal route for transporting a high-priority patient from the first incident site (Martyrs of Uganda) to the two Grade A hospitals (Komfo Anokye Teaching Hospital and KNUST Hospital, respectively). However, as shown in table 4.5, the cost of transporting a high-priority patient from Martyrs of Uganda to KNUST Hospital is greater than that of Komfo Anokye Teaching Hospital. Hence, Komfo Anokye Teaching Hospitals is selected as the destination for the patient, and the optimal route for transporting the patient is indicated in red arrows.

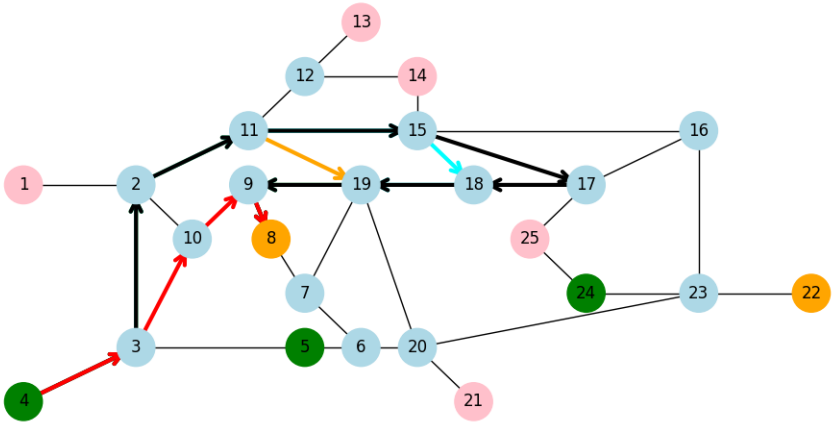

Figure 4.5: Optimal route as well as alternate routes from the first incident site to the nearest Grade A hospital

Alternate Routes	**Cost**
4-3-10-9-8	4812.0
4-3-2-11-19-9-8	9867.0
4-3-2-11-15-18-19-9-8	11980.0
4-3-2-11-15-17-18-19-9-8	14699.0

Table 4.6: Alternate Route and cost for Figure 4.5

Figure 4.5 illustrates other alternate routes to transport a high-priority patient from the first incident site (Martyrs of Uganda) to Komfo Anokye Teaching Hospital. The various alternate routes are represented in other coloured arrows, and the respective costs associated with these alternate routes are shown in table 4.6

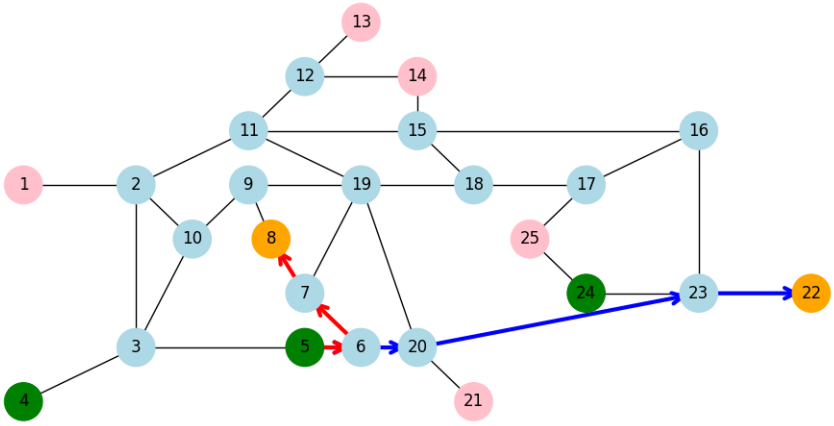

Figure 4.6: Optimal route for transporting a high-priority patient from the second incident site to available Grade A hospitals

Routes	**Cost**
5-6-20-23-22	7086.0
5-6-7-8	3820.0

Table 4.7: Routes and Cost for Figure 4.6

Figure 4.6 illustrates the optimal routes for transporting a high-priority patient from the second incident site (Children Park, Amakom) to the two Grade A hospitals (Komfo Anokye Teaching Hospital and KNUST Hospital, respectively). However, as shown in table 4.7, the cost of transporting a high-priority patient from Children Park, Amakom, to KNUST Hospital is greater than that of Komfo Anokye Teaching Hospital. Hence, Komfo Anokye Teaching Hospital is selected as the destination for the patient, and the optimal route for transporting the patient is indicated in red arrows.

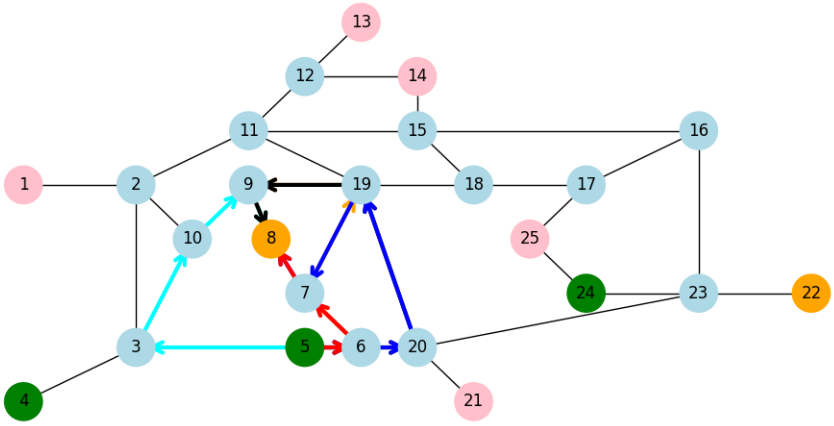

Figure 4.7: Optimal route as well as alternate routes from the second incident site to the nearest Grade A hospital

Alternate Routes	**Cost**
5-6-7-8	3820.0
5-3-10-9-8	6071.0
5-6-7-19-9-8	5191.0
5-6-20-19-9-8	6852.0
5-6-20-19-7-8	7149.0
5-3-11-19-9-8	11126.0

Table 4.8: Alternate Routes and Cost for Figure 4.7

Figure 4.7 illustrates other alternate routes to transport a high-priority patient from the second incident site (Children Park, Amakom) to Komfo Anokye Teaching Hospital. The various alternate routes are represented in other coloured arrows, and the respective costs associated with these alternate routes are shown in table 4.8

