AN ANT-COLONY APPROACH TO SCHEDULING CHARGING SESSIONS OF ELECTRIC VEHICLE FLEETS WITH HETEROGENEOUS SCHEDULING CONSTRAINTS

by

Derek D. Redmond

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Electrical Engineering

Approved:

Greg Droge, Ph.D. Major Professor	Burak Sarsilmaz, Ph.D. Committee Member
Hongjie Wang, Ph.D. Committee Member	David F. Feldon, Ph.D. Vice Provost of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2025

ABSTRACT

An Ant-Colony Approach to Scheduling Charging Sessions of Electric Vehicle Fleets with Heterogeneous Scheduling Constraints

by

Derek D. Redmond, Master of Science

Utah State University, 2025

Major Professor: Greg Droge, Ph.D.
Department: Electrical and Computer Engineering

Electric vehicles and electric vehicle fleets are gaining widespread usage. A major challenge in managing an electric vehicle fleet is creating a cost-effective strategy for scheduling the electric vehicles in the fleet. This work focuses on creating schedules for heterogeneous vehicles with heterogeneous scheduling constraints that aim to minimize the cost of charging the fleet. The proposed technique is to use Ant Colony Optimization, as metaheuristic approaches are effective in terms of both accuracy and speed. The heterogeneous schedule types are referred to as open (no routing constraints), flexible (routing constraints, but no time constraints), and fixed (both routing and timing constraints). An ant colony algorithm converts open schedules into either flexible or fixed schedules, and another ant colony algorithm creates schedules from flexible and fixed schedules. The route-generating ant colony is effective against a hybrid genetic algorithm, and the schedule-generating ant colony is affective against bin-packing and network-flow approaches.

(153 pages)

PUBLIC ABSTRACT

An Ant-Colony Approach to Scheduling Charging Sessions of Electric Vehicle Fleets with Heterogeneous Scheduling Constraints

Derek D. Redmond

Electric vehicles and electric-vehicle fleets are gaining widespread usage. A major challenge of managing an electric-vehicle fleet is scheduling when the vehicles can charge and when the vehicles can complete their tasks. Finding smart ways to schedule electric-vehicle charging reduces both the cost of charging vehicles and increases the health of the grid and the environment. This best-of-both-worlds outcome is because of the pricing structure that electric utility companies use. When minimizing the cost of charging the vehicles, it is important to consider that vehicle fleets have constraints on when they can charge. Vehicles are usually limited by the tasks they need to perform. Examples of these tasks include buses attending to their stops and packages being delivered to houses. Since the objective is to minimize the cost of charging, and there are constraints on when the vehicles can charge, this problem can be formulated as an optimization problem. Minimizing a cost given constraints creates an optimization problem. An optimization solution approach called Ant Colony Optimization is used firstly to create energy-efficient routes and secondly to schedule the routes in a way that aims to minimize the cost of charging the electric-vehicle fleet. The electric-vehicle fleet in this work has different types of electric vehicles and different schedule constraints based on the tasks the vehicles complete. The ant colony approach makes good solutions in a relatively short period of time.

To the curious, the creative, and the kind

ACKNOWLEDGMENTS

No man is an island,
entire of itself;
every man is a piece of the continent,
a part of the main.
- John Donne, Meditation XVII.

First and foremost, I would like to thank my amazing advisor, Dr. Greg Droge, for always helping and supporting me for these past few years and especially for helping me stretch, learn new things, and come up with new ideas. This work would not have been possible without him. I would also like to thank the rest of my committee, Dr. Burak Sarsilmaz and Dr. Hongjie Wang, for their help and support in making this thesis what it is. I would also like to thank the members of my lab who have served as mentors for my academic and future career as well as provide me with day-to-day support.

I am also grateful for the countless supervisors, mentors, teachers, and fellow students who have inspired me, helped me follow my academic pursuits, and cultivated my love and interest in engineering, mathematics, literature, and science. They have touched my life in ways that I cannot express. Although I am not the best at presenting it, I am deeply grateful for them and the many lessons they have taught me.

And last, but not least, I would like to thank my family and friends. They have been the basis for all my support, learning, and growth. They have been the most influential, the most empowering, and the most loving force in my life. They have been with me through my highs and lows and have shaped me into who I am today.

Derek D. Redmond

CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Some of the biggest challenges of electric vehicle (EV) adaptation stem from the limited range of EVs. This limited range, along with longer fueling times, adds complexity to the management of a vehicle fleet. Longer refueling times directly affects the efficacy of the fleet, especially if the fleet must adhere to a rigid schedule. Furthermore, the power drawn from the grid to quickly charge vehicles with larger battery capacities is significant and can be very costly. Therefore, to enhance the adaptation of electric vehicle fleets, the management of an EV fleet must be more economically viable. The challenge addressed in this work is the scheduling of charging sessions for EV fleets with heterogeneous scheduling constraints, with the objective of minimizing the cost of electricity to manage an EV fleet. In this work, this challenge is called "the scheduling problem."

The scheduling problem in this work considers a fleet of EVs with a mix of fixed-, flexible-, and open-schedule vehicles, similar to that of [2]. *Fixed-schedule vehicles* have specific times for when each vehicle must be at a location. An example of a fixed-schedule vehicle is a bus that has to arrive at specific stops at predefined times. *Flexible-schedule vehicles*, on the other hand, have specific routes that must be done, but no specific times where any vehicle must be at a certain location. Examples of flexible-schedule vehicles include mail delivery and waste collection, which do have routes, but no predefined times that mail must be delivered or that waste must be collected. Lastly, fleets or vehicles without a predefined schedule or route, referred to as *open-schedule vehicles* do not have specific routes and also do not have specific times when they must be at a location. An example of an open-schedule vehicle is a package-delivery van, which has a list of addresses to deliver the packages, but no predetermined route or delivery times.

The goal of the scheduling problem is to generate charging schedules for the heterogeneous fleet that minimize the cost of electricity. The cost of electricity may depend on

multiple factors, including the maximum power drawn from the grid for the month, how much energy was consumed over the month, the time of day the energy was consumed, the time of day the maximum power drawn from the grid for the month occurred, and the time of year the energy was consumed [3]. Thus, it is important to consider power demand, energy consumption, and timing when minimizing the cost of managing an EV fleet. Power demand considerations can be simple, such as ensuring that no two vehicles are charging at the same time. Consumption costs are divided into on-peak and off-peak charging and depend on the time of day the vehicles charge their batteries. The main contributions of this research are to create energy-aware routes that can be used in a route-scheduling paradigm and a charging schedule for an electric vehicle fleet with heterogeneous constraints that minimizes the electricity costs of powering the fleet.

The remainder of this work is as follows: Chapter 2 lays the foundation for understanding the context of this work, Chapter 3 covers the details of the routing problem and solution techniques, Chapter 4 discusses the scheduling problem, and finally Chapter 5 concludes this work.

CHAPTER 2

BACKGROUND

This chapter provides the context and contributions of the research presented in this work. The essential concepts that lay the groundwork for this thesis include routing problems, scheduling problems, and solution techniques, with a focus on Ant Colony Optimization. The contributions include novel ant colony algorithms for solving specific problems that are not in the literature as well as a modified genetic algorithm used for a comparison against one of the ant colony algorithms. The remainder of this chapter contains a literature review and a section on contributions.

2.1 Literature Review

The following literature review helps provide context to the contributions of this work. It details the concepts and evolution of routing problems, the concepts of scheduling problems, solution techniques, a background of ant colony optimization, and a background of genetic algorithms.

2.1.1 Routing Problems

The family of selective routing problems (SRPs) is a subset of combinatorial optimization problems. They are graph-theoretical problems that determine optimal vehicle routes, often represented by a sequence of edges, to visit, or otherwise serve vertices in the problem [4]. Some famous examples of SRPs include the Traveling Salesman Problem (TSP) [5], the Vehicle Routing Problem (VRP) [6], and the Location Routing Problem (LRP) [7].

The VRP is an NP-hard problem introduced by Dantzig and Ramser as the "Truck Dispatching Problem" in 1959 [6]. As time has progressed, the VRP has become a canonical optimization problem because it is applicable to many situations. It also has several variations, which build in complexity and bridge the gap between research and real-world

applications [8]. Variations of and additions to the original VRP include a time window constraint, first formulated by Solomon in [9] and distance constraints for an asymmetric problem [10] by Laporte et al. Another variation of the VRP is the electric vehicle routing problem (EVRP), which includes the range of the vehicle, constrained by its battery, and also has a number of its own subvariations [11] [12].

In the literature, the EVRP usually minimizes distance [12]. However, it is an area of interest to solve the EVRP in order to create *energy-aware* routes that minimize the energy consumed by the fleet. There are relatively few cases that minimize energy, while taking traveled distance, vehicle speed, vehicle load, and road gradients into account, such as [13] and [14]. Almost all works use a constant energy-to-distance unit, where only 17 of the 136 papers surveyed in [12] used nonlinear models. The charging considerations in the EVRP, introduced in [15], include charging on-route such as that of [16].

Electric vehicle routing has also been formulated as a variant of an orienteering problem (OP), but this technique appears to be less popular. The OP is based on the sport of orienteering, where a participant in an orienteering exercise visits unique waypoints and receives a score at each waypoint. The optimization problem is to maximize the participant's total score by visiting waypoints in a given amount of time [17]. A Team Orienteering Problem (TOP) is where members of a team "divide and conquer" the waypoints to try to maximize the overall score of the team [18].

In [19], the authors cover multiple routing problems applied to sustainable transportation and show that TOPs are used more in unmanned aerial vehicles, while OPs are used for tourist trips. An example of an EV OP for tourist trips is [20] where time windows are used, battery recharge stations are along the route, and energy is a constraint. In [21], a TOP for EVs is mentioned, but does not explicitly state the objective function or constraints.

A gap in the literature is the use of a team orienteering problem for electric vehicles which maximizes the number of destinations visited by the team while minimizing the energy used by the vehicles.

2.1.2 Scheduling Problems

Scheduling problems are also a subset of combinatorial optimization problems. In general, scheduling problems consist of assigning tasks to someone or something to do the task, where the entity completing the task is commonly referred to as an agent. Task-agent pairs usually have a time element with them, where there are time constraints on the agent, the task, or both. Classical examples include the family of assignment problems, which assigns workers to jobs, and optimal job scheduling, which assigns jobs to different resources. Problems including the planning and scheduling of electric vehicles is of growing interest as road electrification increases [22]. This section covers previous works on open-, flexible-, and fixed-schedule problems, as well as the gaps in the literature.

In general, open-schedule problems focus more on the routing aspect than the charging aspect of the problem [11] [12], with a notable exception being [23] which does not consider routing at all. Furthermore, the distance is usually minimized instead of the cost of charging, and if the cost of charging is considered, the time-of-use costs are not considered, with a few exceptions such as [24] and [25]. To the best knowledge of the author, no open-schedule methods take power usage or the financial costs thereof into account.

Flexible-schedule problems are mainly focused on scheduling bus routes and charging sessions, and many of these problems take mixed fleets into consideration. In general, only a few flexible-schedule problems in the literature, such as [26], use heterogeneous electric vehicles. Generally, flexible-schedule problems consider distance, such as in [27], with others considering time, such as in [28]. However, many flexible-schedule problems consider energy more than distance, such as [29], which studies the base flexible-schedule problem. Pricing consideration strategies include [30], which considers the energy consumption costs of charging the vehicle fleet, and [31] and [32] which consider energy consumption costs as well as time-of-use costs. In [2], both power demand and energy consumption costs are used with time-of-use pricing.

Fixed-schedule problems focus on scheduling charging sessions without interfering with an existing schedule. Some fixed-schedule problems consider distance instead of energy, such

as open-schedule vehicles, with [33] being prominent among them. Pricing consideration strategies include considering energy consumption costs along with time-of-use pricing, with [34] as an example. Others, such as [35] and [36] consider demand costs with time-of-use pricing. Both [2] and [37] use power demand and energy consumption costs with time-of-use pricing.

A notable gap in the literature is the consideration and usage of the three types of schedules (open, flexible, and fixed) in a single problem [2]. The heterogeneous fleet in [2] consists of only flexible- and fixed-schedule vehicles.

2.1.3 Solution Techniques

There are multiple solution techniques for both routing problems and scheduling problems. The technique for solving routing problems can be lumped into three basic groups: exact approaches, market-based approaches, and metaheuristic approaches. Exact approaches find the optimal solution, but generally take a considerable amount of time to do so. Market-based approaches are fast, but generally perform poorly, quickly converging to a solution with no guarantees of optimality. Metaheuristic approaches, on the other hand, are a balance between the aforementioned approaches, combining speed and accuracy to create good solutions in a quick manner. They use heuristics on solutions generated by heuristics to iteratively find increasingly better solutions. Although these solutions generally tend to be closer to the optimal solution than market-based approaches, there is still no guarantee of optimality for metaheuristic approaches.

Since metaheuristic algorithms combine the advantages of speed and accuracy, they have become popular in solving routing problems. There exist several metaheuristic algorithms for optimization problems. However, there is no known way to objectively compare algorithms to find a universal solution technique that will be the best for every problem [38]. Two popular metaheuristic algorithms include genetic algorithms and ant colony optimization, both of which are discussed in detail in subsequent subsections.

Solution techniques for scheduling problems include exact, heuristic, and metaheuristic approaches. In a recent survey on electric-bus-scheduling strategies [39], the authors identi-

fied that in scheduling problems which aim to minimize operating costs, around 35% of the surveyed papers used exact methods, 11% used heuristic methods, 39% used metaheuristic methods, and 15% used hybrid methods which mixed either exact with heuristic methods or metaheuristic with heuristic methods. Of all the metaheuristic techniques used in the papers surveyed, there were no papers that used Ant Colony Optimization (ACO) to solve a scheduling problem.

However, ACO has been used in scheduling problems, such as in [28] which uses ACO to create routes and another algorithm to schedule them, and in [40] which uses ACO to convert a flexible-schedule problem into a fixed-schedule problem. It should also be noted that most formulations of a scheduling problem directly go to a mixed-integer, linear program, with some notable exceptions being to formulate the problem as a network-flow [41] or bin-packing [42] problem. Therefore, notable gaps in the literature include formulating a charging-scheduling problem as a routing problem and using ACO to solve a scheduling problem.

2.1.4 Ant Colony Optimization

Ant Colony Optimization (ACO) is a metaheuristic technique used to solve selective routing problems that simulates the behavior of ant colonies in nature. In nature, ants forage for food by traveling through different paths and leave a pheromone behind them for other ants to follow. Likewise, ACO leaves pheromones on edges, and use these pheromones to either increase or decrease the likelihood that an ant uses an edge in forming a solution. In general, paths that are good solutions to the optimization problem will get stronger pheromones, which, in turn, will lead the ants to find better solutions. Figure **??** shows a visual representation of ants exploring the solution space on a small graph.

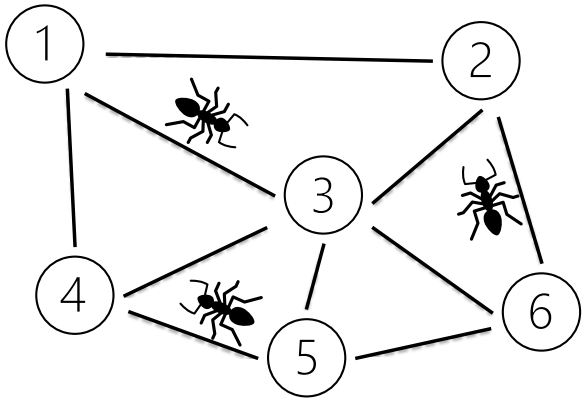


Fig. 2.1: A visual representation of ants exploring the solution space on a small graph

Ant Colony Optimization (ACO) was first introduced by Marco Dorigo in his doctoral thesis [43]. Early research on ACO had two main branches of ACO variation: improvement for general problems and improvement for specific families of problems [44]. Notable ACO variations for general problems include an update to the original algorithm by Dorigo [45], Parallel ACO [46], Rank-Based ACO [47], Max-Min ACO [48], Population-Based ACO [49], Hyper-Cube Framework ACO [50], and Elitist ACO [51]. The most popular and influential variants include the update from Dorigo [45] and Min-Max ACO [48]. Specific families of problems include multiobjective optimization [52], which has become very popular [53], stochastic optimization [54], integer nonlinear optimization problems [55], and continuous optimization problems [56]. However, ACO approaches to continuous optimization problems did not take off, but instead hybridized with other techniques [44]. In fact, there has been a major shift from the development of new ACO variations to the hybridization of ACO with other metaheuristic techniques [44].

Further families of problems that have been solved with ACO, outside of selective routing problems, include traditional scheduling problems such as the sequential ordering problem (SOP) [57] and the open-shop scheduling problem (OSP) [58], as well as assignment problems such as the generalized assignment problem [59], and set problems such as the multiple knapsack problem [60].

2.1.5 Genetic Algorithms

Another popular metaheuristic technique for solving optimization problems is through the use of a genetic algorithm (GA). Genetic algorithms are based on Darwin's theory of evolution. The main elements of a genetic algorithm are a population of solutions, a fitness function to compare solutions, and operators for selection, crossover, and mutation. The selection process imitates natural selection by comparing solutions to the fitness function and ensuring the "survival of the fittest." The crossover operation creates one or more child solutions using traits from two or more parent solutions, and the mutation operator adds additional changes from parent to child. The challenges of genetic algorithms include creating an appropriate schema for population creation and premature convergence [61]. Figure 2.2 shows a visual representation of a small population of chromosomes as solutions to a single-vehicle problem.

Fig. 2.2: A visual representation of multiple chromosomes as solutions to a single-vehicle problem

Given the extensive usage of genetic algorithms, there are several unique ways in the literature to perform selection, crossover, and mutation operators. The remainder of this subsection details information regarding these operators.

Some popular selection schemes include the roulette-wheel method, stochastic universal sampling, linear rank selection, exponential rank selection, tournament selection, and truncation selection [62]. [61] lists the advantages and disadvantages of several popular techniques. One of these techniques is the roulette-wheel method, which is used in [1] and this work. The roulette-wheel method has the advantages of being simple, free from bias, and easy to implement. However, the roulette-wheel method has the disadvantages of risking premature convergence and depends on the variance present in the fitness function [61].

Some crossover schemes for distance-constrained VRPs include sequential constructive crossover, cycle crossover, partially mapped crossover, and alternate edge crossover [63]. In [63], the authors identified that the sequential constructive crossover is the best-suited

crossover operation for distance-constrained VRPs.

Some popular mutation schemes include displacement mutation, inversion mutation, scramble mutation, and reversing mutation [61]. In [1] and this work, mutations for insertion, inversion, and swap are used.

2.2 Contributions

The contributions of this work are the formulation of a routing problem that aims to minimize energy and maximize visited vertices, a novel genetic algorithm based on [1] to solve the routing problem, a novel ant colony optimization algorithm to solve the routing problem, the novel formulation of an EV charge-scheduling problem as a routing problem, and an ant colony optimization algorithm to solve the EV charge-scheduling problem.

CHAPTER 3
ROUTING

Between the three types of schedules in this study (open, flexible, fixed), only open schedules contain both a routing and scheduling aspect to them. This chapter focuses primarily on open-schedule vehicle routing and scheduling and two novel, metaheuristic, solution techniques. The method of solving the heterogeneous scheduling problem in this work has been separated into two stages, namely (1) representing all schedule types as flexible schedules, and (2) solving the problem with the newly-converted, homogeneous schedules. Thus, for open-schedule problems, the first step is to create energy-aware, flexible-schedule routes that can be used in a scheduling paradigm. This chapter details solving the routing part of open-schedule problems and leaves the scheduling part to be solved in Chapter 4.

The contributions of this chapter are two new solvers to an asymmetric routing problem with time windows and distance constraints, a problem that to the best of the author's knowledge has not been solved in the literature. Beyond adopting a new cost function, the solvers also have novel route-generating strategies.

The remainder of this chapter discusses the formulation of the routing problem, thorough details of a hybrid genetic algorithm, the ant colony solution itself, a comparison between the two approaches, and implementation details of a realistic scenario. The chapter ends with the use of ACO-generated routes in a scheduling paradigm.

3.1 Formulation of the EATOP

In order to solve the routing problem, it must be defined. This section describes the formulation of the routing problem used in this work. The routing problem herein is an orienteering problem at its core, consisting of a set of labeled vertices to be visited, edges to connect the vertices, agents to visit the vertices, an objective to optimize, constraints to consider, and solutions to select: all of which are detailed in the following paragraphs.

Every routing problem has at least one agent to visit or otherwise serve vertices. The agents that visit the vertices are electric vehicles and are simply referred to as *vehicles*. The problem is intricately intertwined with the vehicles, since vehicles travel along the edges to visit the vertices, the solutions to the problem are created for each vehicle in the form of **routes**, and the constraints of the problem are due to the constraints of the vehicles. The constraints of the vehicles themselves are mainly the range of the vehicles and environmental constraints, such as the timing of the vehicles.

Now, given that the agents in a routing problem visit vertices, it is important to discuss the vertices of the problem. The vertices are divided into two groups: where a vehicle starts and/or ends its journey, called *depots*, and where a vehicle simply visits, called *destinations*. Note that in an orienteering problem, not all destination vertices must be visited for a legitimate solution. Furthermore, the vertices of the problem may inhabit the same physical space. Therefore, the theoretical formulation does not change in single- or multiple-depot versions of the problem. In cases where two or more vehicles share a depot, an electric vehicle may be able to complete multiple routes, given that the times the vehicles are on route do not overlap and there is enough time spent at the depot to charge the EV. Furthermore, each vertex cannot be visited more than once, and there exist vertices that must be visited during certain time windows.