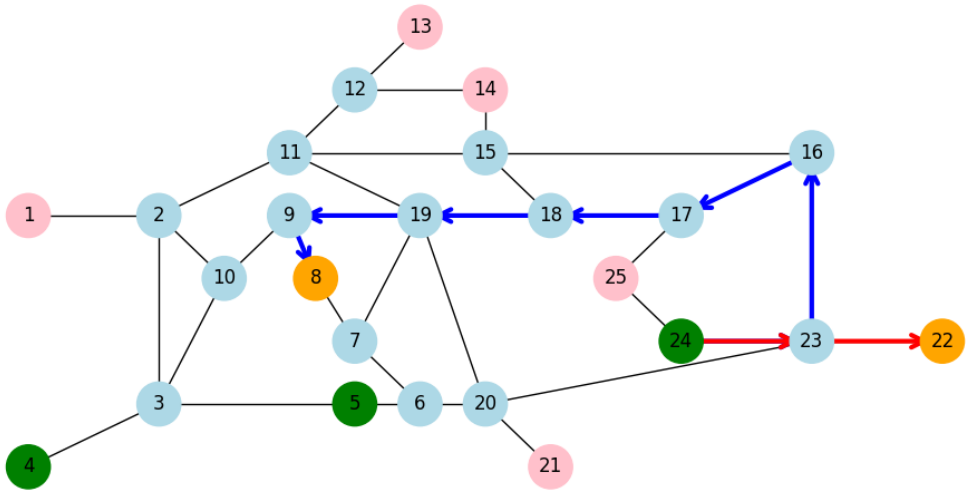

Figure 4.8: Optimal route for transporting a high-priority patient from third incident site to available Grade A hospitals

Routes	**Cost**
24-23-22	3545.0
24-23-16-17-18-19-9-8	7877.0

Table 4.9: Route and Cost for Figure 4.8

Figure 4.8 illustrates the optimal routes for transporting a high-priority patient from the third incident site (Ashanti Regional School Feeding Farm) to the two Grade A hospitals (Komfo Anokye Teaching Hospital and KNUST Hospital, respectively). However, as shown in table 4.9, the cost of transporting a high-priority patient from Ashanti Regional School Feeding Farm to Komfo Anokye Teaching Hospital is greater than that of KNUST Hospital. Hence, KNUST Hospital is selected as the destination for the patient, and the optimal route for transporting the patient is indicated in red arrows. Also, per the graph, there are no alternate routes from Ashanti Regional School Feeding Farm to KNUST Hospital.

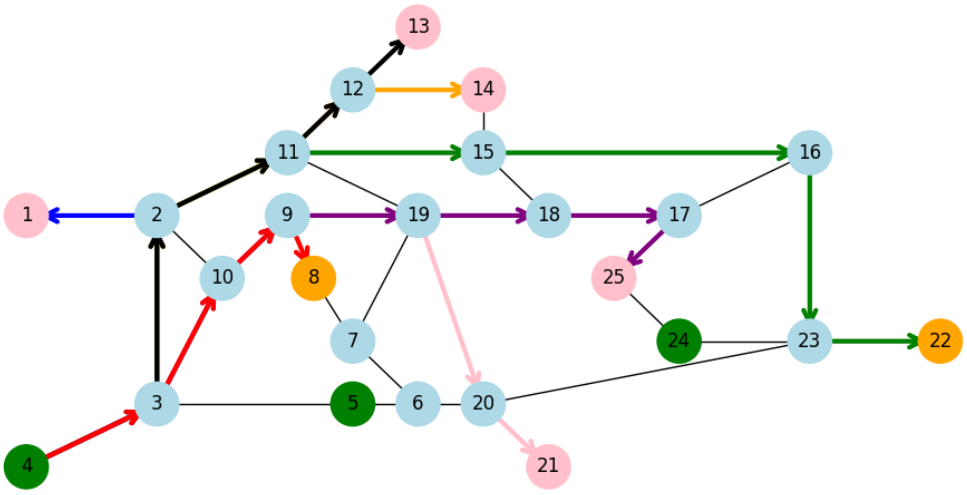

Figure 4.9: Optimal routes for transporting a medium-priority patient from the first incident site to the nearest Hospital(Grade A or Grade B) as well as optimal routes to other hospitals

Route	Cost
4-3-2-11-15-16-23-22	15481.0
4-3-10-9-8	5501.0
4-3-10-9-19-7-6-20-21-21	14402.0
4-3-2-11-12-14	9719.0
4-3-2-11-12-13	9889.0
4-3-10-9-19-18-17-25	7317.0
4-3-2-1	6098.0

Table 4.10: Routes and Cost for Figure 4.9

Figure 4.9 illustrates the optimal routes for transporting a medium-priority patient from the first incident site (Martyrs of Uganda) to the nearest Grade A or Grade B hospital. In this scenario, since any hospital available can serve the medium-priority patient, priority is given to the nearest hospital in terms of proximity to the incident site. The various optimal routes from Martyrs of Uganda to all the hospitals (Grade A or Grade B) are represented in coloured arrows, and the respective costs associated with these optimal routes are shown in table 4.10. From table 4.10 above, the nearest hospital from Martyrs of Uganda, considering the cost, is Komfo Anokye Teaching Hospital. Hence, Komfo Anokye Teaching Hospital is selected as the destination for

the patient, and the optimal route for transporting the patient is indicated in red arrows.