In order for the agents to travel between one vertex to the next, there must be a set of edges that enable movement between vertices. The edge set, along with the vertex set, creates a complete directed graph, or in other words, for any vertex in the problem, there exists a directed edge from that vertex to any other vertex in the problem. Two costs are associated with each edge: the first being the energy expenditure, or how much energy, in kilowatts, it takes the vehicle to travel from one vertex to the next, and the second being the time cost, or how much time it takes, in seconds, to travel from one vertex to the next. Furthermore, the cost of traveling between two depots is defined to be infinite.

Given the agents, vertices, and edges, the problem can be constructed with an objective and constraints. The objective of the orienteering problem herein is to maximize an overall

score based on the number of destinations visited and the energy used by the EVs. Both the number of destinations and the energy used are scaled to produce results that aim to simultaneously maximize the number of destinations and minimize the amount of energy used by the fleet. The constraints of the problem herein are included to better reflect the reality of operating a vehicle fleet and consist of time windows and energy constraints. Time windows allow for any timing constraints placed on situations such as package delivery and can represent a number of scenarios including different employee shifts or deadlines from the customer. The energy constraint is due to the battery capacity of the vehicle, although the constraint does not need to be the full capacity of the battery.

Given the objective and constraints of the problem herein, the solution is represented as a set of routes for each vehicle. The set of routes with the highest score has aimed to maximize the number of destinations visited while simultaneously minimizing the overall energy usage of the fleet, finding a balance between the two objectives. The problem herein is titled the Energy-Aware Team Orienteering Problem for Electric Vehicle Routing with Time Windows and Energy Constraints (EATOP). The remainder of this section covers the ideas of routes and constraints, and the section ends by describing the EATOP in greater detail.

3.1.1 Routes

Given a high-level view of the problem, it is imperative to more rigorously describe the EATOP and its solution. This subsection provides the definitions necessary to understand the problem, the notion of a route, and route-related terms.

The EATOP itself is formally defined using a weighted, directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Each destination is represented as a vertex, $v \in \mathcal{V}$, and a connection between two destinations is represented by an edge, $e = (i, j) \in \mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. The graph in the EATOP is fully connected, i.e. $(i, j) \in \mathcal{E}\ \forall i \neq j$.

The solution to the EATOP is a set of routes. A route is defined to be a finite sequence of labels that correspond to the vertices that the vehicle visits, in the order that the vehicle visits, starting and ending with the vehicle's depot. Without loss of generality, the depot

of each vehicle corresponds to the vehicle's number, i.e., vehicle 0 starts at vertex 0, vehicle 1, starts at vertex 1, etc. Let m be the total number of vehicles and n be the number of vertices, $n = |\mathcal{V}|$. Therefore, there are $n - m$ *destinations*.

The route for each vehicle is determined using the decision variable x_{ij}, which equals one when the edge from i to j is used in a solution and equals zero otherwise. Given a solution, the route for each vehicle can be extracted as a finite sequence, $\mathcal{P}_k$, that contains the vehicle's depot index. For ease of expressing the vertices of a route in the order that they are visited, the l^{th} vertex of route $\mathcal{P}_k$ is denoted as $\sigma_l^k \in \mathcal{V}$. The route is of length n_k, and thus the route may be expressed as

$$\mathcal{P}_k = (\sigma_0^k, \sigma_1^k, \ldots, \sigma_{n_k}^k). \tag{3.1}$$

Note that if $\mathcal{P}_k = (\sigma_0^k, \sigma_1^k, \ldots) = (i, j, \ldots)$, then $x_{ij} = 1$ and $i = k$, as the vehicle will start at its depot. Also, note that since only one vehicle will visit a vertex, $\sigma_l^i \notin \mathcal{P}_j$ for every $l \in \{0, \ldots n_i - 1\}$, and for every $i \neq j$. In other terms, the set of all vertices in the finite sequence $\mathcal{P}_i$ and the set of all vertices in the finite sequence $\mathcal{P}_j$ are disjoint. Furthermore, note that the total number of vertices visited by the vehicles form a subset of the total number of vertices, and generally form a proper subset of all vertices. The way that the solution is represented is as a set of routes $\mathcal{R}$, which consists of every $\mathcal{P}_k$ in which the vehicle k completes a route, or $\mathcal{R} = \{\mathcal{P}_k\} \;\; \forall k \in \{0, \ldots, m\}$.

3.1.2 Energy and Time-Window Constraints

Beyond routes, important elements of the EATOP include energy and time-window constraints. This subsection discusses both types of constraints and defines a schema of how to calculate the time and energy of a vehicle at a given vertex.

The energy constraints of the EATOP are due to the nature of electric vehicles and their batteries. The energy constraint of the vehicles is denoted as $e_{\max}$ and the energy used by the vehicle to travel from a vertex i to a vertex j is denoted as e_{ij}. The energy constraint could be the battery capacity of the vehicle; however, in general, it is better for

the longevity of the battery to not use its full capacity.

Time windows are crucial in many applications, such as package delivery, where package delivery is constrained by the timing of an employee's shift and breaks, or perhaps the package is being delivered to a customer that has a deadline, to name a few examples. The time window for a vehicle to visit vertex j is denoted as an ordered pair, $(\tau_{e,j}, \tau_{l,j})$, where the first entry is the earliest time a vehicle is allowed to visit vertex j and the second entry is the latest time a vehicle is allowed to visit vertex j.

To ensure that the time-window constraints are not violated, a schema for calculating the time that the vehicles arrive and depart from destinations is necessary, and therefore is presented here. The starting time of a vehicle is denoted as t_0^k. The dwell time at vertex i plus the time it takes to travel from vertex i to vertex j, is denoted as t_{ij}. The time that a vehicle arrives at a location is denoted as t_j^k and is found by adding the time costs of each vertex in the path of vehicle k. If vehicle k does not visit j, $t_j^k = 0$ by definition.[1] Mathematically, the time it takes for a vehicle k to reach a vertex j is

$$t_j^k = \begin{cases} t_0^k + \sum_{i=0}^{\sigma_i^k = j-1} t_{\sigma_i^k, \sigma_{i+1}^k} & \text{if } j \in \mathcal{P}_k \\ 0 \quad \text{otherwise.} \end{cases} \tag{3.2}$$

In a similar fashion, the energy usage of a vehicle can be calculated by replacing the time costs with energy costs. Using methods such as these to calculate cumulative time and energy is crucial to checking if a solution adheres to the time-window and energy constraints which reflect real-world applications.

3.1.3 The Optimization Problem

Now, given the background of routes and constraints, the objective and constraints of the EATOP can be fully outlined. This subsection formally defines the optimization problem and concludes Section 3.1.

[1] This will be clearer in the constraint definitions of the optimization problem in Section 3.1.3.

As an optimization problem, the EATOP is defined as a maximization of a utility function, where the utility is the overall score minus the overall cost. The overall score factors in the number of locations visited, multiplied by a scaling factor $\bar{\alpha}$. The cost factors in both the total energy used by the vehicles multiplied by a scaling factor $\bar{\beta}$.

The objective and constraints are defined as follows, with $|\mathcal{P}_k|$ denoting the length of the sequence $\mathcal{P}_k$:

$$\text{Maximize } -\left(\bar{\beta}\sum_{j=0}^{n}\sum_{i=0,\ i\neq j}^{n} e_{ij}\ x_{ij}\right) + \bar{\alpha}\sum_{k=0}^{m}|\mathcal{P}_k|$$

subject to

$$\begin{aligned}
&(1) \quad \sum_{i=0,\ i\neq j}^{n} x_{ij} = \sum_{i=0,\ i\neq j}^{n} x_{ji} \quad \forall j \in \{0,\ldots,n-1\} \\
&(2) \quad k = \sigma_0^k = \sigma_{n_k}^k \quad \forall k \in \{0,\ldots,m-1\} \\
&(3) \quad \sigma_l^i \notin \mathcal{P}_j \quad \forall l \in \{0,\ldots|\mathcal{P}_j - 1|\}, \quad \forall i \neq j. \\
&(4) \quad \sum_{i\in\mathcal{P}_k}\sum_{j\in\mathcal{P}_k} x_{ij} \leq |\mathcal{P}_k| - 1 \ \ \forall k \in \{0,\ldots,m-1\} \\
&(5) \quad \sum_{i\in\mathcal{P}_k}\sum_{j\in\mathcal{P}_k} e_{ij}\ x_{ij} \leq e_{\max} \ \ \forall \mathcal{P}_k \in \mathcal{R} \\
&(6) \quad \tau_{e,i} \leq t_i^k \leq \tau_{l,i} \ \ \forall i \in \mathcal{P}_k, \ \ \forall \mathcal{P}_k \in \mathcal{R}
\end{aligned} \tag{3.3}$$

Constraint 1 ensures that any vehicle that travels to vertex j will travel out of j [64]. Constraint 2 ensures that the route for each vehicle k starts and ends at the depot. Constraint 3 ensures that each visited vertex is visited by one vehicle. Constraint 4 ensures that there are no subtours for any vehicle's tour [64]. The left side of the inequality counts edges in vehicle k's path, while the right side of the inequality counts the vertices. To have no subtours, $\mathcal{P}_k$ needs to have one less edge than the number of vertices. Constraint 4 thus ensures that there is a path from the first vertex in $\mathcal{P}_k$ to the last vertex in $\mathcal{P}_k$. Constraint 5 is the energy constraint, where each vehicle has an upper limit of how much energy it can consume. Constraint 6 is the time window constraint, ensuring that a vehicle k arrives at vertex i during the time window.

The notion of a set of routes as a solution to the EATOP, the importance and formal definitions of the energy and time-window constraints, as well as the formal definitions of the the objective and constraints of the EATOP make up the formulation of the problem. Given this formulation of the problem, the subsequent sections focus on solution techniques for the EATOP.

3.2 Solving the EATOP Using a Genetic Algorithm Approach

Given the formulation of the problem in Section 3.1, the next step is to introduce a solution to the EATOP. The solution presented in this section is a genetic algorithm approach.

Genetic algorithms (GAs) are metaheuristic techniques that solve optimization problems by simulating the evolutionary process. The specific genetic traits of an individual are encoded by chromosomes. Within the evolutionary process, there are various mechanisms that change a species from one generation to another. Chromosomes undergo changes through a mechanism called crossover, in which a new chromosome is created by receiving traits from two parents. In addition, mutations can occur from a parent to a child, causing changes in genetic traits.

In a routing problem, a solution is called a *chromosome*. A GA starts with a population of chromosomes and runs its iterative process for a specified number of generations. During each generation, the population undergoes a selection process that simulates natural selection, where each chromosome is compared against a fitness function, which is usually the objective function of the optimization problem. Chromosomes that are more fit are saved to reproduce, following the Darwinian idea of "survival of the fittest." After the selection process, the population undergoes a crossover operator, which creates a child chromosome that receives traits from both parents, and a mutation operator, which causes changes in the child chromosome. Some genetic algorithms, such as the GA herein, include additional operations to create diversity in the population and explore the solution space further.

The EATOP hybrid genetic algorithm (EHGA) presented herein is adapted from the GA-ADP, a hybrid genetic algorithm from [1]. The authors of [1] proposed one simple

GA and four HGAs, where the simple HGA generated an initial population that undergoes selection, crossover, and mutation until the maximum number of generations is reached. The four HGAs include the simple GA with the immigration method plus the inclusion of insertion, inversion, swap, or the random selection of one of the three local searches. The HGAs in [1] were designed to solve the asymmetric, distance-constrained, vehicle routing problem (ADVRP) whose objective is to find the shortest routes for the vehicle fleet, or, in other words, the cost is defined by the total distance traveled. The HGAs were implemented for a single-depot instance of the ADVRP. Each vertex is only visited once, vehicles start and end at the depot, and there are no return visits to the depot. The similarities between the ADVRP and the EATOP are that both have asymmetric costs as well as constraints on how far a vehicle may travel.

The results of [1] showed that the GA-ADP, which randomly selects one of the three local searches, performs the best. Given the similarities between the ADVRP and EATOP, the GA-ADP was chosen to be modified to solve the EATOP through the introduction of some new algorithms and functionalities. The main differences between the EHGA and the HGAs of [1] are, namely, the evaluation of utility instead of distance for fitness, using zero-indexing, and explicitly stating each step of each algorithm. The remainder of this section details various elements of the EHGA, including notation, supporting algorithms, operators, and the EHGA itself. In this section, every difference between EHGA and GA-ADP is explicitly stated.

3.2.1 Notation for the EHGA

This section details the notation used in the EHGA solution technique. Table 3.1 shows the notation used for the HGAs and their supporting algorithms, and Table 3.2 shows the structures and other variables used in the HGA. In general, subscripts such as "best" or a numeral are used to distinguish objects, such as $\mathcal{C}_1$ and $\mathcal{C}_2$ denoting two different chromosomes or $\mathcal{C}_{\text{best}}$ denoting the best-performing chromosome. A "star" superscript denotes a copy of a variable, i.e. $\mathcal{C}^*$ is a copy of $\mathcal{C}$. Note that $\mathcal{P}_k$ is now represented as a list. The implementation details of the values of these parameters are found in Section 3.4.

Name	Description	Name	Description
Graph and Optimization Variables			
$\mathcal{G}$	The graph used in the problem	$\bar{\alpha}$	The weight on the number of destinations visited
$\mathcal{E}$	The edge set of the graph.	$\bar{\beta}$	The weight on the total energy consumed by the fleet
$\mathcal{V}$	The vertex set of the graph, representing the destinations		
x_{ij}	A decision variable, which equals one if the edge (i, j) is used and is zero otherwise		
Parameters			
N	The number of generations to run the algorithm	$\mathcal{T}$	The list of starting times
		$\mathcal{W}$	The list of time windows
p_s	The desired population size	$\tau_{e,j}$	The earliest time a vertex j can be visited
m	The number of vehicles in the problem	$\tau_{l,j}$	The latest time a vertex j can be visited
n	The number of vertices		
$e_{\max}$	The maximum energy that vehicle k can use	p_c	The probability of crossover
		p_m	The probability of mutation
Indices, Iterators and Vertices			
i	Used as an index or iterator	v	Used as a vertex
j	Used as an index or iterator	p	Used as a vertex
k	Used as an index or iterator	q	Used as a vertex
		c	The current vertex
Operators			
$\oplus$	An operator where $A \oplus b$ appends element b to (the end of) a set or list A	$\oslash$	An operator where $A \oslash b$ divides every element of A by b
$\ominus$	An operator where $A \ominus b$ removes element b from a set or list A		

Table 3.1: Notation of the HGA Process

Name	Description	Name	Description
Structures			
$\mathcal{C}$	A chromosome or solution	P	The probability matrix
$\mathcal{P}$	The population of chromosomes	T	The time cost matrix
$\mathcal{R}$	The solution as a set of routes	$\vec{e}$	The vector of "current energies"
$\mathcal{P}_k$	The path or route of vehicle k expressed as a list	$\vec{t}$	The vector of "current times"
		$\vec{v}$	The vector of "current vertices"
$\mathcal{A}$	The list of vertices after a certain vertex	$\vec{w}$	The weights vector
		$\vec{u}$	A vector for storing utilities of the population
$\mathcal{B}$	The list of vertices before a certain vertex	$\vec{f}$	A vector for storing the fitnesses of the population
$\mathcal{S}$	A list of randomly-selected indices		
$\mathcal{Q}$	The index list of integers which correspond to a specific entry in a vector	$\vec{p}$	A vector for storing the probabilities of the population
		$\vec{c}$	A vector for storing the cumulative probabilities of the population
E	The energy cost matrix		
E'	The element-wise, inverse energy matrix		
Other Variables			
e	The energy of a solution	t	The time of a solution
n_v	The number of vehicles used in a solution	n_p	The number of phantom depots
		n_i	The number of immigrants
n_d	The number of destinations used in a solution	i_c	An iteration counter
		r	A randomly-generated number
u	The utility of a solution		

Table 3.2: HGA Structures and Variables

3.2.2 Supporting Algorithms for the EHGA

Before a description of the EHGA or its operators is given, it is crucial to introduce important concepts and supporting algorithms which are used in the EHGA process. First, the relationship between a chromosome and a route is presented and accompanied by an algorithm that converts a chromosome into a set of routes. Second, novel algorithms are defined for computing the fitness of a chromosome and evaluating constraints. Third, some supporting algorithms for the population creation algorithm are presented and, finally, the novel population creation algorithm is discussed.

Chromosomes and Routes

Given the background in Section 3.1, as well as the ideas of this section, it is important to relate the notion of a chromosome to the idea of a route. In the GA-ADP, a solution, or *chromosome* is defined to be a path whose length is at most $n + m - 1$, with n as the number of vertices and m as the number of vehicles in the solution. The extra $m-1$ vertices represent a return to the depot and separate different routes. These vertices are known as *phantom depots*. The zero-indexed version of an example of a chromosome, $\mathcal{C}$, in [1] uses 3 vehicles and 10 destinations:

$$\mathcal{C} = [0, 4, 2, 10, 9, 7, 1, 6, 11, 5, 3, 8].$$

In this example, the first vehicle leaves the depot, vertex 0, then visits vertices 4 and 2. Vertex 10 is a phantom depot, and thus ends the first vehicle's route and starts the second vehicle's route. The second vehicle departs from the depot and visits vertices 9, 7, 1, and 6 before finishing its route with the phantom depot, vertex 11. The third vehicle then departs from the depot and visits vertices 5, 3, and 8.

A useful function is the ability to convert chromosomes to routes, where every vehicle starts and ends at the depot, vertex 0, $\mathcal{R} = \{\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3\}$, and

$$\mathcal{P}_1 = [0, 4, 2, 0]$$
$$\mathcal{P}_2 = [0, 9, 7, 1, 6, 0]$$
$$\mathcal{P}_3 = [0, 5, 3, 8, 0].$$

This functionality is described by Algorithm 1, which is not in [1].

Algorithm 1 converts a chromosome $\mathcal{C}$ to a set of routes $\mathcal{R}$, given the chromosome and the number of vertices in the problem, n. The algorithm starts by initializing both the routes variable $\mathcal{R}$ and the path variable $\mathcal{P}$ as empty on line 1. The **for** loop on line 2 iterates through each vertex in the chromosome. Line 3 compares the vertex's label with the number of vertices in the problem. If the label is less than the number of vertices in

the problem, then the vertex i is assigned to the path $\mathcal{P}$, where $\mathcal{A} \oplus b$ signifies adding the element b to the end of list $\mathcal{A}$. Otherwise, i corresponds to a phantom depot, and thus the path ends and a new path begins. The completed path is added to the set of routes on line 7, and a new path is started at the depot on line 8. After the **for** loop ends, line 11 adds the last path to the set of routes, and the algorithm returns the set of routes.

Algorithm 1 Convert Chromosome to Routes $\mathcal{R} \leftarrow$ `chromosome_to_routes`$(\mathcal{C}, n)$

```
1: R ← {}; P ← []
2: for i = 0, ... |C| do
3:     if i < n do
4:         P ⊕ i // Add vertex to route
5:     else do
6:         P ⊕ 0 // Add return to depot
7:         R ⊕ P // Add path to routes
8:         P ← [0] // Start a new path at the depot
9:     end if
10: end for
11: R ⊕ P // Add last path to set of routes
12: return R
```

Another useful function is to convert routes into chromosomes, which is shown in Algorithm 2. Algorithm 2 begins by creating a chromosome that contains only the depot and initializing the phantom depot counter n_p. The **for** loop on line 3 iterates through each route in the set of routes and the **for** loop on line 5 iterates through each vertex in the route. Each non-depot vertex is added to the chromosome on line 7, where $\mathcal{A} \oplus b$ signifies adding the element b to the end of list $\mathcal{A}$. After adding the vertices of a route, a phantom depot is created and added to the chromosome on line 10. The phantom depot counter increases on line 11. After adding all vertices and phantom depots to the chromosome, the extra phantom depot is removed on line 13, where $\mathcal{A} \ominus b$ signifies taking the element b out of list $\mathcal{A}$. The chromosome is then returned from the algorithm.

Algorithm 2 Convert Routes to Chromosome $\mathcal{C} \leftarrow$ `routes_to_chromosome`$(\mathcal{R}, n)$

```
1: C ← [0]
2: n_p ← 0 // Initialize phantom depot counter
3: for i = 0, ... |R| do
4:     P_k ← R[k] // Extract route for vehicle k
5:     for j = 0, ..., |P_k| do
6:         if j ≠ 0 do
7:             C ⊕ P_k[j] // Add vertex to chromosome
8:         end if
9:     end for
10:    C ⊕ n_p + n // Add a phantom depot
11:    n_p ← n_p + 1 // Update counter
12: end for
13: C ⊖ C[|C| − 1] // Remove extra phantom depot
14: return C
```

Fitness, Constraints, and Best Solutions

Beyond converting chromosomes and routes, the EHGA technique incorporates supporting algorithms to determine which solutions are "most fit." To ensure the survival of the fittest, there must be a way to determine the fitness of a chromosome and to check for constraint violations. Algorithm 3 calculates the fitness, or *utility*, of a solution by calculating the total energy consumption of the fleet and the number of destinations visited. Algorithm 4 checks a chromosome for constraint violations, and Algorithm 5 determines the best-performing chromosome in the population.

Algorithm 3, which is not in [1], calculates the utility of a chromosome, given the energy cost matrix, E, the number of vertices in the problem, n, the chromosome $\mathcal{C}$, and the weights used in the objective function. The weights include the number of destinations visited and the energy used by the fleet: $\bar{\alpha}$ and $\bar{\beta}$, respectively. Algorithm 3 starts by initializing the total energy of the fleet as zero and the number of employed vehicles as one. The **for** loop on line 2 iterates through the chromosome to sum the energy usage of the fleet and count the vehicle employment. Line 3 checks to see if the destination vertex is a phantom depot, where, again, any phantom depot is labeled a number higher than the number of vertices in the problem. If the vertex is a phantom depot, then the number of

vehicles employed increases by one on line 4 and the energy to return to the depot is added to the total energy on line 5. Otherwise, line 7 adds the energy costs of the edges used in the solution. After the **for** loop finishes, the number of destinations n_d is calculated on line 10. The utility is then calculated on line 11, and the algorithm returns the utility.

Algorithm 3 Calculate Utility $u \leftarrow$ `calc_util`$(E, n, \mathcal{C}, \bar{\alpha}, \bar{\beta})$

```
1:  e ← 0; n_v ← 1
2:  // Iterate through chromosome to sum energy usage and count vehicle employment
    for i = |C| − 1 do
3:      if C[i + 1] > n do
4:          n_v ← n_v + 1
5:          e ← e + E[C[i]][0] // Add energy to return to depot
6:      else do
7:          e ← e + E[C[i]][C[i + 1]] // Add energy from route
8:      end for
9:  end for
10: n_d ← |C| − n_v // Calculate number of destinations visited
11: u ← ᾱn_d − β̄e // Calculate utility
12: return u
```

Algorithm 4, which is not in [1], checks a solution to see if any constraints are violated and returns true if any constraint is violated. It checks if a solution employs all available vehicles and if a solution violates the energy or time window constraints. The chromosome is converted into a set of routes on line 2 and the cardinality of $\mathcal{R}$, the number of vehicles employed, is compared against the length of $\mathcal{T}$, the number of available vehicles on line 3. If the employment does not equal the number of available vehicles, the solution is not valid, and the algorithm returns `true` on line 4. If the solution passes the employment check, the energy and time constraints are checked using the **for** loop on line 7. The route of a vehicle k is extracted from the set of routes on line 8, the vehicle's starting time, t, is extracted from the list of starting times, $\mathcal{T}$, and the energy variable e is initialized on line 10.

The next part of the algorithm iterates through each vertex on the path and checks the constraints. The inner **for** loop on line 11 iterates through the vertices on the path and calculates the time and energy consumed by running a route on lines 13 and 15, respectively,

following the form used in Equation 3.2. The time window is extracted on line 16. If the time is outside the window, the algorithm returns `true` on line 18. Finally, the algorithm checks the solution for an energy-constraint violation. The total energy of the route is compared with $e_{\max}$ on line 21. If the energy constraint is violated, the algorithm returns `true` on line 22. If no constraints are violated, then the algorithm returns `false` on line 25.

Algorithm 4 Constraint Checking `bool` $\leftarrow$ `constraints_violated`$(E, T, \mathcal{T}, \mathcal{W}, n, e_{\max}, \mathcal{C})$

1: // Check number of vehicles constraint
2: $\mathcal{R} \leftarrow$ `chromosome_to_routes`$(\mathcal{C}, n)$ // Convert to routes using Algorithm 1
3: **if** $|\mathcal{R}| \neq |\mathcal{T}|$ **do** // The number of vehicles is $|\mathcal{T}|$
4: **return** `true`
5: **end if**
6: // Check energy and time window constraints
7: **for** $k = 0, \ldots, |\mathcal{R}|$ **do** // Iterate through routes
8: $\mathcal{P}_k \leftarrow \mathcal{R}[k]$
9: $t \leftarrow \mathcal{T}[k]$ // Starting time for vehicle k
10: $e \leftarrow 0$
11: **for** $j = 0, \ldots, |\mathcal{P}_k| - 1$ **do** // Iterate through path to find time and energy
12: **if** $j \neq 0$ **do**
13: $t \leftarrow t + T[[\mathcal{P}_k[j-1]][\mathcal{P}_k[j]]$ // Sum time
14: **end if**
15: $e \leftarrow e + E[[\mathcal{P}_k[j]][\mathcal{P}_k[j+1]]$ // Sum energy
16: $(\tau_{e,\sigma}, \tau_{l,\sigma}) \leftarrow \mathcal{W}[\mathcal{P}_k[j+1]]$ // Extract time window for vertex $\sigma = \mathcal{P}_k[j+1]$
17: **if not** $\tau_{e,\sigma} \leq t \leq \tau_{l,\sigma}$ **do**
18: **return** `true` // Exit if outside the time window
19: **end if**
20: **end for**
21: **if** $e > e_{\max}$ **do**
22: **return** `true` // Exit if energy constraint is violated
23: **end if**
24: **end for**
25: **return** `false`

Algorithm 5, which is not explicitly defined in [1], finds the best-performing chromosome of the population and is not explicitly featured in [1]. However, Algorithm 5 is crucial in determining the best solution generated by the genetic algorithm and allows the best solution to be saved and compared against future solutions. It takes in the necessary parameters to find the utility of each solution, calculates the utility of each function, and

returns the chromosome with the highest utility. The algorithm begins by initializing a vector, $\vec{u} \in \mathbb{R}^{|\mathcal{P}|}$, as all zeros on line 1. The **for** loop on line 2 iterates through each chromosome in the population, finds its fitness, and adds its fitness to the utility vector $\vec{u}$. The chromosome is extracted on line 3, the utility is calculated on line 4, and the utility is added to $\vec{u}$ on line 5. After $\vec{u}$ is populated, the chromosome associated with the maximum value of $\vec{u}$ is returned on line 7.

Algorithm 5 $\mathcal{C} \leftarrow$ `get_best_chromosome`$(E, n, \mathcal{P}, \bar{\alpha}, \bar{\beta})$

1: $\vec{u} \in \mathbb{R}^{|\mathcal{P}|} \leftarrow [0]$
2: **for** $i = 1, \ldots, |\mathcal{P}|$ **do**
3: $\quad \mathcal{C}_i \leftarrow \mathcal{P}[i]$
4: $\quad u_i \leftarrow$ `calc_util`$(E, n, \mathcal{C}_i, \bar{\alpha}, \bar{\beta})$ // Calculate utility via Algorithm 3
5: $\quad \vec{u}[i] \leftarrow u_i$
6: **end for**
7: **return** $\mathcal{P}[$`argmax`$(\vec{u})]$

Population Creation and Local Optimization

Given ways to convert chromosomes and routes, calculate fitness, observe constraints, and find the best solutions, population creation and local optimization become easier. The remainder of this section on supporting algorithms for the EHGA process discusses population creation and local optimization. The local optimization technique is first discussed and is used in the population creation algorithm.

The local optimization technique is a modified 2-opt algorithm. A 2-opt algorithm is a solution to the traveling salesman problem, which is a single-vehicle version of the vehicle routing problem [65] [66]. It can be modified slightly to handle more complex routing problems, such as the EATOP. A 2-opt algorithm methodically changes the order of vertices in a route until there are no improvements. It selects two vertices, reverses the order of vertices between the two selected vertices, and keeps the change if it improves the solution. The authors of [1] did not include an algorithm for their 2-opt approach in their work. In this work a modified 2-opt algorithm for the EATOP is presented.

Algorithm 6 shows the modified 2-opt algorithm used for the EHGA. The algorithm

takes in energy and time costs, constraints, and the chromosome to be improved and returns the improved chromosome. The algorithm starts by assigning the chromosome that is passed in as the best chromosome on line 1 and calculates the energy of the chromosome on line 2. The main loop starts on line 4 and repeats until there is no improvement. The nested **for** loops on lines 6 and 7 iterate through the vertices in the chromosome. The algorithm continues by copying the chromosome $\mathcal{C}$ on line 8, and the order of the vertices between i and j is reversed on line 9. The energy of the copied chromosome is then calculated on line 10. The energy of the copy is compared against the original chromosome and the chromosome's observance of the constraints of the problem is checked on line 11. If the solution has been improved and the constraints are not violated, then the solution is updated on line 12. Otherwise, the best solution is returned on line 19. After the nested for loops are complete, the chromosome is replaced by the best-performing chromosome on line 17.

Algorithm 6 $\mathcal{C} \leftarrow$ `two_opt`$(E, T, \mathcal{T}, \mathcal{W}, n, \mathcal{C}, e_{\max})$

$\mathcal{C}_{\text{best}} \leftarrow \mathcal{C}$ // Initialize best chromosome
$e_{\text{best}} \leftarrow \sum_{l=0}^{|\mathcal{C}|-1} E[\mathcal{C}[l-1]][\mathcal{C}[l]]$ // calculate total energy
`improved` $\leftarrow$ `true`
while `improved` **do**
    `improved` $\leftarrow$ `false`
    for $i = 1, \ldots, |\mathcal{C}| - 2$ **do**
        for $j = i + 1, \ldots |\mathcal{C}| - 1$ **do**
            $\mathcal{C}^* \leftarrow \mathcal{C}$
            // Reverse the order of the vertices between i and j
            $\mathcal{C}^*[i:j] \leftarrow \mathcal{C}[j-1:i-1:-1]$
            $e \leftarrow \sum_{l=0}^{|\mathcal{C}^*|-1} E[\mathcal{C}^*[l-1]][\mathcal{C}^*[l]]$
            // Compare against best and run Algorithm 4 to check constraints
            if $e < e_{\text{best}}$ **and not** `constraints_violated`$(E, T, \mathcal{T}, \mathcal{W}, n, e_{\max}, \mathcal{C}^*)$
                $\mathcal{C}_{\text{best}} \leftarrow \mathcal{C}^*$; $e_{\text{best}} \leftarrow e$
                `improved` $\leftarrow$ `true`
            end if
        end for
    end for
    $\mathcal{C} \leftarrow \mathcal{C}_{\text{best}}$
end while
return C_{best}

The population creation algorithm is Algorithm 1 in [1]. In [1], the population creation algorithm iterates p_s times, where p_s is the desired population size. During the iterative process, a new chromosome is created with the depot as its first element. Then, a list of remaining vertices is created. A second loop randomly selects a vertex q from the remaining cities and checks if adding q to the chromosome violates the distance constraint. If the distance constraint is not violated, then q is taken out of the set of remaining vertices and added to the chromosome. Otherwise, a phantom depot is added to the chromosome. After the population is created, each chromosome is improved using a local optimization technique called the 2-opt algorithm [65] [66]. The population creation algorithm in [1] is very effective in solving the ADVRP. However, given the constraints of the EATOP, namely, a finite number of available vehicles and time-window constraints, adding a phantom depot

after a constraint is violated is not suitable for solving the EATOP.

The population creation algorithm is shown as Algorithm 7 and begins with an initialization process. The population is initialized as an empty set on line 1. In the subsequent lines, a probability matrix is created with weights based on energy costs. First, an element-wise, inverse matrix of E, labeled E' is created on line 3. Then, the sum of all the elements of E' is computed on line 4. The probability matrix is then calculated by performing element-wise division, where each element of E' is divided by e on line 5.

After the initialization process, the population creation process begins by creating new solutions and preparing the solutions for the assignment process. The **for** loop on line 6 iterates for the desired population size p_s, creating a new chromosome during each iteration. It starts with creating a set of routes with each route starting at the depot. The probability matrix is copied on line 10 and the depot is assigned the probability of zero for selection. Then, vectors are created to keep track of each vehicle's current vertex and current energy on line 13. An index set used to map an index to a specific weight is created on line 14.

After a new solution is created, the assignment process begins with the **while** loop on line 16 and repeats until the probability of selecting an edge is zero. First, line 17 selects the vehicle k with the earliest time. Then, on line 18, the weight vector $\vec{w}$ is assigned to be the row of P^* associated with vehicle k's current vertex position $\vec{v}[k]$. This vector weighs the probabilities of selecting a new vertex j from $\mathcal{Q}$ to add to $\mathcal{P}_k$. If the weights are zero, then the loop ends on line 20. If there are nonzero weights in $\vec{w}$, line 22 selects the next vertex j, using the `random_choice(A,B)` function which returns a randomly selected element of `A` given the weights of `B`. Line 23 calculates the time that vehicle k can arrive at j, line 24 calculates the energy needed for vehicle k to arrive at j, and line 25 calculates the energy needed to return to the depot. Line 27 extracts the time window for vertex j, where the earliest and latest times j can be visited, denoted as $\tau_{e,j}$ and $\tau_{l,j}$, respectively.

The next part of the algorithm ensures that the time and energy constraints are not violated. Line 28 ensures that the energy and time-window constraints are not violated. If the vehicle's consumed energy is less than or equal to $e_{\max}$ and the time constraints are

not violated, then j is added to vehicle k's route on line 29, where $A \oplus b$ signifies the act of adding or *appending* an element, b, to the end of a list, A. The vertex is marked as visited on line 30, and then set as the next vertex for vehicle k on line 31. The time vector is then updated on line 32, and the energy vector is updated on line 33. If the constraints are violated, the probability matrix is updated so that the probability of choosing j during the next round is zero.

The next part of the algorithm adds the newly-constructed solution to the population. After completing the **while** loop, the routes are converted to a chromosome on line 38 using Algorithm 2 and the chromosome is improved using Algorithm 6 and added to the population on line 39. After the population reaches p_s chromosomes, the population is returned on line 41.

Algorithm 7 $\mathcal{P}, E \leftarrow$ `create_population`$(E, T, p_s, n, m, e_{\max}, \vec{T}, \mathcal{W}, \bar{\alpha}, \bar{\beta})$

$\mathcal{P} \leftarrow \{\}$ // Initialize population
// Create a probability matrix
$E' \leftarrow [1/E[i][j]]\ \forall i, j \in \{0, \ldots, n\}$
$e \leftarrow \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} E'[i][j]$
$P \leftarrow E' \oslash e$ // Element-wise divide E' by e
for $l = 0, \ldots, p_s$ **do**
    // Start routes with depots
    $\mathcal{P}_k \leftarrow [0]\ \forall k \in \{0, \ldots, m\}$
    $\mathcal{R} \leftarrow \{\mathcal{P}_k\}\ \forall k \in \{0, \ldots, m\}$
    $P^* \leftarrow P$ // Copy P
    $P^*[i][0] \leftarrow 0\ \forall i \in \{0, \ldots, n-1\}$ // Zero out the depot
    // Keep track of vertices and energy
    $\vec{c} \in \mathbb{R}^m \leftarrow [0]$; $\vec{e} \in \mathbb{R}^m \leftarrow [0]$
    $\mathcal{Q} \leftarrow \{0, \ldots, n\}$ // Create an index set to map an index to a specific entry of $\vec{w}$
    // Start assignment process
    while `true` **do**
        $k = \text{argmin}(\vec{t})$
        $\vec{w} \in \mathbb{R}^n \leftarrow P^*[\vec{v}[k]][:]$ // Set weights based on current position
        if $\vec{w} = 0$ **do**
            break // End the process
        end if
        $j \leftarrow$ `random_choice`$(\mathcal{Q}, \vec{w})$ // Get a random index based on weights
        $t \leftarrow T[\sigma^k_{n_k-2}][j]$ // Track passage of time for vehicle k to travel to j
        $e \leftarrow E[\sigma^k_{n_k-2}][j]$ // Track energy usage for vehicle k to travel to j
        $e_{\text{return}} \leftarrow \vec{e}[k] + E[j][0]$ // Track energy usage for vehicle k to travel to j
        // Ensure that energy and time window constraints are not violated
        $(\tau_{e,j}, \tau_{l,j}) \leftarrow \mathcal{W}[j]$ // Extract the time window for vertex j
        if $\vec{e}[k] + e + e_{\text{return}} \leq e_{\max}$ **and** $\tau_{e,j} \leq \vec{t}[k] + t \leq \tau_{l,j}$ **do**
            $\mathcal{R}[k] \leftarrow \mathcal{R}[k] \oplus j$ // Add vertex j to vehicle k's route
            $P^*[i][j] \leftarrow 0, \forall i \in \{0, \ldots, n-1\}$, // Mark j as visited
            $\vec{v}[k] \leftarrow j$ // Set j as current vertex for vehicle k
            $\vec{t}[k] \leftarrow \vec{t}[k] + t$ // Track passage of time for vehicle k to travel to j
            $\vec{e}[k] \leftarrow \vec{e}[k] + e$ // Track energy usage for vehicle k to travel to j
        else do // Reject the move as constraints are not met
            $P^*[i][j] = 0$ // Eliminate probability of re-selection
        end if
    end while
    $\mathcal{C} \leftarrow$ `routes_to_chrom`$(\mathcal{R}, n)$ // Convert to chromosome via Algorithm 2
    $\mathcal{P} \oplus$ `two_opt`$(E, T, \mathcal{T}, \mathcal{W}, n, \mathcal{C}, e_{\max})$ // Add chromosome improved by Algorithm 6
end for
return $\mathcal{P}$

3.2.3 Operators

Now, given the auxiliary algorithms to the EHGA, along with having the ability to create a population of chromosomes, the focus shifts to the operations that work on the population and individuals within the population. Operators are crucial to genetic algorithms, as they help explore the solution space. The HGAs in [1] use operators for selection, crossover, mutation, and immigration, and some HGAs use operators for insertion, inversion, and swapping. The GA-ADP, the best-performing HGA in [1], uses selection, crossover, mutation, and immigration, and randomly selects the usage of insertion, inversion, and swapping. Given that the technique using all available operators provided the best results, the EHGA uses the operators of selection, crossover, mutation, insertion, inversion, swapping, and immigration. The selection and immigration operators operate on the whole population, the crossover operation operates on two chromosomes, while the remaining operators operate on a single chromosome. This subsection discusses the specific details of each operator, as well as some supporting algorithms for these operators, where appropriate.

Selection

The selection operator simulates natural selection in nature to ensure the "survival of the fittest," by keeping solutions with high utility for further improvement. Chromosomes are probabilistically chosen via the roulette-wheel method[2] to remain in the population given a fitness or *utility*, with the process being described in Algorithm 2 of [1]. The selection operator in this work differs from [1] by using the utility as a means of calculating the fitness of a chromosome instead of the inverse of the total distance traveled. The selection process imitates a roulette wheel and is described in Algorithm 8.

Algorithm 8 simulates natural selection using the roulette wheel method. It creates three vectors for the roulette-wheel process: the fitness vector $\vec{f}$, the probability vector *vecp*, and the cumulative probability vector $\vec{c}$. The fitness vector $\vec{f}$ has entries associated

[2]The roulette-wheel method partitions the interval [0,1] into smaller intervals whose range correlates with the probability of selection and selects an interval by generating a random number which falls into an interval, imitating a roulette wheel.

with the fitness of each chromosome. The probability vector *vecp* has entries associated with the probability of selecting the chromosome for each chromosome. The cumulative probability vector acts as the roulette wheel where chromosomes with higher probabilities correspond to larger wheel partitions.

Algorithm 8 starts by calculating the fitness of every chromosome in the population. It defines the fitness $\vec{f}$ to be a vector of all zeros in $\mathbb{R}^{|\mathcal{P}|}$ on line 2. The **for** loop which starts on line 3 iterates through each chromosome and assigns the chromosome's fitness to its corresponding index in the vector. The chromosome is extracted from the population on line 4, and Algorithm 3 is used to calculate the fitness of the chromosome, where the fitness of the chromosome is equivalent to the chromosome's utility. The fitness is calculated and recorded on line 5. Note that in implementing the algorithm, the utility of some solutions will not be finite. Therefore, a precaution to make infinite utilities zero may be required in implementation.

The next part of the algorithm is to calculate the probability of selection for each chromosome in the population as well as calculate the cumulative probability. The probability of selection vector, $\vec{p}$, and the cumulative probability vector, $\vec{c}$, are defined as vectors of all zeros on lines 8 and 9, respectively. The **for** loop on line 10 iterates through each chromosome, $\mathcal{C}$, in the population and constructs the partition of the roulette wheel that corresponds to $\mathcal{C}$, as well as an interval of numbers in $\mathbb{R}$ that correspond to the partition. Line 11 finds and records the size of the partition associated with a chromosome at index i by calculating the ratio of the chromosome's fitness to the cumulative fitness of all chromosomes. Thus, solutions which have higher utilities have greater probabilities of being selected. If the **for** loop is on its first iteration, then a zero is added to the cumulative probability vector on line 13; otherwise, the previous cumulative probability is added to the current probability and stored as the current cumulative probability on line 15. The cumulative probability assigns an interval of real numbers within $[0, 1]$ which corresponds to the partition associated with the chromosome.

The final part of the algorithm creates the new population. The new population variable is created on line 19. The **for** loop on line 20 recreates the population with selected chromosomes. First, a random number, r, is selected from the interval $[0, 1]$ on line 21 to simulate the ball landing on a partition of the roulette wheel. The **for** loop on line 22 finds the chromosome associated with the chosen partition by iterating through each chromosome's partition and checking if the ball has landed in that chromosome's partition. Thus, if r is between the previous cumulative probability and the current cumulative probability, then the chromosome associated with the current cumulative probability is added to the new population. After the **for** loops are finished, the population is returned on line 30.

Algorithm 8 $\mathcal{P} \leftarrow$ `selection`$(E, T, n, \mathcal{P})$

// Calculate fitness $\vec{f}$
$\vec{f} \in \mathbb{R}^{|\mathcal{P}|} \leftarrow [0]$
for $i = 0, \ldots, |\mathcal{P}|$ **do**
    $\mathcal{C} \leftarrow \mathcal{P}[i]$ // Extract chromosome from population
    // Calculate and record utility of each chromosome via Algorithm 3
    $\vec{f}[i] \leftarrow$ `calc_util`$(E, n, \mathcal{C}, \bar{\alpha}, \bar{\beta})$
end for
// Calculate probability $\vec{p}$ and cumulative probability $\vec{c}$
$\vec{p} \in \mathbb{R}^{|\mathcal{P}|} \leftarrow [0]$
$\vec{c} \in \mathbb{R}^{|\mathcal{P}|+1} \leftarrow [0]$
for $i = 0, \ldots, |\mathcal{P}|$ **do**
    $\vec{p}[i] \leftarrow \vec{f}[i] \oslash \sum_{j=0}^{|\mathcal{P}|} \vec{f}[j]$
    if $i = 0$ **do**
        $\vec{c}[0] \leftarrow 0$
    else do
        $\vec{c}[i] \leftarrow \vec{c}[i-1] + \vec{p}[i]$
    end if
end for
// Create new population
$\mathcal{P}^* \leftarrow \{\}$
for $i = 0, \ldots, |\mathcal{P}|$ **do**
    $r \leftarrow$`random`$([0, 1])$ // Choose a random number from $[0, 1]$
    for $j = 1, \ldots, |\mathcal{P}|$ **do**
        if $\vec{c}[j-1] \leq r \leq \vec{c}[j]$
            $\mathcal{P}^* \oplus \mathcal{P}[j]$
        else do
            $\mathcal{P}^* \oplus \mathcal{P}[$`argmax`$(\vec{f})]$ // Keep population size constant
        end if
    end for
end for
return $\mathcal{P}^*$

Crossover

After the selection process, the population undergoes more individualistic changes through operators such as crossover. The crossover function takes two chromosomes, or *parents*, and takes vertices from both parents to produce another chromosome, or *child*. One problem with randomly choosing vertices from parent chromosomes is that each parent could contain the same index. Thus, if not addressed, a child could contain the same

vertex twice, which is not allowed. Thus, an important subroutine of the crossover process is to find an unvisited, or *legitimate* vertex given a chromosome and one of its parents.