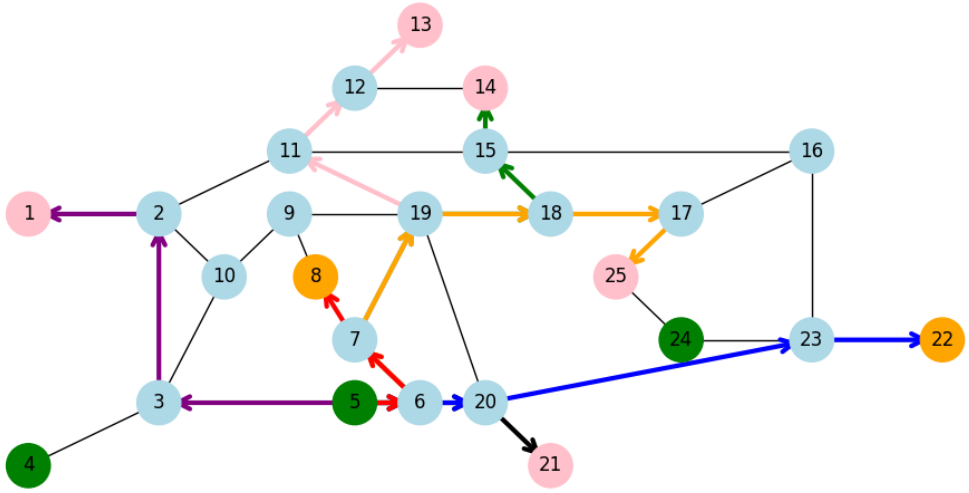

Figure 4.10: Optimal route for transporting a medium-priority patient from second incident site to the nearest Hospital(Grade A or Grade B) as well as optimal routes to other hospitals

Route	**Cost**
5-6-20-23-22	7086.0
5-6-7-8	3820.0
5-6-7-19-18-15-14	7822.0
5-6-7-19-11-12-13	8519.0
5-6-7-19-18-17-25	5290.0
5-3-2-1	7357.0
5-6-20-21	4416.0

Table 4.11: Route and Cost for Figure 4.10

Figure 4.10 illustrates the optimal routes for transporting a medium-priority patient from the second incident site (Children Park, Amakom) to the nearest Grade A or Grade B hospital. In this scenario, since any hospital available can serve the medium-priority patient, priority is given to the nearest hospital in terms of proximity to the incident site. The various optimal routes from Children Park, Amakom, to all

the hospitals (Grade A or Grade B) are represented in coloured arrows, and the respective costs associated with these optimal routes are shown in table 4.11. From table 4.11 above, the nearest hospital from Children Park, Amakom, considering the cost, is Komfo Anokye Teaching Hospital. Hence, Komfo Anokye Teaching Hospital is selected as the destination for the patient, and the optimal route for transporting the patient is indicated in red arrows.

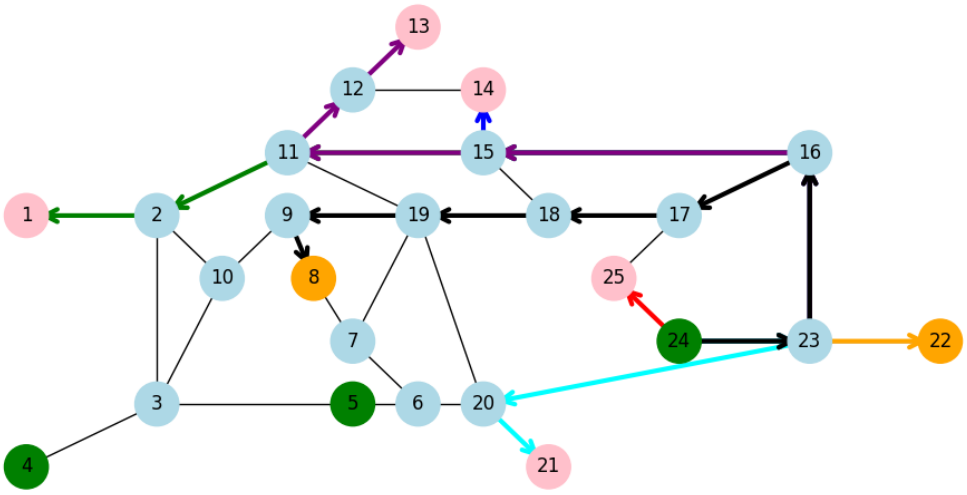

Figure 4.11: Optimal route for transporting a medium-priority patient from third incident site to the nearest Hospital(Grade A or Grade B) as well as optimal routes to other hospitals

Route	Cost
24-23-22	3545.0
24-23-16-17-18-19-9-8	7877.0
24-23-20-21	6983.0
24-25	2519.0
24-23-16-15-11-12-13	10173.0
24-23-16-15-11-2-1	11116.0
24-23-16-15-14	7363.0

Table 4.12: Route and Cost for Figure 4.11

Figure 4.11 illustrates the optimal routes for transporting a medium-priority patient from the third incident site (Ashanti Regional School Feeding Farm) to the nearest Grade A or Grade B hospital. In this scenario, since any hospital available can serve the medium-priority patient, priority is given to the nearest hospital in terms of proximity to the incident site.

The various optimal routes from Ashanti Regional School Feeding Farm to all the hospitals (Grade A or Grade B) are represented in coloured arrows, and the respective costs associated with these optimal routes are shown in table 4.12. From table 4.12 above, the nearest hospital from Ashanti Regional School Feeding Farm,

considering the cost, is Manhyia Hospital. Hence, Manhyia Hospital is selected as the destination for the patient, and the optimal route for transporting the patient is indicated in red arrows.

Chapter 5

Conclusion and Recommendations

5.1 Conclusion

In conclusion, a mathematical model was formulated for the vehicle routing problem, incorporating the hospital capability data. Hospitals were classified into ranks based on their resources and capability to handle an injury severity. Optimal routes were found from the various incident sites to the hospital that meet the standard of treatment. Multiple scenarios were simulated to test the performance of the model.

5.2 Recommendation

Our road network was offline; we suggest future studies examine a more dynamic road network system that adjusts real-time traffic data to discover the best ambulance routes. We also urge that future studies consider additional constraints, such as hospital capacity constraints.

Bibliography

Akhtar, A. (2019). Evolution of ant colony optimization algorithm – a brief literature review.

Aringhieri, R., Bigharaz, S., Duma, D., and Guastalla, A. (2022). Fairness in ambulance routing for post disaster management. *Central European journal of operations research*, pages 1–23.

Berkoune, D., Renaud, J., Rekik, M., and Ruiz, A. (2012). Transportation in disaster response operations. *Socio-Economic Planning Sciences*, 46(1):23–32. Special Issue: Disaster Planning and Logistics: Part 1.

Bettinelli, A., Cordone, R., Ficarelli, F., Righini, G., et al. (2014). Simulation and optimization models for emergency medical systems planning. *Journal of Emergency Management*, 12(4):287–302.