Algorithm 9 finds a legitimate vertex, and its ideas are encapsulated in a section of Algorithm 3 in [1]. In the ADVRP of [1], every parent is compatible, which means that the elements of every chromosome are the same; however, given that not every vertex must be visited, there exist cases where one parent has a vertex that the other parent does not. The algorithm's arguments consist of the parent chromosome $\mathcal{C}_p$, the child chromosome $\mathcal{C}$, and the current vertex of the child chromosome, c. The first part of Algorithm 9, which is not in Algorithm 3 in [1], evaluates the compatibility of the parent chromosome. Line 2 checks if the current vertex of the child is in the parent chromosome. If the current vertex is not in the parent chromosome, then the algorithm iterates through the parent chromosome using the **for** loop on line 3 to find a legitimate vertex, being a vertex that is not already in $\mathcal{C}$. During each iteration, line 4 extracts a vertex v from the parent chromosome. If v is not in the child chromosome, then v is a legitimate vertex and is returned on line 6.

The second part of the algorithm more closely mirrors Algorithm 3 in [1], given that the parents are compatible. The algorithm continues with line 10 by extracting the index of vertex c from the parent chromosome. Then, a list of vertices that are before c and a list of vertices that are after c are created on lines 11 and 12, respectively. The first **for** loop iterates through the vertices in the after-list $\mathcal{A}$. First, a vertex v is extracted from $\mathcal{A}$ on line 14. If the vertex is not already in the child chromosome, then the vertex is legitimate and is returned on line 16. The second **for** loop iterates through the vertices in the before-list $\mathcal{B}$. First, a vertex v is extracted from $\mathcal{B}$ on line 20. If the vertex is not already in the child chromosome $\mathcal{C}$, then the vertex is legitimate and is returned on line 22. If no legitimate vertex is found, then an error code -1 is returned from the algorithm on line 25, which is not in [1], since a legitimate vertex can always be found in the ADVRP.

Algorithm 9 $v \leftarrow$ `find_vertex`$(\mathcal{C}_p, \mathcal{C}, c)$

```
1:  // Check compatibility (Not in [1])
2:  if c not in C_p do
3:      for i = 0, ..., |C_p| do
4:          v ← C_p[i] // Extract a vertex from the parent
5:          if v not in C do
6:              return v // If v isn't already in C, then it is legitimate
7:          end if
8:      end for
9:  else do
10:     j ← C_p.index(c)
11:     B ← C_p[: j] // List of vertices before c
12:     A ← C_p[j + 1 :] // List of vertices after c
13:     for i = 0, ..., |A| do
14:         v ← A[i] // Extract a vertex from the parent
15:         if v not in C do
16:             return v // If v isn't already in C, then it is legitimate
17:         end if
18:     end for
19:     for i = 0, ..., |B| do
20:         v ← B[i] // Extract a vertex from the parent
21:         if v not in C do
22:             return v // If v isn't already in C, then it is legitimate
23:         end if
24:     end for
25:     return −1 // Return an error code (Not in [1])
26: end if
```

The crossover operator simulates crossover in nature by taking two parent chromosomes and creating a child chromosome that shares traits with both parents. The crossover operator, defined by Algorithm 10, takes in two parent chromosomes, the probability of performing a crossover operation, and necessary parameters for calculating utility and checking constraint violations.

The algorithm begins by initializing variables and determining whether or not a crossover operation will be performed. First, on line 1, the utilities of the parents are calculated using Algorithm 3. The phantom depot is then initialized, and line 3 selects a random number on the interval $[0, 1]$. If the random number is greater than the probability of doing a crossover operation, then no crossover operation happens, and the algorithm returns the parent with

the highest utility. This crossover operation differs from that of [1] because it checks both time and energy constraints and includes a protocol for when crossover does not occur.

If the crossover operation is to occur, then the next several lines find legitimate vertices from both parents and select the better vertex. The current vertex, the energy, and the time of the routes are initialized to be the depot, no energy, and the starting time of vehicle zero, respectively, on line 5. The child chromosome is created on line 6. The **for** loop which starts on line 7 adds vertices from the parent chromosomes to the child chromosome. The legitimate vertices are found using Algorithm 9 on line 9. If no legitimate vertices are found, then a number of checks are performed. If no constraints are violated in the child chromosome, then the child chromosome is returned. Otherwise, the parent with the higher utility is returned. In the case where legitimate vertices are found from both parents, then the utilities of adding the vertex are calculated on lines 16 and 17. Line 18 selects the vertex which results in a higher utility to be added to the chromosome.

The next part of the algorithm ensures that the addition of the selected vertex does not violate any constraints. The selected vertex is then checked by line 21 to see if it is a phantom depot. If the selected vertex is a phantom depot, then the route ends, the energy resets, and the time is changed to be the starting time of the next vehicle. The energy constraint is checked on line 24, and the time window constraints are checked on line 25. If the vertex passes both checks and the vertex is a phantom depot, then the phantom depot is added to the chromosome and the number of phantom depots is updated. If the number of phantom depots reaches the number of vehicles, then the chromosome is finished and the algorithm exits the loop on line 28. If the vertex is not a depot and passes all constraints, it is added to the chromosome on line 30, and the time, energy, and current vertex variables are updated on line 31. If the energy constraint is violated, then a phantom depot is added, and the number of phantom depots is updated. If the number of phantom depots reaches the number of vehicles, then the chromosome is finished and the algorithm exits the loop on line 36. After the loop is completed, the chromosome is returned on line 40.

Algorithm 10 $\mathcal{C} \leftarrow$ `crossover`$(E, T, \mathcal{C}_1, \mathcal{C}_2, p_c, n, \mathcal{T}, \mathcal{W}, e_{\max})$

$u_1 \leftarrow$`calc_util`$(E, n, \mathcal{C}_1, \bar{\alpha}, \bar{\beta})$; $u_2 \leftarrow$`calc_util`$(E, n, \mathcal{C}_2, \bar{\alpha}, \bar{\beta})$
$n_p \leftarrow 0$ // Initialize phantom depot counter
$r \leftarrow$`random`$([0, 1])$ // Choose a random number on the interval $[0, 1]$
if $r < p_c$ **do** // Perform crossover
    $c \leftarrow 0$; $e \leftarrow 0$; $t \leftarrow \mathcal{T}[0]$; // Track current vertex, energy, and time
    $\mathcal{C} \leftarrow [0]$ // Create chromosome
    for $i = 1, \ldots, n$ **do**
        // Find next legitimate vertex in both parents using Algorithm 9
        $v_1 \leftarrow$`find_vertex`$(\mathcal{C}_1, \mathcal{C}, j)$; $v_2 \leftarrow$`find_vertex`$(\mathcal{C}_2, \mathcal{C}, j)$
        if $v_1 = -1$ **or** $v_2 = -1$ **do**
            if not `constraints_violated`$(E, T, \mathcal{T}, \mathcal{W}, n, e_{\max}, \mathcal{C})$ **return** $\mathcal{C}$
            else if $u_1 > u_2$ **return** $\mathcal{C}_1$ // Return better parent
            else return $\mathcal{C}_2$
        end if
        // Calculate utilities of adding either v_1 or v_2 via Algorithm 3
        $u_1^* \leftarrow$ `calc_util`$(E, n, \mathcal{C} \oplus v_1, \bar{\alpha}, \bar{\beta})$
        $u_2^* \leftarrow$ `calc_util`$(E, n, \mathcal{C} \oplus v_2, \bar{\alpha}, \bar{\beta})$
        if $u_1^* > u_2^*$ **do** $v \leftarrow v_1$ // Select better vertex
        else do $v \leftarrow v_2$
        end if
        if $v \geq n$ **do** $v \leftarrow 0$; $e \leftarrow 0$; $t \leftarrow \mathcal{T}[n - v + 1]$ // Reset variables
        end if
        $(\tau_{e,v}, \tau_{l,v}) \leftarrow \mathcal{W}[v]$ // Extract time window
        if $e + E[c][v] \leq e_{\max}$ **do**
            if $\tau_{e,v} \leq t + T[c][v] \leq \tau_{l,v}$ **do** // Vertex passes both checks
                if $v = 0$ **and** 0 **in** $\mathcal{C}$ **do**
                    $\mathcal{C} \oplus n + n_p$; $n_p \leftarrow n_p + 1$ // Add and update a phantom depot
                    if $n_p = m$ **break**
                    end if
                else do $\mathcal{C} \oplus v$
                $e \leftarrow e + E[c][v]$; $t \leftarrow t + T[c][v]$; $c \leftarrow v$; // Update variables
                end if
            end if
        else do // Add a phantom depot
            $\mathcal{C} \oplus n + n_p$; $n_p \leftarrow n_p + 1$ // Add and update a phantom depot
            if $n_p = m$ **break**
            end if
        end if
    end for
    return $\mathcal{C}$
end if
if $u_1 > u_2$ **return** $\mathcal{C}_1$
end if
return $\mathcal{C}_2$

Mutation

Mutation is an important element of evolution, as genes sometimes mutate and cause variations within the population. In the EHGA, the mutation operator, like the crossover operator, has a prespecified probability of performing the operation. In general, the mutation probability is much lower than the crossover probability. The mutation operator randomly chooses two vertices in the chromosome and swaps them, given that neither of the vertices chosen represents a phantom depot. Algorithm 11 is a minimally-modified implementation of Algorithm 4 of [1], mainly where the constraint-violation check replaces the distance check of [1], and having a procedure for the case that mutation does not occur.

Algorithm 11 describes the procedure for the mutation operator. It starts by generating a random number on the interval $[0, 1]$ on line 1. If the mutation operation is to occur, two random indices are selected on line 4 using the `random_sample`$(\mathcal{A}, b)$ function. The `random_sample`$(\mathcal{A}, b)$ function takes in a list $\mathcal{A}$ and a number of elements b to choose from the list, and returns b randomly-selected elements of $\mathcal{A}$ as a smaller list. The **while** loop on line 5 ensures that neither randomly-selected index corresponds to a phantom depot by generating new indices on line 6 until neither index i nor index j correspond to phantom depots. Line 8 mutates the chromosome by swapping the chosen vertices. The mutated chromosome is checked for constraint violations on line 10. If the constraints are violated, then the vertices are replaced to their original order on line 11. The authors of [1] return the mutated chromosome; however, this implementation returns the original chromosome if mutation does not occur.

Algorithm 11 $\mathcal{C} \leftarrow$ `mutation`$(E, T, \mathcal{C}, p_m, n, \mathcal{T}, \mathcal{W}, e_{\max})$

```
1: r ←random([0, 1]) // Choose a random number from [0, 1]
2: if r ≤ p_m do
3:     // Swap two random vertices
4:     (i, j) ← random_sample([1, ..., |C| − 1], 2)
5:     while i > n or j > n do
6:         (i, j) ← random_sample([1, ..., |C|], 2)
7:     end while
8:     (C[i], C[j]) ← (C[j], C[i]) // Swap vertices at i and j
9:     // Check to see if new chromosome is valid
10:    if not constraints_violated (E, T, T, W, n, e_max, C) do
11:        (C[i], C[j]) ← (C[j], C[i]) // Undo swapping
12:    end if
13: end if
14: return C
```

Insertion

Given that different kinds of mutations occur in nature, more mutation operations, such as insertion, are presented and used in this work. The insertion operator is one of the local search techniques used to hybridize the GAs in [1], and by extension, hybridize the EHGA. The insertion operator removes a vertex from the chromosome and inserts the vertex into another place in the chromosome. If the insertion improves the solution, then the changes are kept. Otherwise, the changes are undone. The insertion operator is Algorithm 5 in [1] and is shown in this work, with a few minor changes, as Algorithm 12.

Algorithm 12 shows the procedure for the insertion operator. It begins with two nested **for** loops on lines 1 and 2, respectively. The **for** loops iteratively select a vertex to be removed from index i and inserted after an index j. Line 4 extracts the vertex v from index i of chromosome $\mathcal{C}$, line 5 creates a copy of the chromosome, and line 6 removes v from the chromosome's copy. The vertex v is then inserted after index j on line 7. The utilities of the original chromosome and the modified copy are calculated on lines 9 and 10, respectively. If the modified copy has a higher utility than the original and if the modified copy does not violate any constraints, then the modified copy replaces the original chromosome on line 12. The algorithm ends by returning the chromosome on line 16.

Algorithm 12 $\mathcal{C} \leftarrow$ `insertion`$(E, T, \mathcal{C}, n, \mathcal{T}, e_{\max})$

1: **for** $i = 1, \ldots, |\mathcal{C}| - 1$ **do**
2: **for** $j = i + 1, \ldots, |\mathcal{C}|$ **do**
3: // Insert i after j
4: $v \leftarrow \mathcal{C}[i]$
5: $\mathcal{C}^* \leftarrow \mathcal{C}$
6: $\mathcal{C}^* \ominus v$
7: $\mathcal{C}^* \leftarrow \mathcal{C}^*[: j] \oplus v \oplus \mathcal{C}^*[j :]$ // Insertion step
8: // Calculate utility via Algorithm 3
9: $u_{\text{old}} \leftarrow$ `calc_util`$(E, n, \mathcal{C}, \bar{\alpha}, \bar{\beta})$
10: $u_{\text{new}} \leftarrow$ `calc_util`$(E, n, \mathcal{C}^*, \bar{\alpha}, \bar{\beta})$
11: **if** $u_{\text{new}} > u_{\text{old}}$ **and not** `constraints_violated`$(E, T, \mathcal{T}, \mathcal{W}, n, e_{\max}, \mathcal{C}^*)$ **do**
12: $\mathcal{C} \leftarrow \mathcal{C}^*$
13: **end if**
14: **end for**
15: **end for**
16: **return** $\mathcal{C}$

Inversion

Another local search technique and mutation operator is the inversion operator. The inversion operator is used to further hybridize the GAs in [1] and EHGA. The inversion operator takes a series of adjacent vertices, called a *substring*, and inverts the order of the substring, given that no constraints are violated. The inversion operator shown in Algorithm 13 is Algorithm 6 from [1] with some minor modifications.

Algorithm 13 describes the procedure of the inversion operator. It starts with the nested **for** loops on lines 1 and 2 that iterate through the vertices in the chromosome to perform the inversion operation shown on line 4. The utilities of the original and inverted chromosomes are computed on lines 6 and 7, respectively. The two chromosomes are compared against each other on line 8, which also checks for constraint violations. If the inverted chromosome is better and does not violate any constraints, then the inverted chromosome replaces the original chromosome, and the iterative process continues. The algorithm returns the chromosome on line 13, terminating the algorithm.

Algorithm 13 $\mathcal{C} \leftarrow$ `inversion`$(E, T, \mathcal{C}, n, \mathcal{T}, \mathcal{W}, e_{\max})$

1: **for** $i = 1, \ldots, |\mathcal{C}| - 1$ **do**
2: **for** $j = i + 1, \ldots, |\mathcal{C}|$ **do**
3: // Reverse the order of vertices between vertices indexed at i and j
4: $\mathcal{C}^* \leftarrow \mathcal{C}[: i] \oplus \mathcal{C}[i : j][:: -1] \oplus \mathcal{C}[j :]$ // Inversion step
5: // Calculate utility via Algorithm 3
6: $u_{\text{old}} \leftarrow$ `calc_util`$(E, n, \mathcal{C}, \bar{\alpha}, \bar{\beta})$
7: $u_{\text{new}} \leftarrow$ `calc_util`$(E, n, \mathcal{C}^*, \bar{\alpha}, \bar{\beta})$
8: **if** $u_{\text{new}} > u_{\text{old}}$ **and not** `constraints_violated`$(E, T, \mathcal{T}, \mathcal{W}, n, e_{\max}, \mathcal{C}^*)$ **do**
9: $\mathcal{C} \leftarrow \mathcal{C}^*$
10: **end if**
11: **end for**
12: **end for**
13: **return** $\mathcal{C}$

Swapping

The final mutation operator and local search technique is the swap operator. The swap operator is used to hybridize the GAs in [1] and EHGA, and is presented as Algorithm 7 in [1]. The implementation of the swap operator in this work is shown as Algorithm 14 and is slightly modified from Algorithm 7 in [1].

Algorithm 14 shows the procedure of the swapping operator. It begins with the nested **for** loops on lines 1 and 2 that iteratively select the vertices at indices i and j to participate in the swapping process. Line 4 creates a copy of the chromosome, and the copy is changed to have the vertices at indices i and j swapped on line 5. The utilities of the original chromosome and the swapped chromosome are computed on lines 7 and 8, respectively, using Algorithm 3. If the swapped chromosome has a higher utility and does not violate any constraints, then the swapped chromosome replaces the original chromosome on line 10 and the process continues. After each possible combination of swapping vertices is complete, the chromosome is returned on line 14.

Algorithm 14 $\mathcal{C} \leftarrow$ `swap` $(E, T, \mathcal{C}, n, \mathcal{T}, \mathcal{W}, e_{\max})$

1: **for** $i = 1, \ldots, |\mathcal{C}| - 1$ **do**
2: **for** $j = i + 1, \ldots, |\mathcal{C}|$ **do**
3: // Swap vertices at indices i and j
4: $\mathcal{C}^* \leftarrow \mathcal{C}$
5: $\mathcal{C}^*[i] \leftarrow \mathcal{C}[j]; \mathcal{C}^*[j] \leftarrow \mathcal{C}[i]$ // Swapping step
6: // Calculate utility via Algorithm 3
7: $u_{\text{old}} \leftarrow$ `calc_util`$(E, n, \mathcal{C}, \bar{\alpha}, \bar{\beta})$
8: $u_{\text{new}} \leftarrow$ `calc_util`$(E, n, \mathcal{C}^*, \bar{\alpha}, \bar{\beta})$
9: **if** $u_{\text{new}} > u_{\text{old}}$ **and not** `constraints_violated`$(E, T, \mathcal{T}, \mathcal{W}, n, e_{\max}, \mathcal{C}^*)$ **do**
10: $\mathcal{C} \leftarrow \mathcal{C}^*$
11: **end if**
12: **end for**
13: **end for**
14: **return** $\mathcal{C}$

Immigration

To avoid getting trapped in local optima due to over-homogeneity of the solutions, the immigration method creates a population of new chromosomes to introduce diversity into the overall population. The rule in [1] is that given N generations to run the algorithm, if there has been no improvement within the last 10% of N generations, then 10% of the population is replaced by random chromosomes, which are further improved by the 2-opt local search algorithm. The immigration method, as mentioned in [1], does not explicitly state which chromosomes are replaced by immigrants, nor does there exist an algorithm for the immigration method in [1]. In this work, the immigration method is explained using Algorithm 15.

Algorithm 15 describes the immigration method. It starts by determining the number of immigrants n_i to be 10% of the population size on line 2. Then, a population of size n_i is created using Algorithm 7. Line 5 constructs a list of random indices of $\mathcal{P}$ using the `random_choices` function, where `random_choices`$(\mathcal{A}, b)$ randomly chooses b elements of $\mathcal{A}$ and returns those b elements in a list. The **for** loop on line 6 iterates through the population and replaces randomly-selected chromosomes with immigrants. Line 7 checks to see if an index, i, of $\mathcal{P}$ is one of the randomly-chosen indices. If the chromosome at i has been

chosen for replacement, then i is assigned to the corresponding replacement chromosome in the immigrant population, and the chromosome is replaced by the immigrant on line 8. After the replacement process, the new population is returned on line 11.

Algorithm 15 $\mathcal{P} \leftarrow$ `immigration`$(E, T, \mathcal{T}, \mathcal{W}, n, m, e_{\max}, p_s, \mathcal{P})$

1: // Create immigrant population
2: $n_i \leftarrow 0.1 \cdot p_s$ // Number of immigrants
3: $\mathcal{P}^* \leftarrow$`create_population`$(E, T, n_i, n, e_{\max}, \mathcal{T}, \mathcal{W}, \bar{\alpha}, \bar{\beta})$ // Using Algorithm 7
4: // Replace a random 10% of the original population
5: $\mathcal{S} \leftarrow$`random_choices`$([1, \ldots, |\mathcal{P}|], n_i)$
6: **for** $i = 1, \ldots p_s$ **do**
7: **if** i **in** $\mathcal{S}$ **do**
8: $\mathcal{P}[i] \leftarrow \mathcal{P}^*[\mathcal{S}.\texttt{index}(i)]$ // Replace chromosome with immigrant
9: **end if**
10: **end for**
11: **return** $\mathcal{P}$

3.2.4 The EATOP Hybrid Genetic Algorithm

Given the notation, supporting algorithms, and operators used in the EHGA, the EHGA can be fully defined. This section covers the EHGA and provides a discussion of the differences between the EHGA and the GA-ADP. The HGAs in [1] are shown as Algorithm 8 in [1]. Algorithm 16 is based on the GA-ADP and has a few minor variations from Algorithm 8 of [1].

The EHGA starts by initializing crucial variables. Algorithm 16 creates the initial population on line 1. Line 2 finds the best chromosome of the initial population, and its utility is calculated on line 3. The improvement counter, i_c, is initialized on line 4.

After the initialization step, the population undergoes several operations over several generations. The **for** loop on line 5 is the main loop of the EHGA, where each iteration represents a generation of the population. During each generation, the population goes through the selection process, then each chromosome in the population is replaced by a new chromosome created and modified using the available operators, the best solution is saved, and immigration is performed when necessary. The main loop starts with line 6, which performs the selection process using Algorithm 8. The **for** loop on line 7 operates

on each chromosome, thus creating the population for the next generation. Two random chromosomes are selected to be parents on lines 8 and 9, respectively, using a `random_choice` function which returns a random element from a list. After the parents are selected, a child chromosome is created using the crossover operation in Algorithm 10 on line 10. The child chromosome undergoes the mutation operator, Algorithm 11, on line 11. Then, following the method used in the GA-ADP, the EHGA randomly chooses between using the insertion (Algorithm 12), inversion (Algorithm 13), or swap (Algorithm 14) operators, and uses the chosen operator on the child chromosome on line 13. The child chromosome then replaces one of the chromosomes in the population on line 14.

After operations are performed on the population, the EHGA continues by selecting and recording the best chromosome, measuring improvement, and performing the immigration method when necessary. Line 16 finds the best chromosome using Algorithm 5 and calculates its utility on line 18 using Algorithm 3. Line 18 compares the new utility against the best utility. If the newly-calculated utility is better, lines 19 and 20 store the new solution, and line 21 resets the improvement counter. Otherwise, the improvement counter increases. If the improvement counter reaches 10% of all generations N, then the immigration process is applied on line 27 using Algorithm 15. When the EHGA has gone through each generation, the best chromosome is converted into a set of routes using Algorithm 1, on line 30, and the EHGA concludes by returning the set of routes on line 31.

Algorithm 16 $\mathcal{R} \leftarrow$`ehga`()

$\mathcal{P} \leftarrow$ `create_population`$(E, T, p_s, n, e_{\max}, \mathcal{T}, \mathcal{W}, \texttt{true}, \bar{\alpha}, \bar{\beta})$ // using Algorithm 7
$\mathcal{C}_{\text{best}} \leftarrow$ `get_best_chromosome`$(E, n, \mathcal{P})$ //using Algorithm 5
$u_{\text{best}} \leftarrow$ `calc_util`$(E, n, \mathcal{C}_{\text{best}}, \bar{\alpha}, \bar{\beta})$ // Calculate utility via Algorithm 3
$i_c \leftarrow 0$
for $i = 0, \ldots, N$
    $\mathcal{P} \leftarrow$ `selection`$(E, T, n, \mathcal{P})$ // Using Algorithm 8
    for $j = 0, \ldots, p_s$
        $\mathcal{C}_1 \leftarrow$ `random_choice`$(\mathcal{P})$
        $\mathcal{C}_2 \leftarrow$ `random_choice`$(\mathcal{P})$
        $\mathcal{C} \leftarrow$ `crossover`$(E, T, \mathcal{C}_1, \mathcal{C}_2, p_c, n, m, \mathcal{T}, \mathcal{W}, e_{\max})$ // Using Algorithm 10
        $\mathcal{C} \leftarrow$ `mutation`$(E, T, \mathcal{C}, p_m, n, \mathcal{T}, \mathcal{W}, e_{\max})$ // Using Algorithm 11
        // Randomly choose to use one of Algorithms 12, 13, or 14
        $\mathcal{C} \leftarrow$ `random_choice`$([\texttt{insertion}, \texttt{inversion}, \texttt{swap}])(E, T, \mathcal{C}, n, \mathcal{T}, \mathcal{W}, e_{\max})$
        $\mathcal{P}[j] \leftarrow \mathcal{C}$
    end for
    $\mathcal{C} \leftarrow$ `get_best_chromosome`$(E, n, \mathcal{P})$ // Using 5
    $u \leftarrow$ `calc_util`$(E, n, \mathcal{C}, \bar{\alpha}, \bar{\beta})$ // Calculate utility via Algorithm 3
    if $u > u_{\text{best}}$
        $\mathcal{C}_{\text{best}} \leftarrow \mathcal{C}$
        $u_{\text{best}} \leftarrow u$
        $i_c \leftarrow 0$
    else do
        $i_c \leftarrow i_c + 1$
    end if
    if $i_c = 0.1 \cdot N$ **do**
        // Perform immigration process
        $\mathcal{P} \leftarrow$ `immigration`$(E, T, n, m\mathcal{T}, \mathcal{W}, e_{\max}, p_s, \mathcal{P})$ // Using Algorithm 15
    end if
end for
$\mathcal{R} \leftarrow$ `chromosome_to_routes`$(\mathcal{C}, n)$ // Using Algorithm 1
return $\mathcal{R}$

Discussion of Differences Between This Work and [1]

Now, after giving the full definition of the EHGA, it is important to give a discussion regarding the main differences between the HGA approaches in [1] and this work. The main differences between the HGAs introduced in [1] and the EHGA in this work are those associated with adding more details to the algorithms and changing the objective and constraints. The EHGA focuses on maximizing utility instead of minimizing distance and adds time-window and vehicle-employment constraints. These changes are highlighted with the

utility-calculation algorithm (Algorithm 3) and the constraint-checking algorithm (Algorithm 4) which are not included in [1].This subsection highlights the differences between the EHGA and the HGA approaches in [1] using Table 3.3.

Table 3.3 highlights the differences between the EHGA and the HGA approaches in [1]. The first column of Table 3.3 shows each algorithm within the EHGA. The second column mentions the role of the algorithm in [1]. Most algorithms are either explicitly defined in [1] or their functionality is implied or *mentioned*. The third column of Table 3.3 provides a small description of each algorithm, and the fourth column mentions the changes made in this work. Given that some algorithms were only mentioned, the changes would be to formally describe or *formalize* the ideas as an algorithm.

Alg.	In [1]	Description	Changes
1	Mentioned	Convert Chromosomes to Routes	Formalized as an algorithm
2	N/A	Convert Routes to Chromosomes	N/A
3	N/A	Calculate Utility	N/A
4	N/A	Check Constraint Violations	N/A
5	Mentioned	Find Best Chromosome	Formalized as an algorithm
6	Mentioned	Modified 2-opt	Formalized as an algorithm
7	Algorithm 1	Population Creation	Complete change
8	Algorithm 2	Selection Process	Utility used for fitness
9	Mentioned	Find a Legitimate Vertex	Formalized as an algorithm
10	Algorithm 3	Crossover Operator	Added compatibility check
			Handled no-crossover case
			Handled no-legit-vertex case
			Changed obj. and constr.
11	Algorithm 4	Mutation Operator	Handled no-mutation case
			Changed constraints
12	Algorithm 5	Insertion Operator	Changed obj. and constr.
13	Algorithm 6	Inversion Operator	Changed obj. and constr.
14	Algorithm 7	Swapping Operator	Changed obj. and constr.
15	Mentioned	Immigration Method	Formalized as an algorithm
16	Algorithm 8	The Hybrid Genetic Algorithm	Made minor changes

Table 3.3: Highlighted differences between the algorithms of this work and [1]. For brevity, the terms *objective* and *constraint* are abbreviated as *obj.* and *constr.*, respectively.

3.3 Solving the EATOP Using an ACO Approach

Given the formulation of the problem in Section 3.1 and a solution technique in 3.2, the next step is to introduce a second solution technique for the EATOP. The solution technique presented in this section is an Ant Colony Optimization approach.

Ant colony optimization is a metaheuristic technique based on the food-foraging behavior of ant colonies in nature. Initially, ants move in random directions to explore the space around them. As ants forage for food, they leave pheromones behind them for other ants to follow. Once food is found, the ant that finds the food brings a sample back to the nest and increases the pheromone to the food. As more ants travel through the same path, the pheromone grows stronger and more ants make their way to the food to bring it back to the colony, creating a feedback loop. If one of the places where they find food runs out, then the ants go to other food sources, and the pheromone evaporates.

In the ACO algorithm, virtual ants explore the solution space for a good solution. Each edge of $\mathcal{G}$ has a pheromone value associated with it, which is initialized as the maximum value for each edge to make the ants explore the solution space. The first ant generates an initial solution, which is equivalent to finding food, and is used to compare subsequent solutions. The ants take turns generating solutions and each solution is compared against the best solution. If better solutions are found, the pheromones increase the likelihood that future ants will select the same, or similar, paths. Generally, ACO approaches repeat their process until a certain number of iterations is reached or the quality of the solution stagnates.

The ACO technique used to solve the EATOP is the main contribution of this section. The remainder of this section covers the notation of the ACO approach, the supporting algorithms for the ACO algorithms, and the ACO algorithms used to solve the EATOP.

3.3.1 Notation for the ACO Approach

The notation used in the Ant Colony implementation is found in Tables 3.4 and 3.5. Again, note that $\mathcal{P}_k$ is represented as a list. The values used for the parameters in these tables are found in Section 3.4, which covers implementation details.

Name	Description	Name	Description
Parameters			
m	The number of vehicles in the problem	N	The number of iterations to run the algorithm
n	The number of vertices in the problem	n_i	Local optimization is run every n_i iterations
$e_{\max}$	The maximum energy that any vehicle can use	α	The power on the pheromone
t_0	The starting time	β	The power on the element-wise, inverse cost
$\tau_{e,j}$	The earliest time a vertex j can be visited	$p_{\min}$	The minimum value for the pheromone
$\tau_{l,j}$	The latest time a vertex j can be visited	$p_{\max}$	The maximum value for the pheromone
n_a	The number of ants	ρ	The forgetting factor
Indices and Iterators			
i	An index generally depicting a vertex	k	An index depicting a vehicle
j	An index generally depicting a vertex	l	A generic iteration variable
Graph and Optimization Variables			
$\mathcal{G}$	The graph used in the problem	$\bar{\alpha}$	A scalar on the number of vertices in a solution
$\mathcal{E}$	The edge set of the graph.		
$\mathcal{V}$	The vertex set of the graph, representing the destinations	$\bar{\beta}$	A scalar on the total consumed energy
x_{ij}	A decision variable, which equals one if the edge (i, j) is used and is zero otherwise		
Operators			
$\oplus$	An operator where $A \oplus b$ appends element b to (the end of) a set or list A	$\otimes$	An operator where $A \otimes B$ performs element-wise multiplication
$\ominus$	An operator where $A \ominus b$ removes element b from a set or list A		

Table 3.4: Notation of the ACO Process

Name	Description	Name	Description
E	The energy matrix	P	The probability matrix
E_k	A copy of the energy matrix that only has the edges of $\mathcal{P}_k$ within it	P_k	A copy of P that only has the edges of $\mathcal{P}_k$ within it
E'	The element-wise inverse energy matrix	$\mathcal{Q}$	The index list of integers which correspond to a specific entry in a vector
E'_k	A copy of E' that only has the edges of $\mathcal{P}_k$ within it	$\vec{w}$	The weights vector
e	A variable which records the total energy consumed by all vehicles	H	The pheromone matrix
$e_{\min}$	The minimum energy usage found	Δ_H	This is a matrix which helps the update process of the pheromone matrix H
$\vec{e}$	The vector of "current energies"		
T	The time cost matrix	$\vec{v}$	The vector of "current vertices,"
T_k	A copy of the time matrix that only has the edges of $\mathcal{P}_k$ within it	$\mathcal{R}$	The set of routes assigned to the vehicles
$\mathcal{W}$	The list of time windows	$\mathcal{R}_{\text{best}}$	The best route found so far
$\vec{t}$	The vector of "current times"	$\mathcal{P}_k$	The list of vertices that vehicle k visits on vehicle k's path
u	The utility of a solution	n_k	The length of $\mathcal{P}_k$
$u_{\max}$	The utility of the best solution	σ_i^k	the ith element of k's path
n_d	The total number of visited destinations		
$d_{\max}$	The maximum number of visited destinations		

Table 3.5: Structures and Their Associated Variables in the ACO Process

3.3.2 Supporting Algorithms for the ACO Algorithms

The ACO approach uses supporting algorithms to create and analyze solutions. This section discusses the supporting algorithms used in the ACO implementation, including energy calculation, route generation, and pheromone update rules. The time matrix T, as well as the energy and element-wise inverse energy matrices, E and E', respectively, depend on the physical characteristics of specific problems.

Total Energy Calculation

An integral part of minimizing total energy expenditure is the ability to calculate the total energy usage across all vehicles in the fleet. Algorithm 17 outlines the procedure for calculating the total energy expenditure of all vehicles. Line 2 initializes e, the energy expenditure variable, as zero. The **for** loop on line 3 iterates through each of the vehicles.

Each iteration starts with extracting the vehicle's route $\mathcal{P}_k$ from the set of routes $\mathcal{R}$ on line 4. The second **for** loop on line 5 iterates through each vertex in vehicle k's route, where the energy used to go from the current vertex to the next vertex is added to e. After the energy expenditure of each vehicle is calculated, the total energy expenditure e is returned.

Algorithm 17 Energy Calculation for Route(s) $e \leftarrow$ `routes_energy`$(E, \mathcal{R})$

```
1: // Loop through each of the routes and find the total energy expenditure
2: e ← 0
3: for k = 0, ..., m do
4:     P_k ← R[k]
5:     for i = 0, ..., n_k − 1 do
6:         e ← e + E[σ_i^k][σ_{i+1}^k] // Cumulate energy of path P_k
7:     end for
8: end for
9: return e
```

Route Generation

The route-generating algorithm, Algorithm 18, assigns a list of vertices to each vehicle. The arguments of the algorithm consist of the initial conditions of the vehicles in terms of a vector of current vertices $\vec{v}$, which are the depots, a vector of current times $\vec{t}$, being the starting times, the probability matrix P, cost matrices for energy E and time T, and constraints such as time windows $\mathcal{W}$ and the energy constraint $e_{\max}$. It returns a set of routes $\mathcal{R}$.

The route-generating algorithm starts the routes of every vehicle at its depot and copies the probability matrix. To add vertices to the vehicle's route, the algorithm iteratively chooses a vehicle based on the earliest time and uses the row in the copy of the probability matrix corresponding to the vehicle's current location. The next vertex is chosen by that probability, and the algorithm calculates the total time and energy needed to go to that vertex. Given that adding the vertex to the vehicle's route may cause the constraints to be violated, the constraints are checked for violations, and the vertex is only added to the route if no constraints are violated. The probability matrix is updated to ensure that the algorithm does not select the same vertex in the future, and the iterative process ends when

all probabilities are zero.

The first part of Algorithm 18 is the initialization process. The algorithm starts the route for each vehicle k, $\mathcal{P}_k$, with the depot of vehicle k. Each $\mathcal{P}_k$ is added to the overall set of routes $\mathcal{R}$ on line 2. Lines 3-4 of the algorithm create a copy of the probability matrix P called P^* and ensure that the depots cannot be reassigned. Line 5 initializes a vector used to track the energy usage of the fleet, indexed by vehicle. Line 6 creates an index set, $\mathcal{Q}$, used to map an index to an element of the probability weights vector $\vec{w}$.

The next part of Algorithm 18 is the assignment process. The assignment process begins with the **while** loop on line 7 and the process repeats until the probability of selecting any edge is zero. First, line 8 selects the vehicle k with the earliest time. Then, on line 9, the weight vector $\vec{w}$ is assigned to be the row of P^* associated with vehicle k's current vertex position $\vec{v}[k]$. This vector weighs the probabilities of selecting a new vertex j from $\mathcal{Q}$ to add to $\mathcal{P}_k$. If all the weights are zero, the assignment process ends on line 11. Otherwise, the next vertex j is selected on line 13, using the `random_choice(A,B)` function which returns a randomly selected element of `A` given the weights of `B`. Line 14 calculates the time that vehicle k can arrive at j, line 15 calculates the energy needed for vehicle k to travel to j, and line 16 calculates the energy for the return trip. Line 18 extracts the time window for vertex j, where the earliest and latest times j can be visited, denoted as $\tau_{e,j}$ and $\tau_{l,j}$, respectively.

After a possible assignment has been identified, the next step in the algorithm is to check the validity of the assignment and to make the assignment if it is valid. Line 19 makes comparisons with respect to both energy and time, thereby ensuring that the energy and time window constraints are not violated. If the vehicle's consumed energy is less than or equal to $e_{\max}$ and the time constraints are not violated, then j is added to vehicle k's route on line 20, where $A \oplus b$ signifies the act of appending an element, b, to the end of a list, A. The vertex is marked as visited on line 21, and then set as the next vertex for vehicle k on line 22. The time vector is then updated on line 23, and the energy vector is updated on line 24. If the constraints are violated, then the probability matrix is updated so that the

probability of choosing j on the next round is zero. After completing the **while** loop, the return to the depot is added to each route, and then $\mathcal{R}$ is returned on line 30 as the route assignments for each vehicle.

Algorithm 18 A Route Generator Based on Probabilities
$\mathcal{R} \leftarrow$ `generate_routes`$(\vec{d}, \vec{t}, P, E, T, \mathcal{W}, e_{\max})$

1: $\mathcal{P}_k \leftarrow \{\mathcal{I}[k]\}, \forall k \in \{0, \ldots, m-1\}$ // Start routes with depots
2: $\mathcal{R} \leftarrow \{\mathcal{P}_k\}, \forall k \in \{0, \ldots, m-1\}$
3: $P^* \leftarrow P$ // Make a copy of P
4: $P^*[i][:] \leftarrow 0$ // Zero out the depots
5: $\vec{e} \in \mathbb{R}^{|\mathcal{I}|} \leftarrow [0]$ // Make a vector to track energy usage
6: $\mathcal{Q} \leftarrow \{0, \ldots, n\}$ // Create an index set to map an index to a specific entry of $\vec{w}$
7: **while** `true` **do**
8: $k = \text{argmin}(\vec{t})$
9: $\vec{w} \in \mathbb{R}^n \leftarrow P^*[\vec{v}[k]][:]$ // Set weights based on current position
10: **if** $\vec{w} = 0$ **do**
11: **break** // End the process
12: **end if**
13: $j \leftarrow$ `random_choice`$(\mathcal{Q}, \vec{w})$ // Get a random index based on weights
14: $t \leftarrow T[\sigma^k_{n_k-2}][j]$ // Track passage of time for vehicle k to travel to j
15: $e \leftarrow E[\sigma^k_{n_k-2}][j]$ // Track energy usage for vehicle k to travel to j
16: $e_{\text{return}} \leftarrow E[j][\mathcal{P}_k[0]]$ // Calculate return trip
17: // Ensure that energy and time window constraints are not violated
18: $(\tau_{e,j}, \tau_{l,j}) \leftarrow \mathcal{W}[j]$ // Extract the time window for vertex j
19: **if** $e \leq e_{\max}$ **and** $\tau_{e,j} \leq t \leq \tau_{l,j}$ **do**
20: $\mathcal{R}[k] \leftarrow \mathcal{R}[k] \oplus j$ // Add vertex j to vehicle k's route
21: $P^*[:][j] \leftarrow 0$ // Mark j as visited
22: $\vec{v}[k] \leftarrow j$ // Set j as current vertex for vehicle k
23: $\vec{t}[k] \leftarrow \vec{t}[k] + t$ // Track passage of time for vehicle k to travel to j
24: $\vec{e}[k] \leftarrow \vec{e}[k] + e$ // Track energy usage for vehicle k to travel to j
25: **else do** // Reject the move as constraints are not met
26: $P^*[i][j] = 0$ // Eliminate probability of re-selection
27: **end if**
28: **end while**
29: $\mathcal{P}_k \oplus \mathcal{P}_k[0] \quad \forall \mathcal{P}_k \in \mathcal{R}$ // Return to depot
30: **return** $\mathcal{R}$

Pheromone Update Rules

Another vital part of ACO is the pheromone update. The pheromone matrix, H, is updated with each iteration l of the ACO process. The new pheromone matrix, H_{l+1} is calculated using the current pheromone matrix H_l and the current pheromone update matrix Δ_l, shown by the recurrence relation

$$H_{l+1} = (1 - \rho)H_l + \Delta_l, \tag{3.4}$$

where ρ is the forgetting factor. Every element of the first pheromone matrix H_0 is set to have the highest pheromone level, denoted as $p_{\max}$. The pheromone update matrix is generated using Algorithm 19. The arguments include the utility of a solution, u, the pheromone update matrix, Δ, and the generated routes of a solution, $\mathcal{R}$. It returns the updated Δ. The **for** loop on line 2 iterates through each of the vehicles. The loop starts by extracting the vehicle's route $\mathcal{P}_k$ on line 3. The second **for** loop updates the pheromone for each edge on the path. It creates the variables i and j for starting and ending cities, respectively, and calculates the value used for the pheromone update, based on the overall utility of the solution u. Note that only a portion of Δ gets overwritten on line 7.

Algorithm 19 Delta Pheromone $\Delta \leftarrow$ `update_del_pheromone`$(u, \Delta, \mathcal{R})$

```
1:  // Get Δ given a vehicle's route
2:  for k = 0, ..., m do // Loop through vehicles
3:      P_k ← R[k]
4:      for l = 0, ..., n_k − 1 do
5:          i ← P_k[l] // Starting destination
6:          j ← P_k[l + 1] // Ending destination
7:          Δ[i][j] ← Δ[i][j] + u
8:      end for
9:  end for
10: return Δ
```

3.3.3 Ant Colony Optimization Algorithms

Given the notation and supporting algorithms for the ACO approach, the ACO algorithms can be fully defined. This section covers the ACO algorithms used to solve the EATOP. The first algorithm is a single-vehicle version of ACO which could be used alone or as part of a strategy to locally improve each route during the multi-agent optimization. The second algorithm details the improvement process, and the final algorithm is the multi-vehicle version of ACO.