Boutilier, J. J. and Chan, T. C. (2020). Ambulance emergency response optimization in developing countries. *Operations Research*, 68(5):1315–1334.

Derrow-Pinion, A., She, J., Wong, D., Lange, O., Hester, T., Perez, L., Nunkesser, M., Lee, S., Guo, X., Wiltshire, B., et al. (2021). Eta prediction with graph neural networks in google maps. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, pages 3767–3776.

Eftekhari, A., DehghaniTafti, A., Nasiriani, K., Hajimaghsoudi, M., Fallahzadeh, H., and Khorasani-Zavareh, D. (2019). Management of preventable deaths due to road traffic injuries in prehospital phase; a qualitative study. *Archives of academic emergency medicine*, 7(1).

Hussein, T. D. H., Frikha, M., Ahmed, S., and Rahebi, J. (2021a). Ambulance vehicle routing using bat algorithm. In *2021 International Conference on Electrical, Communication, and Computer Engineering (ICECCE)*, pages 1–5.

Hussein, T. D. H., Frikha, M., Ahmed, S., and Rahebi, J. (2021b). Ambulance vehicle routing using bat algorithm. In *2021 International Conference on Electrical, Communication, and Computer Engineering (ICECCE)*, pages 1–5. IEEE.

Javidaneh, A., Ataee, M., and Alesheikh, A. A. (2010). Ambulance routing with ant colony optimization. In *Geospatial Information Systems Congress*, volume 6.

Mills, A. F., Argon, N. T., and Ziya, S. (2013). Resource-based patient prioritization in mass-casualty incidents. *Manufacturing & Service Operations Management*, 15(3):361–377.

Ningwa, A., Muni, K., Oporia, F., Kalanzi, J., Zziwa, E. B., Biribawa, C., and Kobusingye, O. (2020). The state of emergency medical services and acute health facility care in uganda: Findings from a national cross-sectional survey. *BMC Health Services Research*, 20(1):1–10.

Sadrehaghighi, I. (2022). Optimization problem.

Silva, P. M. S. and Pinto, L. R. (2010). Emergency medical systems analysis by simulation and optimization. In *Proceedings of the 2010 winter simulation conference*, pages 2422–2432. IEEE.

Spangler, D., Hermansson, T., Smekal, D., and Blomberg, H. (2019). A validation of machine learning-based risk scores in the prehospital setting. *PLoS One*, 14(12):e0226518.

Sánchez-Mangas, R., García-Ferrrer, A., de Juan, A., and Arroyo, A. M. (2010). The probability of death in road traffic accidents. how important is a quick medical response? *Accident Analysis & Prevention*, 42(4):1048–1056.

Talarico, L., Meisel, F., and Sörensen, K. (2015). Ambulance routing for disaster response with patient groups. *Computers & operations research*, 56:120–133.

Talbi, E.-G. (2006). *Parallel combinatorial optimization*. John Wiley & Sons.

Tikani, H. and Setak, M. (2019). Ambulance routing in disaster response scenario considering different types of ambulances and semi-soft time windows. *Journal of Industrial and Systems Engineering*, 12(1):95–128.

Tlili, T., Harzi, M., and Krichen, S. (2017). Swarm-based approach for solving the ambulance routing problem. *Procedia Computer Science*, 112:350–357.

Toth, P. and Vigo, D. (2002). An overview of vehicle routing problems. *The vehicle routing problem*, pages 1–26.

Wikipedia (2023). Graph theory — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Graph_theory&oldid=1171835383. [Online; accessed 26-August-2023].

Zeng, Z., Yi, W., Wang, S., and Qu, X. (2021). Emergency vehicle routing in urban road networks with multistakeholder cooperation. *Journal of Transportation Engineering, Part A: Systems*, 147(10):04021064.