The Single-Vehicle ACO Algorithm

Algorithm 20 runs a single-vehicle instance of the problem and creates a route for the vehicle. The algorithm starts by condensing the cost matrices, the pheromone matrix, and the probability matrix to only use the vertices found in the route $\mathcal{P}_k$. The condensing process also changes the indexing between the original problem and the single-vehicle version of the problem, where $\mathcal{P}_k$ in the original problem is equivalent to $[0, \dots |\mathcal{P}_k]$ in the single-vehicle version of the problem. The single-vehicle version of the ACO algorithm uses the current route as the initial solution and iterates through the ant colony process until there is no improvement after a number of iterations directly proportional to the length of the route.

The first part of the algorithm creates single-vehicle versions of the matrices used to solve the problem. After extracting k from the original route, line 3 initializes single-vehicle versions of the energy, element-wise, inverse energy, time, and pheromone matrices. After initialization, the matrices are populated by running the two nested **for** loops which start on lines 4 and 5, respectively. Alternatively, the pheromone matrix can start with every element as the maximum allowed pheromone value to explore the solution space further. In testing, it was observed that keeping the pheromones generally enhanced the solution, and thus is used in this implementation.

The next section of the algorithm initializes the storage variables in preparation to run the ACO process. Line 9 uses the current route as an initial solution; however, Algorithm 18 may be used to generate a route from scratch to explore the solution space further. On the other hand, starting with the already-constructed route has been shown in testing to

enhance the solution. Line 10 calculates the total energy usage of the current route. The number of destinations visited by the vehicle (the full length of the route minus the two depot instances) is recorded on line 11.

The ACO process is contained in the **while** loop that starts on line 14. First, the probability matrix is updated according to the pheromones on line 15. It uses the probability update rule below, where α and β are scalars, the notation $[A]^b$ denotes each element of A, a_{ij}, becomes $(a_{ij})^b$, and the product $A \otimes B$ denotes element-wise multiplication of matrices A and B:

$$P = [H]^{\alpha} \otimes [E']^{\beta}. \tag{3.5}$$

The pheromone update matrix is then initialized as zeros on line 16. On line 17, the ants begin the process of creating and enhancing new routes. First, a new route is generated via Algorithm 18 on line 18, then the total energy of the route is calculated using Algorithm 17 on line 19. Line 20 records the number of destinations visited. The utility of the new solution is calculated on line 21. Then, a check is run to see if the solution generated on line 18 has a higher utility than the best solution found so far. If the utility is higher, then the solution is recorded as the best solution on line 23. The pheromone update matrix is updated on line 25, and the for loop ends.

The next section of the algorithm details the pheromone update after each ant is run. The best utility found is scaled by $p_{\max}$ on line 27 and the pheromone update matrix is once again updated based on the best ant on line 28. The pheromones are updated and saturated on lines 29-30. Line 31 checks if the algorithm is still improving the solution and records the utility of the new solution if it is better than the former best solution. If the solution stagnates after a number of iterations that is directly proportional to the length of the route, the process stops.

The algorithm ends by returning the best route found on line 39, where the function $\texttt{sort}(\mathcal{A}, \mathcal{B})$ rearranges list $\mathcal{A}$ given list $\mathcal{B}$. For an example, let $\mathcal{A} = [0, 56, 23, 19, 7]$ and $\mathcal{B} = [0, 2, 1, 4, 3]$. The new list $\mathcal{A}^* = \texttt{sort}(\mathcal{A}, \mathcal{B})$ would be $\mathcal{A}^* = [0, 23, 56, 7, 19]$. This is used to convert the indices used in the smaller problem to those in the original problem.

Algorithm 20 $\mathcal{P}_k \leftarrow$ single_aco$(\mathcal{P}_k, E, E', T, H)$

// Initialize and populate single-vehicle versions of matrices
$k \leftarrow \mathcal{P}_k[0]$
$E_k, E'_k, T_k, H_k \in \mathbb{R}^{n_k \times n_k} \leftarrow [0]$; $P_k \in \mathbb{R}^{n_k \times n_k} \leftarrow [1]$
for $i = 0, \ldots, n_k$ **do**
    for $j = 0, \ldots, n_k$ **do**
        $E_k[i][j] \leftarrow E[\sigma_i^k][\sigma_j^k]$; $E'_k[i][j] \leftarrow E'[\sigma_i^k][\sigma_j^k]$;
        $T_k[i][j] \leftarrow T[\sigma_i^k][\sigma_j^k]$; $H_k[i][j] \leftarrow H[\sigma_i^k][\sigma_j^k]$
    end
end
$\mathcal{P}_{\text{best}} \leftarrow [0, \ldots, |\mathcal{P}_k|]$ // Use current route as initial solution
$e_{\min} \leftarrow$ `routes_energy`$(E_k, \{\mathcal{P}_{\text{best}}\})$ // Get the total energy usage via Algorithm 17
$d_{\max} \leftarrow |\mathcal{P}_{\text{best}}| - 2$ // Maximum number of destinations visited
$u_{\max} \leftarrow \bar{\alpha} d_{\max} - \bar{\beta} e_{\min}$ // Calculate initial max utility
$l \leftarrow 0$
while `true` **do** // Run ACO process
    $P_k \leftarrow [H_k]^{\alpha} [E'_k]^{\beta}$ // Update the probabilities via Equation 3.5
    $\Delta \in \mathbb{R}^{n_k \times n_k} \leftarrow [0]$ // Initialize pheromone update matrix as zeros
    for $j = 0, \ldots, n_a$ **do** // Run the ants
        // Generate solution via Algorithm 18
        $\{\mathcal{P}_k^*\} \leftarrow$ `generate_routes`$(\{\mathcal{P}_k[0]\}, \{\mathcal{T}[\mathcal{P}_k[0]]\}, P_k, E_k, T_k, \mathcal{W}, e_{\max})$
        $e \leftarrow$ `routes_energy`$(E_k, \{\mathcal{P}_k^*\}, 1)$ // Get total energy usage via Algorithm 17
        $n_d \leftarrow |\mathcal{P}_k^*| - 2$ // Record number of destinations visited
        $u \leftarrow \bar{\alpha} n_d - \bar{\beta} e$ // Calculate the score u
        if $u > \bar{\alpha} d_{\max} - \bar{\beta} e_{\min}$ **do** // If the solution is better, record it
            $e_{\min} \leftarrow e$; $d_{\max} \leftarrow n_d$; $\mathcal{P}_{\text{best}} \leftarrow \mathcal{P}_k^*$
        end if
        // Calculate the pheromone update given the ant's route via Algorithm 19
        $\Delta \leftarrow$ `update_del_pheromone`$(u, \Delta, \mathcal{P}_k^*)$
    end for
    $u^* \leftarrow p_{\max}(\bar{\alpha} d_{\max} - \bar{\beta} e_{\min})$ // Best utility found scaled by $p_{\max}$
    // Calculate pheromone update given best route via Algorithm 19
    $\Delta \leftarrow$ `update_del_pheromone`$(u^*, \Delta, \mathcal{P}_{\text{best}})$
    $H_k \leftarrow (1 - \rho) H_k + \Delta$ // Update the pheromones with Equation 3.4
    // Saturate the pheromones to minimum and maximum allowed values
    $H_k \leftarrow \min(H, p_{\max})$; $H_k \leftarrow \max(H_k, p_{\min})$
    if $\bar{\alpha} d_{\max} - \bar{\beta} e_{\min} > u_{\max}$ **do** // If best ant solution is better than $u_{\max}$
        $u_{\max} \leftarrow \bar{\alpha} d_{\max} - \bar{\beta} e_{\min}$; // Record better solution
        $l \leftarrow 0$
    else if $l > |\mathcal{P}_k|$ **do**
        `break`
    end if
    $l \leftarrow l + 1$
end for
return `sort`$(\mathcal{P}_k, \mathcal{P}_{best})$

Although Algorithm 20 could be used alone in a single instance of the EATOP, it can also be used as part of a strategy to improve the solutions for each individual agent during the multi-agent optimization process. Algorithm 21 improves a solution to the EATOP by running the single-vehicle instance of the problem (Algorithm 20). Lines 1-3 initialize the variables needed to evaluate the utility of the improved solution. The algorithm iterates through each of the routes of the solution, starting on line 4. On line 5, Algorithm 20 is used to make improvements to the route, and the improved route is recorded on line 6. Lines 7-8 record the energy usage of all the routes and destinations visited, respectively. After each route has been improved, the overall utility is calculated on line 10. The improved routes and utility are returned on line 11.

Algorithm 21 $\mathcal{R}_{\text{best}}, u \leftarrow$ improve_solution$(\mathcal{R}, E, E', T, H)$

1: $\mathcal{R}_{\text{best}} \leftarrow \{\}$ // Initialize improved routes
2: $e_{\min} \leftarrow 0$ // Energy usage of all routes
3: $d_{\max} \leftarrow 0$ // Maximum destinations visited
4: **for** $k = 0, \ldots, |\mathcal{R}|$ **do** // Loop through routes
5: $\quad \mathcal{P}_k, e \leftarrow$ single_aco $(\mathcal{R}[k], E, E', T, H)$ // Run single-vehicle ACO (Algorithm 20)
6: $\quad \mathcal{R}_{\text{best}} \oplus \mathcal{P}_k$ // Add improved route to $\mathcal{R}_{\text{best}}$
7: $\quad e_{\min} \leftarrow e_{\min} + e$ // Add improved energy to total energy usage
8: $\quad d_{\max} \leftarrow d_{\max} + |\mathcal{P}_k| - 2$ // Count destinations visited
9: **end**
10: $u \leftarrow \bar{\alpha} n_d - \bar{\beta} e$ // Calculate the score u
11: **return** R_{best}, u

The Multi-Vehicle ACO Algorithm

The multi-vehicle ACO algorithm solves the EATOP by creating and improving routes through a simulated ant colony. The arguments of Algorithm 22 are the costs, initial conditions, and constraints of the problem. The algorithm returns the routes generated for each vehicle.

The first part of Algorithm 22 is the initialization process. The algorithm starts by initializing the probability matrix as all ones and the pheromone as $p_{\max}$ at every element of the matrix. The best routes variable, $\mathcal{R}_{\text{best}}$, is initialized on line 4 using the route generator in Algorithm 18 and stores the set of routes with the highest utility. The minimum energy

variable $e_{\min}$ is initialized on line 5, being calculated using Algorithm 17. The maximum number of total destinations visited, $d_{\max}$, is calculated by adding the destinations visited by each agent on line 6. This calculation is done by adding the destinations visited by the vehicles going on the routes in $\mathcal{R}_{\text{best}}$ and subtracting the number of vehicles (the number of depots) from the sum. The utility of $\mathcal{R}_{\text{best}}$ is calculated and stored on line 7. After the storage variables are initialized, the iterative process begins.

The iterative part of the algorithm starts on line 8 and repeats for N iterations. Line 9 calculates a new probability matrix, P, using the probability update rule in Equation 3.5. The pheromone update matrix $\Delta \in \mathbb{R}^{n \times n}$ is initialized to zeros on line 10. The **for** loop that starts on line 11 is where the virtual ants explore the solution space by generating routes and updating the pheromones. Line 12 generates the routes of every vehicle, denoted as $\mathcal{R}$, using Algorithm 18. The total consumed energy e is then calculated using Algorithm 17 on line 13. The number of destinations visited, n_d, is calculated by adding the number of destinations in each path $\mathcal{P}_k \in \mathcal{R}$, again subtracting the number of depots. Line 15 calculates the utility of the solution, which is calculated by the number of destinations visited, multiplied by a scaling factor $\bar{\alpha}$, minus the total energy expenditure, with both multiplied by its respective scaling factor, $\bar{\beta}$. Line 16 checks the improvement of the utility of the new solution versus the best utility found. If the solution has improved, e, n_d, and $\mathcal{R}$ are stored. The **for** loop ends on line 19 with updating Δ using Algorithm 19.

The next section of the algorithm includes the improvement process and the pheromone update. The utility of the best solution generated by the ants is calculated on line 21. Line 23 runs Algorithm 21 every n_i iterations to improve the best solution. The utility of the improved solution is compared against the best solution on line 24. If the improved solution is better, it is recorded as the best solution on line 25.Algorithm 19 is run again on line 28, using the best utility to update Δ based on the best performing ant. The pheromone matrix H is updated on line 29, using the rule in Equation 3.4. The pheromone matrix is then saturated on line 30 and the cycle repeats until N iterations have been made. The algorithm ends by returning the best overall solution on line 32.

Algorithm 22 $\mathcal{R}_{\text{best}} \leftarrow \text{aco}(\mathcal{I}, \mathcal{T}, E, E', T, \mathcal{W}, e_{\max}, m)$

// Initialize ACO variables
$P \in \mathbb{R}^{n \times n} \leftarrow [1]$ // Initialize probability matrix P
$H \in \mathbb{R}^{n \times n} \leftarrow [p_{\max}]$ // Initialize pheromone matrix H
// Create initial solution using Algorithm 18
$\mathcal{R}_{\text{best}} \leftarrow \texttt{generate\_routes}(\mathcal{I}, \mathcal{T}, P, E', T, \mathcal{W}, e_{\max})$
$e_{\min} \leftarrow \texttt{routes\_energy}(E, \mathcal{R}_{\text{best}})$ // Get the total energy usage via Algorithm 17
$d_{\max} \leftarrow -2m + \sum_{k=0}^{m} n_k$ // Maximum number of destinations visited
$u_{\max} \leftarrow \bar{\alpha} d_{\max} - \bar{\beta} e_{\min}$
for $i = 0, \ldots, N$ **do**
    $P \leftarrow [H]^{\alpha} \otimes [E']^{\beta}$ // Update the probabilities via Equation 3.5
    $\Delta \in \mathbb{R}^{n \times n} \leftarrow [0]$ // Initialize pheromone update matrix as zeros
    for $j = 0, \ldots, n_a$ **do** // Run the ants
        // Create new routes using Algorithm 18
        $\mathcal{R} \leftarrow \texttt{generate\_routes}(\mathcal{I}, \mathcal{T}, P, E', T, \mathcal{W}, e_{\max})$
        $e \leftarrow \texttt{routes\_energy}(E, \mathcal{R})$ // Get the total energy usage via Algorithm 17
        $n_d \leftarrow -2m \sum_{k=0}^{m} n_k$ // Record number of destinations visited
        $u \leftarrow \bar{\alpha} n_d - \bar{\beta} e$ // Calculate the score u
        if $u > u_{\max}$ **do** // If the solution is better, record it
            $e_{\min} \leftarrow e$; $d_{\max} \leftarrow n_d$; $v_{\min} \leftarrow n_v$; $\mathcal{R}_{\text{best}} \leftarrow \mathcal{R}$; $u_{\max} \leftarrow u$
        end if
        // Calculate the pheromone update given the ant's route via Algorithm 19
        $\Delta \leftarrow \texttt{update\_del\_pheromone}(u, \Delta, \mathcal{R})$
    end for
    $u_{\max} \leftarrow \bar{\alpha} d_{\max} - \bar{\beta} e_{\min}$ // Best utility found
    if i **mod** n_i **is** 0 **do**
        // Improve solution with Algorithm 21
        $\mathcal{R}, u \leftarrow \texttt{improve\_solution}(\mathcal{R}_{\text{best}}, E, E', T)$
        if $u > u_{\max}$ **do**
            $u_{\max} \leftarrow u; \mathcal{R}_{\text{best}} \leftarrow \mathcal{R}$
        end if
    end if
    // Calculate the pheromone update given the ant's route via Algorithm 19
    $\Delta \leftarrow \texttt{update\_del\_pheromone}(u_{\max}, p_{\max}, \Delta, \mathcal{R}_{\text{best}})$
    $H \leftarrow (1 - \rho)H + \Delta$ // Calculate Pheromones with Equation 3.4
    // Saturate the pheromones to minimum and maximum allowed values
    $H \leftarrow \min(H, p_{\max}); H \leftarrow \max(H, p_{\min})$
end for
return $\mathcal{R}_{\text{best}}$

3.4 Implementation details

The EHGA in section 3.2 and the ACO algorithm in section 3.3 were both implemented in Python and run on an Intel(R) Core(TM) i7-10700 CPU with 32 GB of RAM for comparison. For clarity and replication purposes, the implementation details for the EHGA and ACO approaches are covered in this section. This section highlights the values used in the objective function and how they were determined and defines the parameters used in both implementations. The parameters and their values are described in Table 3.6.

To tune the parameters of the objective function, namely $\bar{\alpha}$ and $\bar{\beta}$, the Ray library with the submodule Tune, which is used to tune the parameters of neural networks, was used. The parameter tuning was divided into two phases: a broad-search phase and a narrow-search phase. The broad-search phase used 100 samples and ran for 10 rounds. The broad search tested random multiples of 5 between 0 and 2000 for each parameter to find desirable results. For each sample, the ACO algorithm took in different parameters and produced a solution and an overall maximized score. The best sample for each round was stored for the narrow-search phase. The narrow-search phase used the saved values to refine the process based on the first phase, converging on the values of $\bar{\alpha}$ and $\bar{\beta}$ that yielded desirable results. The narrow-search phase also used 100 samples and was run for 10 rounds. The parameters were chosen to be a random multiple of 5 between the minimum and maximum values of the parameters of the best samples from phase 1. It was determined that in general, a larger $\bar{\alpha}$ and a middle-ranged $\bar{\beta}$ yielded desirable results. Thus, the parameters for $\bar{\alpha}$ and $\bar{\beta}$ were selected to be 2000 and 1000, respectively.

Approach	Parameter	Description	Value
EHGA	p_s	Population size	50
	p_c	Probability of crossover	0.8
	p_m	Probability of mutation	0.1
ACO	n_a	The number of ants	5
	$p_{\min}$	Minimum pheromone value	0.01
	$p_{\max}$	Maximum pheromone value	100
	ρ	The forgetting factor	0.1
	n_i	The smaller problem is run every n_i iterations	10
	α	The power on the pheromone	1
	β	The power on E'	5
Both	m	The number of vehicles	3
	n	The number of vertices	51
	N	The number of generations/iterations	200
	$\mathcal{T}$	The starting times	varies
	$e_{\max}$	The energy constraint	varies

Table 3.6: Parameters and their Values

3.5 Method Comparison Against a Hybrid Genetic Algorithm

To compare the ACO implementation against the hybrid genetic algorithm, a series of simple test scenarios were created to test the behavior of the algorithms under time and energy constraints. The first testing scenario, which acts as a control group, gives the vehicle fleet adequate time and energy to visit all the vertices. The second testing scenario gives the vehicle fleet adequate time to visit all vertices but does not give the fleet adequate energy to visit all vertices. This second test is used to show the behavior of each algorithm given an energy constraint. The third testing scenario gives the vehicles adequate energy to visit every vertex, but does not give the fleet enough time to visit every vertex. This third test is used to show the behavior of each algorithm given a time constraint. The fourth

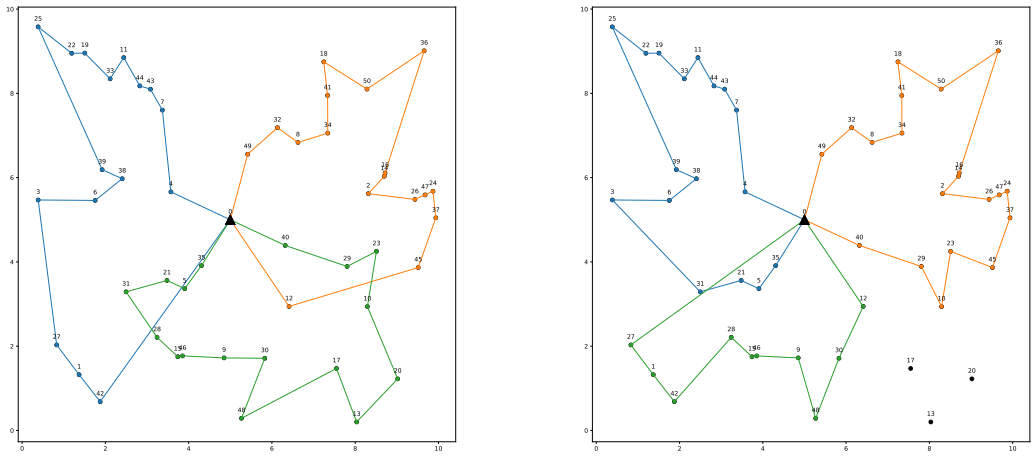
testing scenario constrains both the time and the energy consumption. This final test is used to show the behavior of each algorithm given heavy constraints.

For better replicability, the details of the tests are presented. Three different scenarios containing 50 destination points were randomly generated on a 10-mile square grid with the depot in the center of the grid. Thus, for each of the three scenarios, there were 51 total vertices in the problem. Each scenario was run 20 times for each test. The velocity to travel between these points was 25 miles per hour with a dwell time of 5 minutes per vertex. Using information for the Ford E-transit [67], a somewhat conservative estimate of the miles per kWh ratio is 1.85. The sufficient time window was three hours long, which was verified in simulation. The sufficient energy to visit all the vertices was the full battery capacity. The insufficient time was a 1.5 hour time window, and the insufficient energy to visit all vertices was 10% of the battery capacity. For simplicity in showing time windows, the vertices were placed within the same time window.

The remainder of this section showcases the results from each of the four tests and provides some statistical insight about the overall behaviors of both algorithms. The tests are presented in the following order: the unconstrained test, the energy-constrained test, the time-constrained test, and the time-and-energy-constrained test. After the presentation of each test, a series of histograms containing data for each test is presented and discussed.

3.5.1 Results for an Unconstrained Problem

The unconstrained problem consists of having sufficient energy available to all vehicles and a time window large enough for the vehicles to visit every vertex. Figure 3.1 shows some representative results of the routes generated by the metaheuristic algorithms. Note that in the unconstrained case, the EHGA does not visit all available vertices, whereas the ACO algorithm does visit all available vertices. This is most likely due to the combination of static probabilities used in the population-generating algorithm (See Algorithm 7) and parent incompatibility (see Algorithm 9).

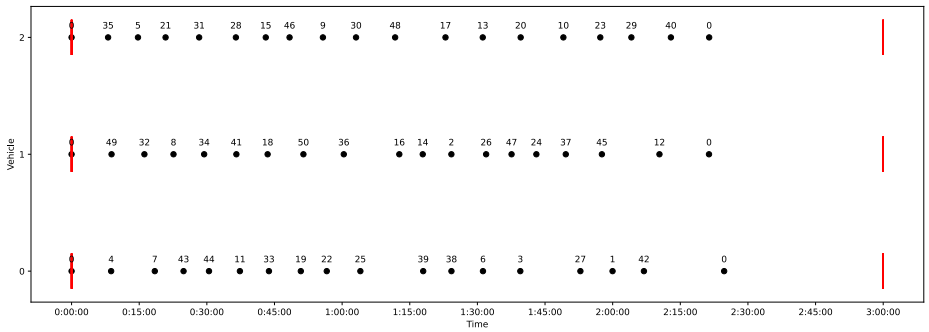


(a) Routes Generated by ACO

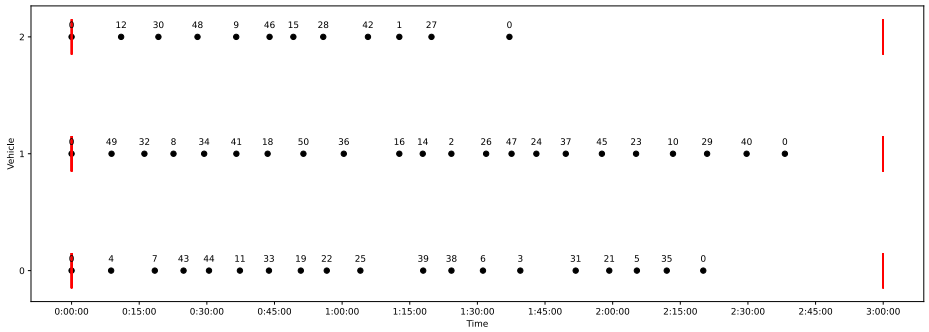
(b) Routes Generated by EHGA

Fig. 3.1: A Visual Representation of ACO and EHGA Unconstrained Results. The depot is depicted as a black triangle, and the labeled dots are the vertices of the problem. The routes for vehicles 0, 1, and 2 are depicted in blue, orange, and green, respectively.

Figure 3.2 shows a different visual representation of the results displayed in Figure 3.1, with a focus on the time at which each vertex was visited. Note that each vehicle in the ACO approach tends to start and finish around the same time, while the EHGA approach does not. Given the similarities between the two approaches in generating results, it may be deduced that the crossover and various mutation operators play a role in having some vehicles having more vertices assigned to them than others.



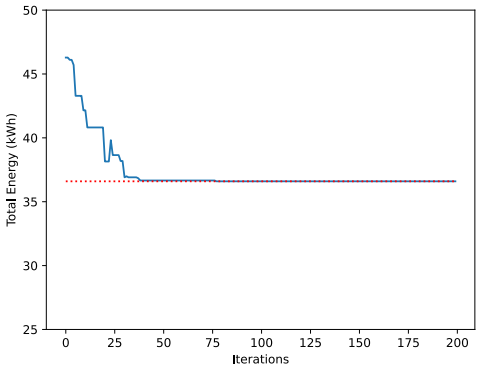
(a) The Timing of Routes Generated by ACO



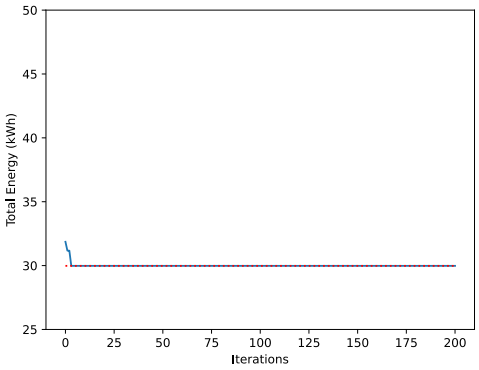
(b) The Timing of Routes Generated by EHGA

Fig. 3.2: Another Visual Representation of ACO and EHGA Unconstrained Results. The red bars show the beginning and end of the time window. The labeled dots show the order in which the labeled vertices are visited. Note that this figure is a companion figure to Figure 3.1.

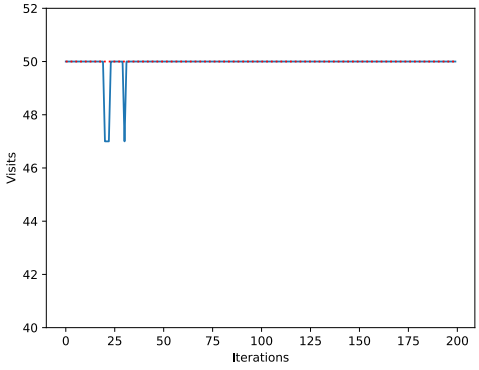
The convergence of both algorithms under the unconstrained test is shown in Figure 3.3. Most of the resulting solutions of the ACO algorithm visited all fifty destinations, and the total energy and utility of each solution converged to the final solution around the 50-iteration mark. Figure 3.3a shows the convergence of the energy of the solutions to the lowest energy found. Figure 3.3e shows the convergence of the utility of the ACO solutions as the algorithm reached its maximum utility found. The energy convergence of the EHGA is shown in Figure 3.3b, where the energy usage converges quickly. The utility convergence of the EHGA is shown in Figure 3.3f. Note that the EHGA implementation converges on a higher score even though it does not visit every destination.



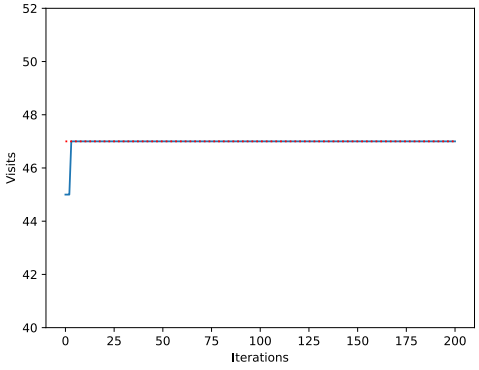
(a) ACO Energy Convergence



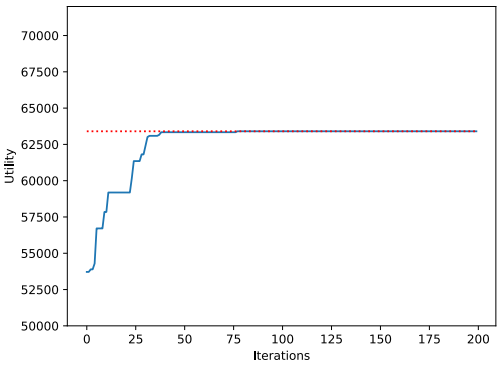
(b) EHGA Energy Convergence



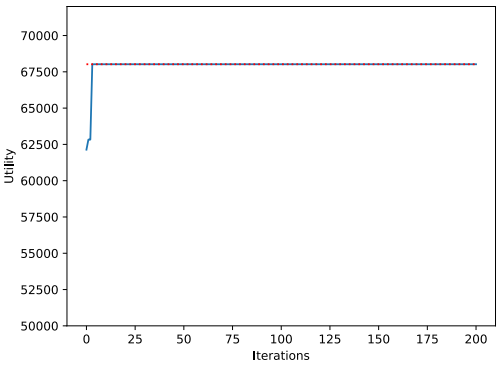
(c) ACO Visits Convergence



(d) EHGA Visits Convergence



(e) ACO Utility Convergence

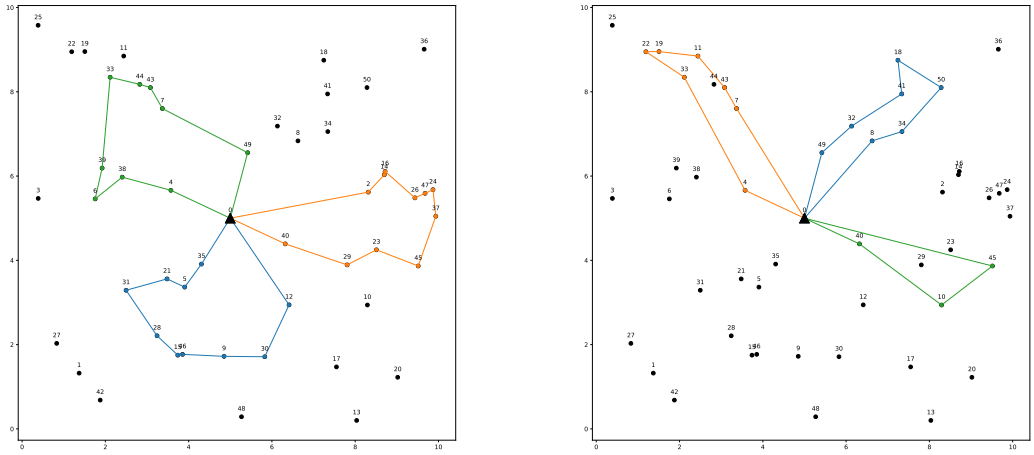


(f) EHGA Utility Convergence

Fig. 3.3: Convergence Results of ACO Unconstrained Test

3.5.2 Results for the Energy-Constrained Problem

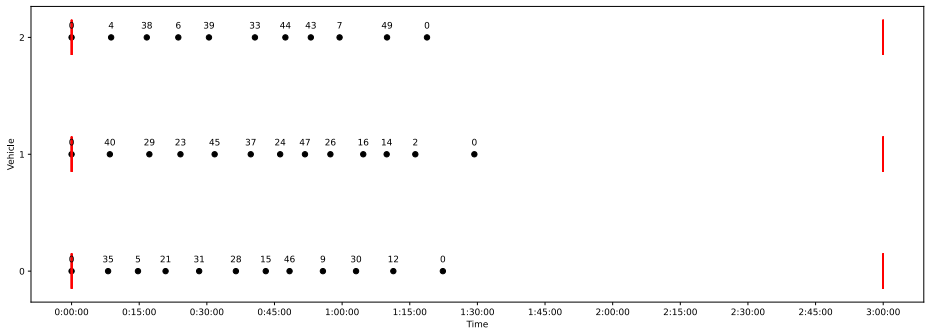
Given the results of the control group in Section 3.5.1, it becomes an area of interest to see what behaviors each algorithm exhibits given an energy constraint. The energy constraint forced each vehicle to use only 10% of its battery capacity, which was not enough to visit each vertex in the problem. Figure 3.4 shows the effect of the energy constraint on a solution. Note that not every vertex in the graph is visited for either solution technique and that both solutions tend to visit vertices that are closer to the center of the map. Figure 3.5a shows the timing of the routes, which ended around the same time for both algorithms.



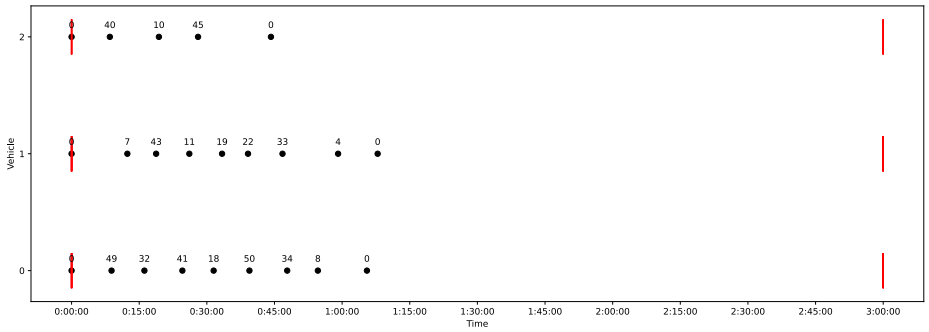
(a) Routes Generated by ACO

(b) Routes Generated by EHGA

Fig. 3.4: Visual Representation of ACO and EHGA Energy-Constrained Results. The depot is depicted as a black triangle, and the labeled dots are the vertices of the problem. The routes for vehicles 0, 1, and 2 are depicted in blue, orange, and green, respectively.



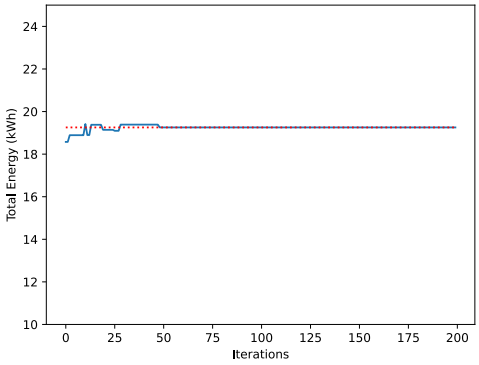
(a) The Timing of Routes Generated by ACO



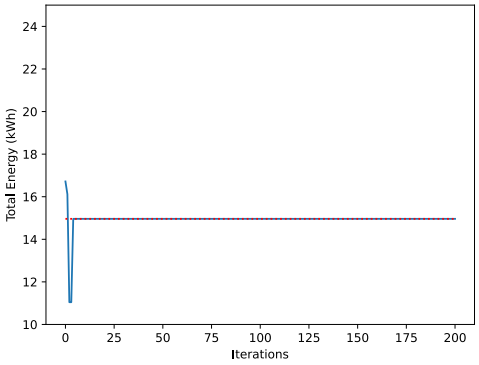
(b) The Timing of Routes Generated by EHGA

Fig. 3.5: Another Visual Representation of ACO and EHGA Energy-Constrained Results. The red bars show the beginning and end of the time window. The labeled dots show the order in which the labeled vertices are visited. Note that this figure is a companion figure to Figure 3.4.

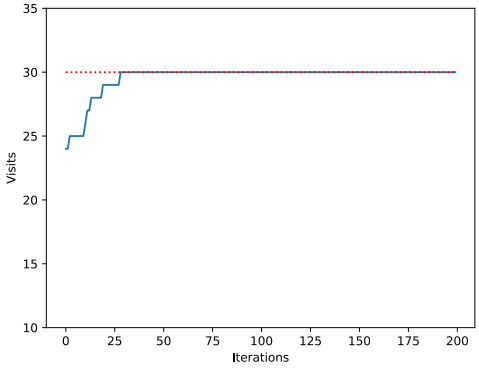
Figure 3.6 shows the convergence results of the energy-constrained test. Given that both algorithms aim to simultaneously maximize the number of visits made and minimize the total energy used, there is a visible trade-off between visiting more destinations and using more energy to visit the destinations in both approaches. Note, in particular, that for the ACO implementation, as the number of visits converges, the total energy increases, and then as the algorithm continues, the total energy decreases. This is because the weights of the objective function are more heavily based on the number of destinations that are visited and the pheromones used to seek shorter paths. In this case, the ACO has a much higher utility than the EHGA.



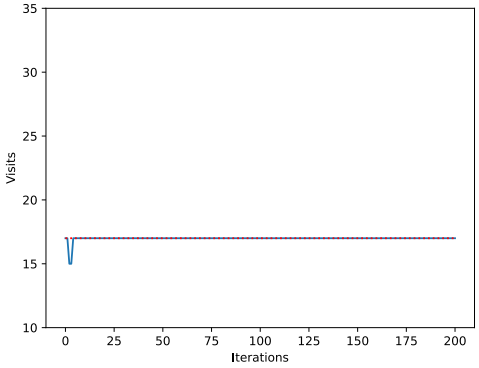
(a) ACO Energy Convergence



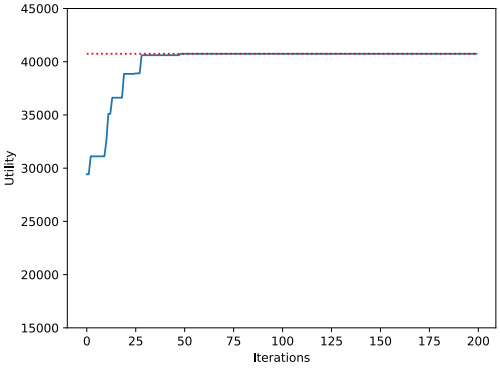
(b) EHGA Energy Convergence



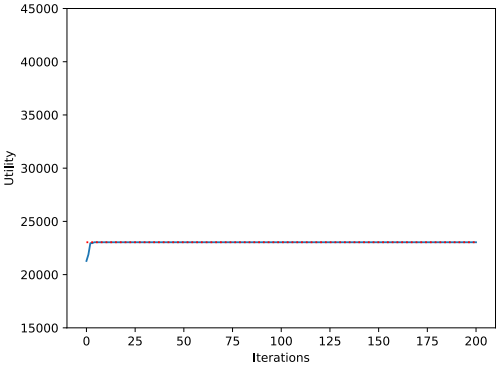
(c) ACO Visits Convergence



(d) EHGA Visits Convergence



(e) ACO Utility Convergence



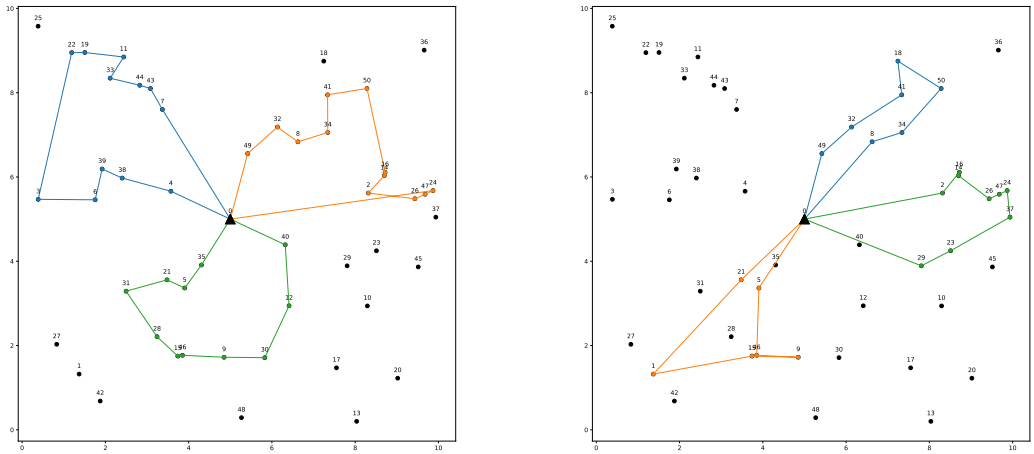
(f) EHGA Utility Convergence

Fig. 3.6: Convergence Results of the Energy-Constrained Test

3.5.3 Results for the Time-Constrained Problem

Given that the 10% energy constraint yielded results in which vehicles finished their routes around the halfway point between the start and end of the three-hour window, to test for similar results, the energy constraint was lifted and a time window of 1.5 hours was created. Note in Figure 3.7 that the ACO-generated routes tend to stay near the center of the map, while the EHGA tends to venture out further.

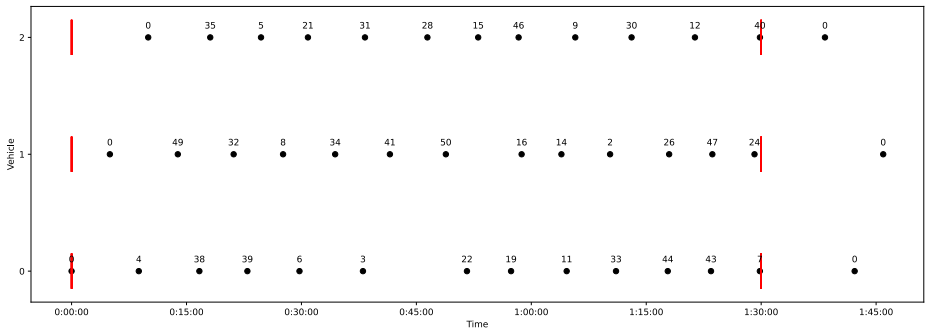
To show the effect of different starting times, each vehicle was started five minutes after the previous vehicle. This is shown in Figure 3.8a. Note that the return to the depot does not need to occur within the time window. This decision is arbitrary and can be easily changed by modifying a few lines of algorithms 4, 7, and 18.



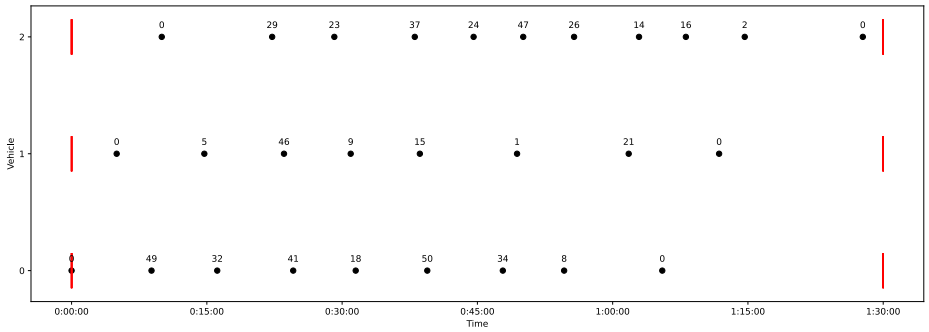
(a) Routes Generated by ACO

(b) Routes Generated by EHGA

Fig. 3.7: Visual Representation of ACO Time-Constrained Results. The depot is depicted as a black triangle, and the labeled dots are the vertices of the problem. The routes for vehicles 0, 1, and 2 are depicted in blue, orange, and green, respectively.