5.3 Appendix

```python
import networkx as nx

import matplotlib.pyplot as plt

from dataclasses import dataclass, field
import random


@dataclass
class Edge:
    """An edge of the graph.

    Args:
        travel_time (float): The time it takes to travel the edge.
         A high value indicates more traffic.
        pheromones: (float): The amount of pheromones deposited by the ants on the edge.
         Defaults to 1.0.
    """

    travel_time: float
    pheromones: float = 1.0
```

```
@dataclass
class Node:
    """A node in the graph.

    Args:
        id (str): The unique ID of the node.
        edges (dict[str, Edge]): Stores all the outgoing edges from this node.
    """

    id: str
    edges: dict[str, Edge] = field(default_factory=dict)

    def add_edge(self, destination: str, travel_time: float):
        self.edges[destination] = Edge(travel_time)


@dataclass
class Graph:
    """A Directed Graph made up of Nodes and Edges.

    Args:
        graph (dict[str, Node]): The actual graph.
        evaporation_rate (float): The evaporation rate of the pheromones.
        Defaults to 0.1
    """

    graph: dict[str, Node] = field(default_factory=dict)
    evaporation_rate: float = 0.5

    def node_exists(self, id: str) -> bool:
        """Checks if the node exists in the graph.

        Args:
            id (str): The ID of the node.

        Returns:
            bool: True if the node exists in the graph, else False.
        """
        return id in self.graph

    def edge_exists(self, source: str, destination: str) -> bool:
        """Checks if the edge exists in the graph.

        Args:
            source (str): The ID of the source node.
            destination (str): The ID of the destination node.

        Returns:
            bool: True if the edge exists in the graph, else False.
        """
        if not self.node_exists(source) or not self.node_exists(destination):
            return False
        return destination in self.graph[source].edges.keys()

    def add_node(self, id: str) -> None:
        """Add a node to the graph.

        Args:
            id (str): The ID of the node to be added to the graph.
        """
        self.graph[id] = Node(id)

    def add_edge(self, source: str, destination: str, travel_time: float) -> None:
```

```
        """Add an edge connecting 2 nodes in the graph.

        Args:
            source (str): The source node of the edge.
            destination (str): The destination node of the edge.
            travel_time (float): The amount of time it takes to travel
            that edge.
        """
        if not self.node_exists(source):
            self.add_node(source)
        if not self.node_exists(destination):
            self.add_node(destination)
        self.graph[source].add_edge(destination, travel_time)

    def get_all_nodes(self) -> list[str]:
        """Returns a list of all the nodes in the graph.

        Returns:
            list[str]: A list of all the nodes in the graph.
        """
        return list(self.graph.keys())

    def get_all_edges(self) -> list[Edge]:
        """Returns all the edges in the graph.

        Returns:
            list[Tuple[str, str, float]]: A list of Tuples ->
            (source, destination, travel_time)
        """
        edges: list[Edge] = []
        for source in self.graph:
            for destination in self.graph[source].edges:
                edges.append(self.get_edge(source, destination))
        return edges

    def get_edge(self, source: str, destination: str) -> Edge:
        """Returns the Edge if it exists in the graph.

        Args:
            source (str): The ID of the source node.
            destination (str): The ID of the destination node.

        Returns:
            Edge: The Edge object.
        """
        return self.graph[source].edges[destination]

    def get_node_edges(self, id: str) -> dict[str, Edge]:
        """Returns all the edges of a node.

        Args:
            id (str): The ID of the node.

        Returns:
            dict[str, Edge]: The outgoing edges of the node.
        """
        return self.graph[id].edges

    def get_node(self, id: str) -> Node | None:
        """Returns the Node if it exists in the graph.

        Args:
            id (str): The ID of the Node.
```

```
        Returns:
            Optional[Node]: The Node if it exists otherwise returns None.
        """
        if self.node_exists(id):
            return self.graph[id]
        return None

    def get_neighbors(self, id: str) -> list[str]:
        """Returns all the neighbors of a Node in the graph.

        Args:
            id (str): The ID of the Node.

        Returns:
            list[str]: A list of all the neighbors of that Node.
        """
        if not self.node_exists(id):
            return []
        return [neighbor for neighbor in self.graph[id].edges]

    def get_travel_times(self, id: str) -> list[float]:
        """Returns a list of travel times of all the edges of the Node.

        Args:
            id (str): The ID of the Node.

        Returns:
            list[float]: A list of travel times of all the edges of the Node.
        """
        if not self.node_exists(id):
            return []
        travel_times = []
        for _, edge in self.graph[id].edges.items():
            travel_times.append(edge.travel_time)
        return travel_times

    def get_edge_travel_time(self, source: str, destination: str) -> float:
        """Returns the travel time of the specified edge if it exists.

        Args:
            source (str): The source node of the edge.
            destination (str): The destination of the edge.

        Returns:
            float: The travel time of that edge.
        """
        if (
            not self.node_exists(source)
            or not self.node_exists(destination)
            or destination not in self.graph[source].edges
        ):
            return float("inf")
        return self.graph[source].edges[destination].travel_time

    def compute_path_travel_time(self, path: list[str]) -> float:
        """Computes the cost of a path (a list of node IDs).

        Args:
            path (list[str]): The ID of the nodes in the path.

        Returns:
            float: The total travel time of the specified path.
```

```
        """
        cost = 0.0
        for i in range(len(path) - 1):
            if self.edge_exists(path[i], path[i + 1]):
                cost += self.get_edge_travel_time(path[i], path[i + 1])
            else:
                return float("inf")
        return cost

    def evaporate(self) -> None:
        """Evaporates the pheromone values of all the edges given the
        evaporation parameter (rho)."""
        for node_id, node in self.graph.items():
            for neighbor, edge in self.graph[node_id].edges.items():
                edge.pheromones = (1 - self.evaporation_rate) * edge.pheromones

    def deposit_pheromones_on_edge(
        self, source: str, destination: str, new_pheromones: float
    ) -> None:
        """Updates the pheromones on an edge in the graph.

        Args:
            source (str): The source node of the edge.
            destination (str): The destination node of the edge.
            new_pheromones (float): The amount of pheromones to be added to
             the existing value on the edge.

        """
        self.graph[source].edges[destination].pheromones += new_pheromones

    def deposit_pheromones_along_path(self, path: list[str]) -> None:
        """Updates the pheromones along all the edges in the path.

        Args:
            path (list[str]): The path followed by the ant.
        """
        path_cost = self.compute_path_travel_time(path)
        for i in range(len(path) - 1):
            self.deposit_pheromones_on_edge(path[i], path[i + 1], 1 / path_cost)

    def normalize_graph_for_dijkstra(self) -> dict[str, dict[str, float]]:
        """Normalizes the graph for the Dijkstra's algorithm implementation.

        Returns:
            dict[str, dict[str, float]]: A simple, dictionary-structured graph
             with only the travel times of the edges.
        """
        dijkstra_graph: dict[str, dict[str, float]] = {}
        for node in self.get_all_nodes():
            dijkstra_graph[node] = {}
            for edge in self.graph[node].edges:
                dijkstra_graph[node][edge] = self.graph[node].edges[edge].travel_time
        return dijkstra_graph

    def update_edge_travel_time(self, edge: Edge, new_travel_time: float) -> None:
        """Updates the travel time of an edge in the graph.

        Args:
            edge (Edge): The Edge object.
            new_travel_time (float): The new travel time.
        """
        if new_travel_time <= 0:
            new_travel_time = 1
```

```
        edge.travel_time = new_travel_time

    def update_edges_travel_time(
        self, max_delta_time: int = 2, update_probability: float = 0.7
    ) -> None:
        """Stochastically updates the travel time of the edges in the graph.

        Args:
            max_delta_time (int, optional): The maximum allowed change in travel
             time of an edge
            (in positive or negative direction). Defaults to 2.
            update_probability (float, optional): The probability that the travel
             time of an edge
            will be updated. Defaults to 0.7.
        """
        for edge in self.get_all_edges():
            if random.random() > update_probability:
                continue
            delta_time = random.choice(
                [i for i in range(-max_delta_time, max_delta_time + 1, 1) if i != 0]
            )
            self.update_edge_travel_time(edge, edge.travel_time + delta_time)

    def __str__(self) -> str:
        """Displays the graph.

        Returns:
            str: The string representation of the graph.
        """
        display = []
        for node_id, node in self.graph.items():
            display.append("----")
            display.append(f"Node {node.id}")
            display.append("")
            display.append("Edges:")
            for edge_id, edge in node.edges.items():
                display.append(
                    f"{node_id} -> {edge_id}, Travel Time: {edge.travel_time},
                    Pheromones: {edge.pheromones}"
                )
        return "\n".join(display)


@dataclass
class Ant:
    """A class for an Ant that traverses the graph.

    Args:
        graph (Graph): The Graph object.
        source (str): The source node of the ant.
        destination (str): The destination node of the ant.
        alpha (float): The amount of importance given to the pheromone
        by the ant. Defaults to 0.9.
        beta (float): The amount of importance given to the travel time
         value by the ant. Defaults to 0.1.
        visited_nodes (Set): A set of nodes that have been visited by the ant.
        path (list[str]): A list of node IDs of the path taken by the ant so far.
        is_fit (bool): A flag which indicates if the ant has reached the
        destination (fit) or not (unfit). Defaults to False.
    """

    graph: Graph
    source: str
```

```
    destination: str
    alpha: float = 0.5
    beta: float = 0.5
    visited_nodes: set = field(default_factory=set)
    path: list[str] = field(default_factory=list)
    is_fit: bool = False

    def __post_init__(self) -> None:
        self.current_node = self.source
        self.path.append(self.source)

    def reached_destination(self) -> bool:
        """Checks if the ant has reached the destination node in the graph.

        Returns:
            bool: True, if the ant has reached the destination.
        """
        return self.current_node == self.destination

    def _get_unvisited_neighbors(
        self, all_neighbors: dict[str, Edge]
    ) -> dict[str, Edge]:
        """Returns a subset of the all the neighbors of the node which
         are unvisited.

        Args:
            all_neighbors (dict[str, Edge]): A set of all neighbors of
            the node.

        Returns:
            dict[str, Edge]: A subset of all the unvisited neighbors.
        """
        unvisited_neighbors = {}
        for neighbor, edge in all_neighbors.items():
            if neighbor in self.visited_nodes:
                continue
            unvisited_neighbors[neighbor] = edge
        return unvisited_neighbors

    @staticmethod
    def _calculate_edges_total(
        unvisited_neighbors: dict[str, Edge], alpha: float, beta: float
    ) -> float:
        """Computes the denominator of the transition probability equation
        for the ant.

        Args:
            unvisited_neighbors (dict[str, Edge]): A set of unvisited neighbors
             of the current node.
            alpha (float): [description]: The alpha value.
            beta (float): [description]: The beta value.

        Returns:
            float: The summation of all the outgoing edges (to unvisited nodes)
            from the current node.
        """
        total = 0.0
        for neighbor, edge in unvisited_neighbors.items():
            total += (edge.pheromones**alpha) * ((1 / edge.travel_time) ** beta)
        return total

    @staticmethod
    def _calculate_edge_probabilites(
```

```
        unvisited_neighbors: dict[str, Edge], alpha: float, beta: float, total: float
    ) -> dict[str, float]:
        """Computes the transition probabilities of all the edges from the current node.

        Args:
            unvisited_neighbors (dict[str, Edge]): A set of unvisited neighbors of
            the current node.
            alpha (float): [description]: The alpha value.
            beta (float): [description]: The beta value.
            total (float): [description]: The summation of all the outgoing edges
             (to unvisited nodes) from the current node.

        Returns:
            dict[str, float]: A dictionary mapping node IDs to their transition
             probabilities.
        """
        probabilities = {}
        for neighbor, edge in unvisited_neighbors.items():
            probabilities[neighbor] = (
                (edge.pheromones**alpha) * ((1 / edge.travel_time) ** beta)
            ) / total
        return probabilities

    @staticmethod
    def _sort_edge_probabilites(probabilities: dict[str, float]):
        """Sorts the probabilities of the edges in descending order.

        Args:
            probabilities (dict[str, float]): A dictionary mapping the node IDs to their
            transition probabilities.

        Returns:
            [type]: A sorted dictionary mapping node IDs to their transition probabilities.
        """
        return {
            k: v for k, v in sorted(probabilities.items(), key=lambda item: -item[1])
        }

    @staticmethod
    def _choose_neighbor_using_roulette_wheel(
        sorted_probabilities: dict[str, float]
    ) -> str:
        """Chooses the next node to be visited using the Roulette Wheel selection technique.

        Args:
            sorted_probabilities (dict[str, Edge]): A sorted dictionary mapping node IDs to
            their transition probabilities.

        Returns:
            str: The ID of the next node to be visited by the ant.
        """
        pick = random.uniform(0, sum(sorted_probabilities.values()))
        current = 0.0
        next_node = ""
        for key, value in sorted_probabilities.items():
            current += value
            if current > pick:
                next_node = key
                break

        return next_node

    def _pick_next_node(
```

```
        self, unvisited_neighbors: dict[str, Edge], alpha: float, beta: float
    ) -> str:
        """Chooses the next node to be visited by the ant using the Roulette Wheel
         selection technique.

        Args:
            unvisited_neighbors (dict[str, Edge]): A set of unvisited neighbors of
            the current node.
            alpha (float): [description]: The alpha value.
            beta (float): [description]: The beta value.

        Returns:
            str: The ID of the next node to be visited by the ant.
        """

        edges_total = self._calculate_edges_total(unvisited_neighbors, alpha, beta)
        probabilities = self._calculate_edge_probabilites(
            unvisited_neighbors, edges_total, alpha, beta
        )
        sorted_probabilities = self._sort_edge_probabilites(probabilities)
        return self._choose_neighbor_using_roulette_wheel(sorted_probabilities)

    def take_step(self) -> None:
        """This method allows the ant to travel to a neighbor of the current node in the graph."""
        # Mark the node as visited.
        self.visited_nodes.add(self.current_node)

        # Find all the neighboring nodes of the current node.
        all_neighbors = self.graph.get_node_edges(self.current_node)

        # Check if the current node has no neighbors (isolated node).
        if len(all_neighbors) == 0:
            return

        # Find unvisited neighbors of the current node.
        unvisited_neighbors = self._get_unvisited_neighbors(all_neighbors)

        # Pick the next node based on the Roulette Wheel selection technique.
        next_node = self._pick_next_node(unvisited_neighbors, self.alpha, self.beta)

        if not next_node:
            return

        self.path.append(next_node)
        self.current_node = next_node


@dataclass
class ACO:
    graph: Graph

    def _forward_ants(self, ants: list[Ant], max_iterations: int) -> None:
        """Deploys forward search ants in the graph.

        Args:
            ants (list[Ant]): A list of Ants.
            max_iterations (int, optional): The maximum number of steps an ant is allowed
             is to take in order to reach the destination.
                If it fails to find a path, it is tagged as unfit. Defaults to 50.
        """
        for idx, ant in enumerate(ants):
            for i in range(max_iterations):
                if ant.reached_destination():
```

```
                ant.is_fit = True
                break
            ant.take_step()

    def _backward_ants(self, ants: list[Ant]) -> None:
        """Sends the ants (which are fit) backwards towards the source while they drop
        pheromones on the path.

        Args:
            ants (list[Ant]): A list of Ants.
        """
        for idx, ant in enumerate(ants):
            if ant.is_fit:
                self.graph.deposit_pheromones_along_path(ant.path)

    def _deploy_search_ants(
        self,
        source: str,
        destination: str,
        num_ants: int,
        random_spawns: bool = False,
        cycles: int = 100,
        max_iterations: int = 50,
    ) -> None:
        """Deploys search ants which traverse the graph to find the shortest path.

        Args:
            source (str): The source node in the graph.
            destination (str): The destination node in the graph.
            num_ants (int): The number of ants to be spawned.
            random_spawns (bool): A flag to determine if the ants should be spawned at
            random nodes or always at the source node.
            cycles (int, optional): The number of cycles of generating and deploying ants
             (forward and backward). Defaults to 100.
            max_iterations (int, optional): The maximum number of steps an ant is allowed
             is to take in order to reach the destination.
                If it fails to find a path, it is tagged as unfit. Defaults to 50.
        """
        for cycle in range(cycles):
            ants: list[Ant] = []
            for _ in range(num_ants):
                spawn_point = (
                    random.choice(self.graph.get_all_nodes())
                    if random_spawns
                    else source
                )
                ants.append(Ant(self.graph, spawn_point, destination))
            self._forward_ants(ants, max_iterations)
            self.graph.evaporate()
            self._backward_ants(ants)

    def _deploy_solution_ant(
        self,
        source: str,
        destination: str,
        max_iterations: int = 1_000,
    ) -> list[str]:
        """Deploys the final ant that greedily w.r.t. the pheromones finds the
        shortest path from the source to the destination.

        Args:
            source (str): The source node in the graph.
            destination (str): The destination node in the graph.
```

```
        Returns:
            list[str]: The shortest path found by the ants (A list of node IDs).
        """
        # Spawn an ant which favors pheromone values over edge costs.

        paths = []

        for _ in range(max_iterations):
            ant = Ant(self.graph, source, destination, alpha=0.999, beta=0.001)
            for __ in range(1_000):
                if not ant.reached_destination():
                    # print(ant.path)
                    ant.take_step()
                else:
                    if ant.path not in paths:
                        paths.append(ant.path)
                        break

        current_minimum = float("inf")
        shortest_path = []
        for path in paths:
            travel_time = self.graph.compute_path_travel_time(path)
            if travel_time < current_minimum:
                current_minimum = travel_time
                shortest_path = path

        return shortest_path

    def find_shortest_path(
        self, source: str, destination: str
    ) -> tuple[list[str], float]:
        """Finds the shortest path from the source to the destination in the
        graph using the traditional Ant Colony Optimization technique.

        Args:
            source (str): The source node in the graph.
            destination (str): The destination node in the graph.

        Returns:
            list[str]: The shortest path found by the ants (A list of node IDs).
            float: The total travel time of the shortest path.
        """
        self._deploy_search_ants(
            source,
            destination,
            num_ants=200,
            random_spawns=False,
            cycles=100,
            max_iterations=50,
        )
        shortest_path = self._deploy_solution_ant(source, destination)
        return shortest_path, self.graph.compute_path_travel_time(shortest_path)


graph = Graph()

graph.add_edge("2", "1", travel_time=1655)
graph.add_edge("2", "3", travel_time=3230)
graph.add_edge("2", "11", travel_time=2367)
graph.add_edge("2", "10", travel_time=1368)
graph.add_edge("3", "2", travel_time=3230)
graph.add_edge("3", "10", travel_time=3230)
```

```
graph.add_edge("4", "3", travel_time=1213)
graph.add_edge("5", "3", travel_time=2472)
graph.add_edge("5", "6", travel_time=152)
graph.add_edge("6", "7", travel_time=2801)
graph.add_edge("6", "20", travel_time=1319)
graph.add_edge("7", "6", travel_time=2801)
graph.add_edge("7", "8", travel_time=867)
graph.add_edge("7", "19", travel_time=834)
graph.add_edge("9", "8", travel_time=201)
graph.add_edge("9", "10", travel_time=857)
graph.add_edge("9", "19", travel_time=1203)
graph.add_edge("10", "2", travel_time=1368)
graph.add_edge("10", "3", travel_time=2541)
graph.add_edge("10", "9", travel_time=857)
graph.add_edge("11", "2", travel_time=2367)
graph.add_edge("11", "12", travel_time=2368)
graph.add_edge("11", "15", travel_time=1409)
graph.add_edge("11", "19", travel_time=1653)
graph.add_edge("12", "13", travel_time=711)
graph.add_edge("12", "14", travel_time=541)
graph.add_edge("15", "11", travel_time=1409)
graph.add_edge("15", "14", travel_time=1678)
graph.add_edge("15", "16", travel_time=2144)
graph.add_edge("15", "18", travel_time=1487)
graph.add_edge("16", "15", travel_time=2144)
graph.add_edge("16", "17", travel_time=1585)
graph.add_edge("16", "23", travel_time=2557)
graph.add_edge("17", "16", travel_time=1585)
graph.add_edge("17", "18", travel_time=477)
graph.add_edge("17", "25", travel_time=156)
graph.add_edge("18", "15", travel_time=1487)
graph.add_edge("18", "17", travel_time=477)
graph.add_edge("18", "19", travel_time=870)
graph.add_edge("19", "20", travel_time=3977)
graph.add_edge("19", "18", travel_time=870)
graph.add_edge("19", "11", travel_time=1653)
graph.add_edge("19", "9", travel_time=1203)
graph.add_edge("19", "7", travel_time=834)
graph.add_edge("20", "21", travel_time=2945)
graph.add_edge("20", "23", travel_time=3054)
graph.add_edge("20", "19", travel_time=3977)
graph.add_edge("20", "6", travel_time=1319)
graph.add_edge("23", "20", travel_time=3054)
graph.add_edge("23", "22", travel_time=2561)
graph.add_edge("23", "16", travel_time=2557)
graph.add_edge("24", "23", travel_time=984)
graph.add_edge("24", "25", travel_time=2519)

source = "4"
destination = "8"

# print(graph)
aco = ACO(graph)

aco_path1, aco_cost1 = aco.find_shortest_path('5', '14')
aco_path2, aco_cost2 = aco.find_shortest_path('5', '1')
aco_path3, aco_cost3 = aco.find_shortest_path('5', '22')
aco_path4, aco_cost4 = aco.find_shortest_path('5', '13')
aco_path5, aco_cost5 = aco.find_shortest_path('5', '21')
aco_path6, aco_cost6 = aco.find_shortest_path('5', '25')
aco_path7, aco_cost7 = aco.find_shortest_path('5', '8')
```

```
print(f"ACO - path: {aco_path1}, cost: {aco_cost1}")
print(f"ACO - path: {aco_path2}, cost: {aco_cost2}")
print(f"ACO - path: {aco_path3}, cost: {aco_cost3}")
print(f"ACO - path: {aco_path4}, cost: {aco_cost4}")
print(f"ACO - path: {aco_path5}, cost: {aco_cost5}")
print(f"ACO - path: {aco_path6}, cost: {aco_cost6}")
print(f"ACO - path: {aco_path7}, cost: {aco_cost7}")



# Create graph('2', '1')
G = nx.Graph()


# Add Nodes
nodes = [
    '1', '2', '3', '4', '5', '6', '7',
    '8', '9', '10', '11', '12', '13', '14',
    '15', '16', '17', '18', '19', '20', '21',
    '22', '23', '24', '25'
]
G.add_nodes_from(nodes)



node_colors = {}

change_nodes = ['4', '24', '5']
for i in change_nodes:
    node_colors[i] = "green"


change_nodes2 = ['8', '22',]
for i in change_nodes2:
    node_colors[i] = "orange"

change_nodes3 = ['1','14','13','25','21']
for i in change_nodes3:
    node_colors[i] = 'pink'
# Extract and add edges
edges = [
    ('2', '1'), ('2', '3'), ('2', '11'), ('2', '10'), ('3', '2'), ('3', '10'),
     ('4', '3'), ('5', '3'), ('5', '6'), ('6', '7'), ('6', '20'), ('7', '6'),
      ('7', '8'), ('7', '19'), ('9', '8'), ('9', '10'), ('9', '19'), ('10', '2'),
      ('10', '3'), ('10', '9'), ('11', '2'), ('11', '12'), ('11', '15'), ('11', '19'),
      ('12', '13'), ('12', '14'), ('15', '11'), ('15', '14'), ('15', '16'), ('15', '18'),
      ('16', '15'), ('16', '17'), ('16', '23'), ('17', '16'), ('17', '18'), ('17', '25'),
      ('18', '15'), ('18', '17'), ('18', '19'), ('19', '20'), ('19', '18'), ('19', '11'),
      ('19', '9'), ('19', '7'), ('20', '21'), ('20', '23'), ('20', '19'), ('20', '6'),
       ('23', '20'), ('23', '22'), ('23', '16'), ('24', '23'), ('24', '25')
]

G.add_edges_from(edges)


# Set positions
# position = {
#     '1': (0,0), '2': (3,0), '3': (3,3), '4': (6,0), '5': (6,3), '6': (3,-3),
#  '7': (6,-3),
#     '8': (0.5,-0.5), '9': (3.5,-0.5), '10': (3.5,2.5), '11':
#  (6.5, -0.5), '12': (6.5,2.5), '13': (3.5,-3.5), '14': (6.5, -3.5),
#  '15': (-0.5,-0.5),
```

```
#      '16': (1,-1), '17': (4,-1), '18': (4,2), '19': (7,-1), '20': (7,2),
# '21': (4,-4), '22': (7,-4), '23': (0,-1),
#      '24': (1.5, -1.5), '25': (4.5,-1.5), '26': (4.5, 1.5), '27': (7.5,-1.5),
#  '28': (7.5, 1.5), '29': (4.5, -4.5), '30': (7.5,-4.5), 'E2': (0.5,-1.5),
#      '31': (5,1), '32': (8,-2), '33': (8,1), '34': (8,-5), 'E1': (2,-2),
#  'E4': (5,-5), 'E3': (5,-2),
#      'E5': (5.5, 0.5), 'E7': (8.5,0.5), 'E6': (8.5,-2.5), 'E8': (8.5,-5.5),
# 'E9': (9,-5),
# }


position = {
    '13': (3,3),
    '12': (2.5,2),'14': (3.5,2),
    '11': (2,1), '15': (3.5,1),'16': (6,1),
    '1': (0,0),'2':(1,0), '9':(2,0), '19':(3,0), '18':(4,0), '17': (5,0),
    '10': (1.5,-1),'8':(2.2,-1), '25':(4.5,-1),
    '7':(2.5,-2),'24': (5,-2),'23': (6,-2),'22': (7,-2),
    '3': (1,-3),'5': (2.5,-3),'6': (3,-3),'20': (3.5,-3),
    '4':(0,-4), '21':(4,-4)
}

fig, ax = plt.subplots()
# Draw graph outline
nx.draw(G, position, with_labels=True, node_color = [node_colors.get(i, "lightblue")
for i in G.nodes()], node_size=800)

# Highlightes path
color = ['blue','green','orange','purple','cyan','black','red']
num = 0
for shortest_path in [aco_path1,aco_path2,aco_path3,aco_path4,aco_path5,aco_path6,aco_path7]:
    highlighted_edges = [(shortest_path[i], shortest_path[i+1]) for i in range(len(shortest_path)-1)]
    nx.draw_networkx_edges(G, position, edgelist= highlighted_edges, edge_color=color[num],
     width=3, ax = ax, arrows=True, arrowstyle="->", arrowsize=20)
    num += 1


# plt.savefig("./exit_1.png")
plt.show()
```