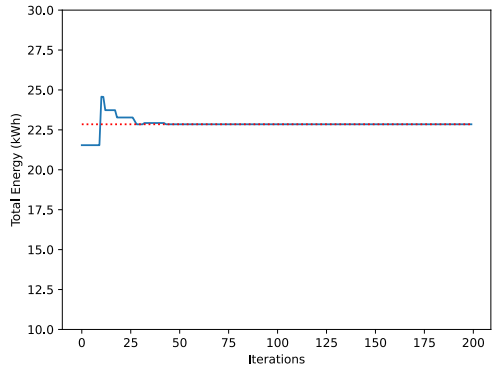
(a) The Timing of Routes Generated by ACO



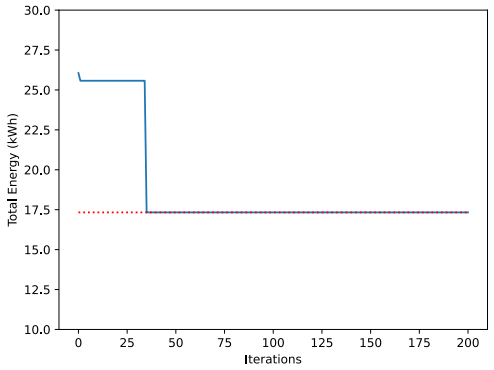
(b) The Timing of Routes Generated by EHGA

Fig. 3.8: Another Visual Representation of ACO and EHGA Time-Constrained Results. The red bars show the beginning and end of the time window. The labeled dots show the order in which the labeled vertices are visited. Note that this figure is a companion figure to Figure 3.7

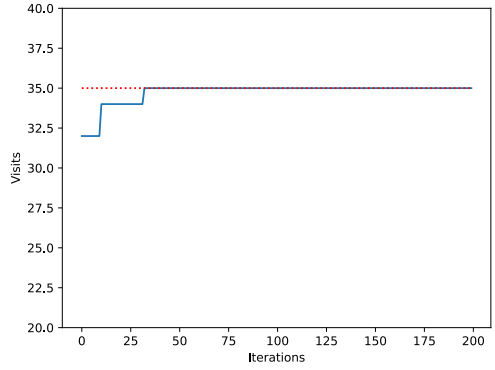
Figure 3.9 shows the convergence results of the ACO time-constrained test. Note how the energy used and the number of visits affect the utility, especially noting that with each increase in visits in the ACO implementation, the energy used by the fleet increases and the utility improves. Note that the EHGA converges to a lower number of visits with much lower energy usage in order to marginally increase the utility.



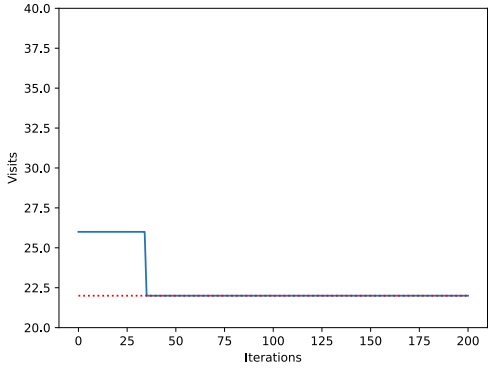
(a) ACO Energy Convergence



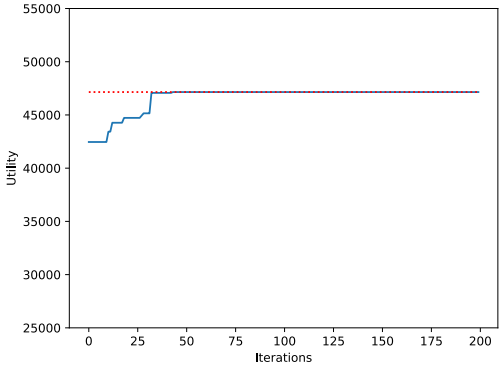
(b) EHGA Energy Convergence



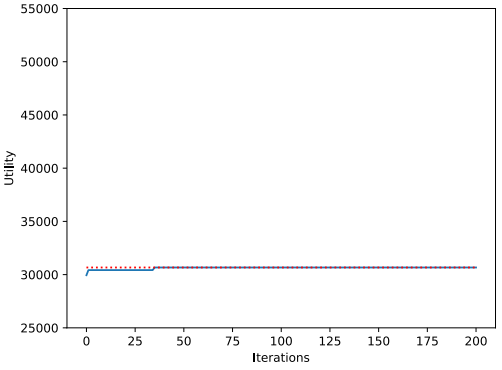
(c) ACO Visits Convergence



(d) EHGA Visits Convergence



(e) ACO Utility Convergence

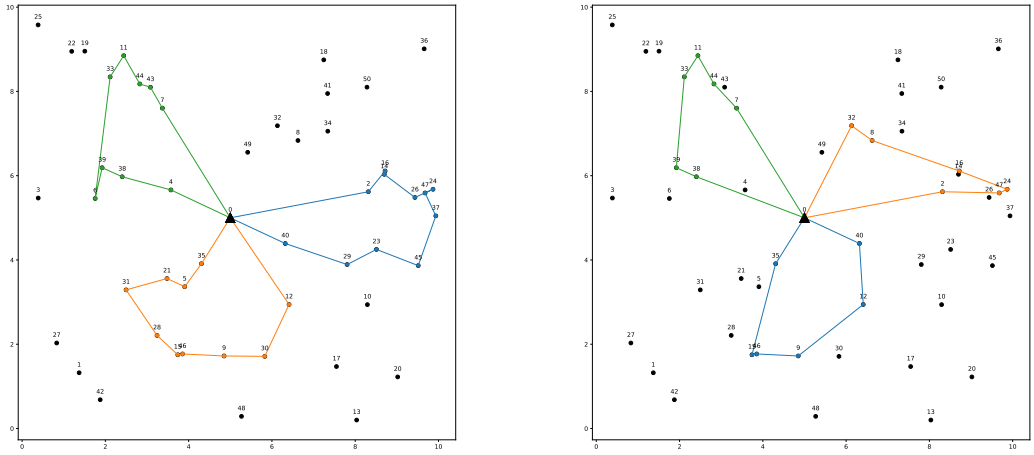


(f) EHGA Utility Convergence

Fig. 3.9: Convergence Results of ACO Time-Constrained Test

3.5.4 Results for the Time-and-Energy-Constrained Problem

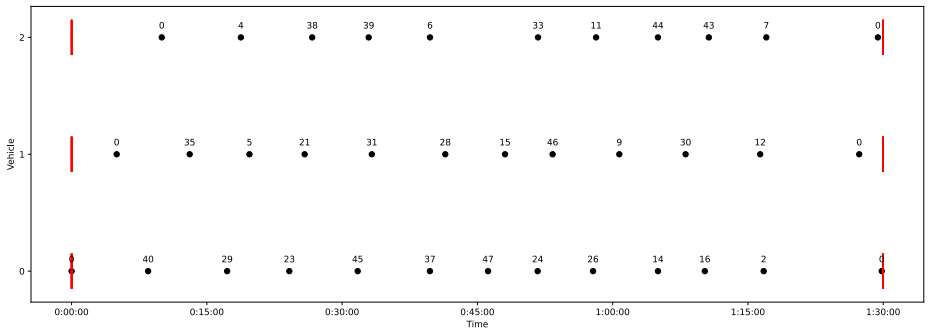
For a final test, the energy constraint was added back into the time-constrained problem to test whether similar results occur when both constraints are at play. Figure 3.10a shows a visual representation of the routes generated using both types of constraints. The times of the routes are shown in Figure 3.11a.



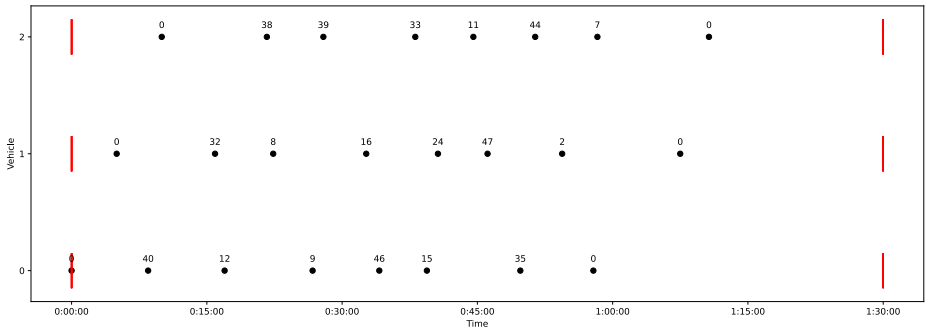
(a) Routes Generated by ACO

(b) Routes Generated by EHGA

Fig. 3.10: Visual Representation of ACO Time-and-Energy-Constrained Results. The depot is depicted as a black triangle, and the labeled dots are the vertices of the problem. The routes for vehicles 0, 1, and 2 are depicted in blue, orange, and green, respectively.



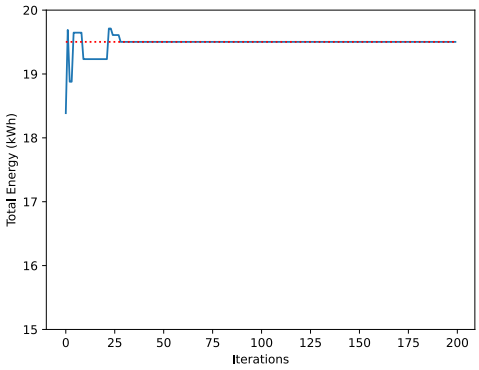
(a) The Timing of Routes Generated by ACO



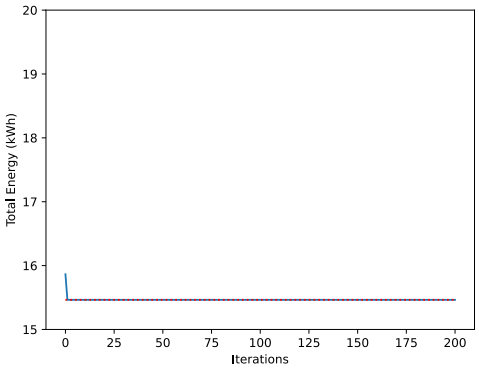
(b) The Timing of Routes Generated by EHGA

Fig. 3.11: Another Visual Representation of ACO and EHGA Time-and-Energy-Constrained Results. The red bars show the beginning and end of the time window. The labeled dots show the order in which the labeled vertices are visited. Note that this figure is a companion figure to Figure 3.10

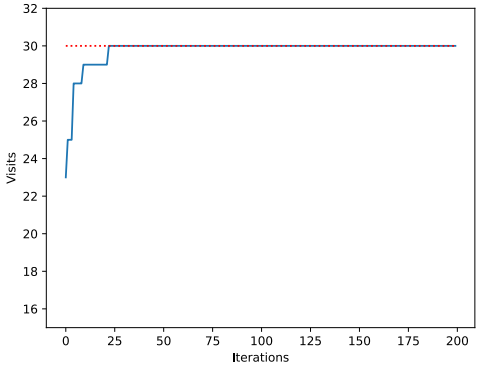
The convergence of the two algorithms under both constraints is shown in Figure 3.12. Note that for the ACO, the number of visits drives the utility, and the energy reduction marginally increases the score. Note that the EHGA struggled to find better solutions after a rapid initial convergence period, which is a normal pitfall for GAs in general [61].



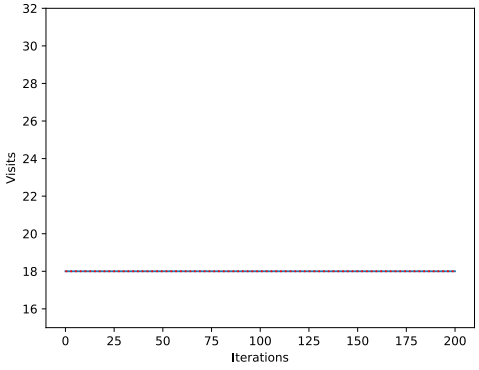
(a) ACO Energy Convergence



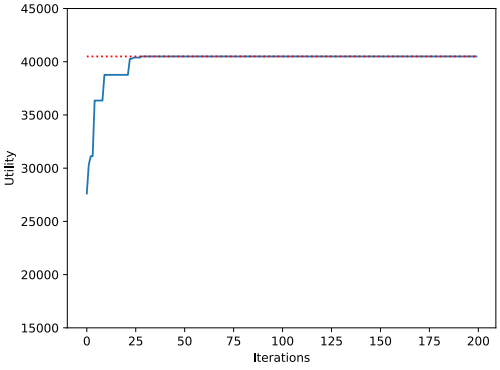
(b) EHGA Energy Convergence



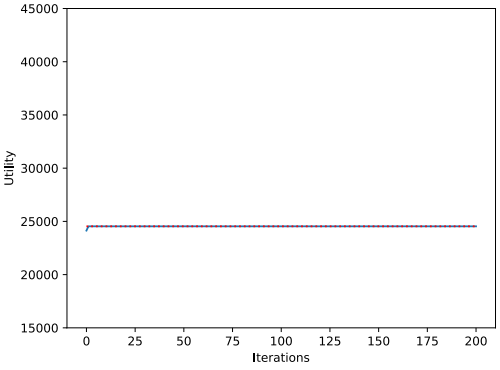
(c) ACO Visits Convergence



(d) EHGA Visits Convergence



(e) ACO Utility Convergence



(f) EHGA Utility Convergence

Fig. 3.12: Convergence Results of ACO Time-and-Energy-Constrained Test

3.5.5 Statistical Comparison

Given that the results in sections 3.5.1-3.5.4 are only singular examples of the two algorithms, it is important to provide statistical data to paint a somewhat more holistic picture of the nature of both algorithms. This section displays the statistical results of running each algorithm twenty times per scenario mentioned in the introduction of this section, totaling 60 samples for technique in each histogram. The remainder of this section shows histogram data of the energy results for each test, the visits results for each test, the utility results for each test, and the run time results for each test. In general, the ACO implementation uses more energy, but also visits more vertices, and takes less time to run.

Energy Histograms

Figure 3.13 shows histograms regarding the energy consumption of each solution. For each of the tests, the ACO tends to have much less variance. This is most likely due to the genetic algorithm's tendency to search more of the solution space.

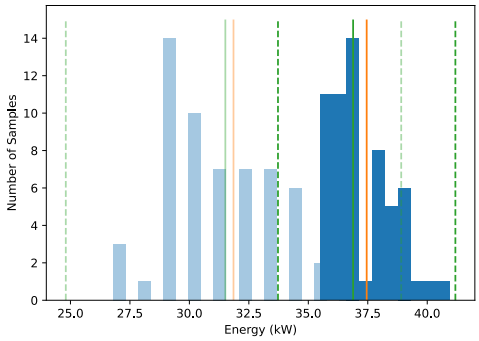
Figure 3.13a shows the test results of the unconstrained case. In general, the ACO implementation tends to use more energy, but this is most likely because the ACO implementation also tends to visit more vertices (see Figure 3.14a).

Figure 3.13b shows the results of the energy-constrained case. Note that 10% of the battery's capacity was 6.8 kW, and therefore, the maximum amount of energy that can be consumed from the three vehicles combined is 20.4 kW. Note that the ACO implementation tends to have energies very close to the energy constraint, while the EHGA implementation does not. This is most likely due to the probability matrices in both approaches. The probability matrix in 7 for the EHGA is more static and thus leads to more stochastic results. The probability matrix in Algorithm 18 uses pheromones to converge to a solution, leading to less stochastic results.

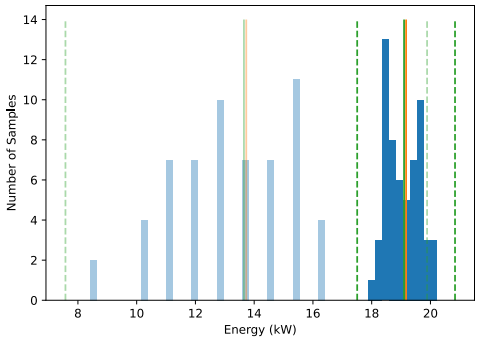
Figure 3.13c shows the results of the time-constrained case. Given time constraints, the two approaches have very similar energy consumption to each other as opposed to the previous two tests where the ACO tends to consume much more energy than the EHGA. There exist a number of contributors as to why this is the case. One possible explanation

is that the EHGA tends to visit fewer vertices that tend to be farther apart and uses an exact method in local optimization, while the ACO algorithm tends to visit more vertices that are closer to each other and uses metaheuristic methods in local optimization.

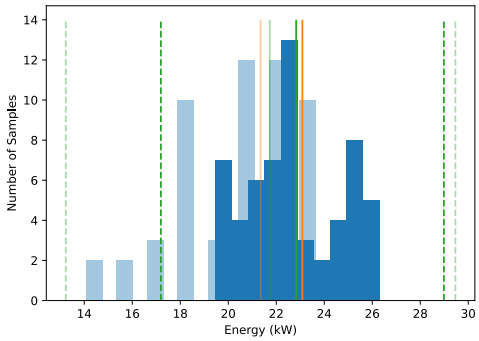
Figure 3.13d shows the results of the time-and-energy-constrained case. It is interesting to note that the characteristics of how the algorithms behave under the energy constraint dominate the behavior of each algorithm when the algorithm is run using both constraints. In other words, the ACO algorithm behaves very similarly under both constraints as it does with using only the energy constraint by residing close to the energy constraint. The EHGA behaves similarly as well, as its mean and median are not as close to the ACO algorithm's mean and median.



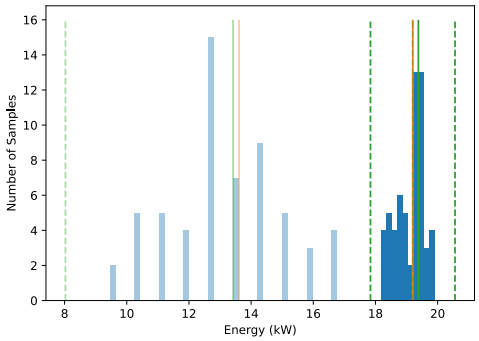
(a) Energy for Unconstrained Test



(b) Energy for Energy-Constrained Test



(c) Energy for Time-Constrained Test



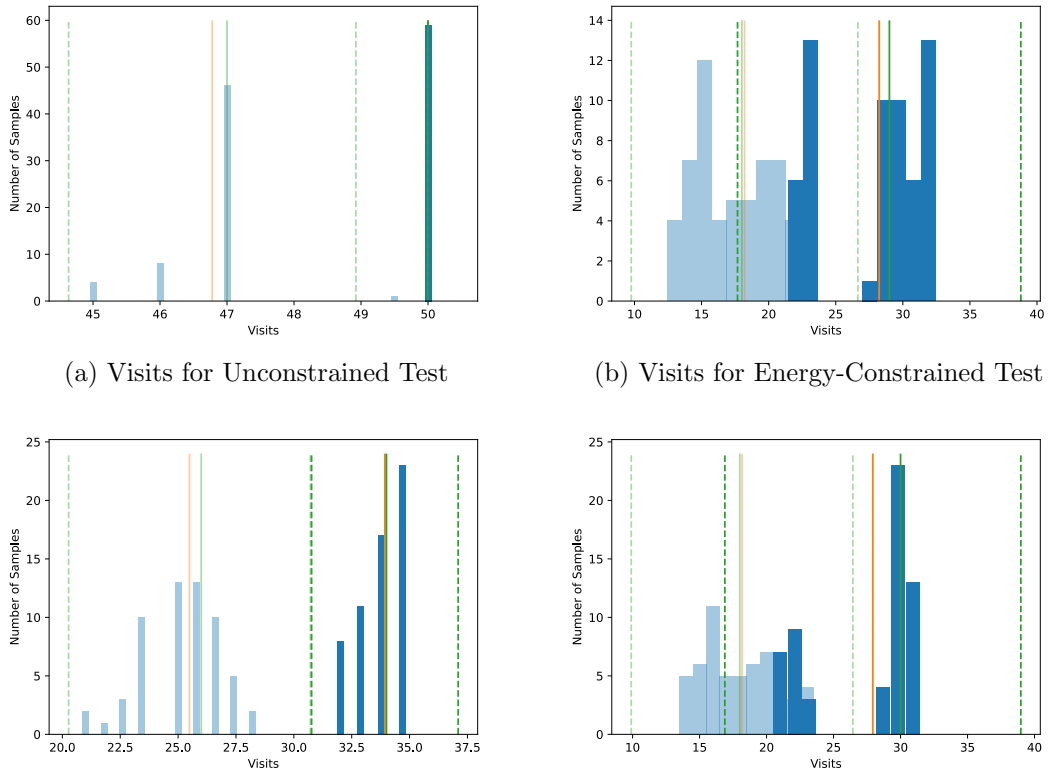
(d) Energy for Time-and-Energy Test

Fig. 3.13: Energy Histograms for Each Test. The vibrant colors represent the ACO results and the muted colors represent the EHGA results. The mean is represented in orange, the median is represented in green, and the dotted lines represent 3σ from the mean.

Visits Histograms

Figure 3.14 shows histograms regarding the number of vertices visited in each solution. Overall, the ACO algorithm tends to visit more vertices, most likely due to the dynamic nature of the probability matrix in Algorithm 18, and the EHGA's necessity of having compatible parents (see Algorithm 9).

A small discussion of Figure 3.14 is as follows. Figure 3.14a shows the results of the unconstrained test. Note that the ACO algorithm was able to visit all 50 destinations, while the EHGA was generally able to visit most of the destinations. This is most likely due to the more random route-generating strategy that the EHGA incorporates, i.e. the EHGA is more likely to select a vertex that violates constraints. Figure 3.14b shows the results for the energy-constrained test. It is interesting to note the overlap between the two approaches. Note that the ACO approach generally visits more vertices given the energy constraint. Figure 3.14c shows the results of the time-constrained test. Note that both approaches tend to visit more vertices given the time constraints versus energy constraints. Figure 3.14d shows the results of the time-and-energy-constrained test. Again, note that the characteristics of the energy-constrained test dominate the characteristics of the time-and-energy-constrained test.



(a) Visits for Unconstrained Test

(b) Visits for Energy-Constrained Test

(c) Visits for Time-Constrained Test

(d) Visits for Time-and-Energy Test

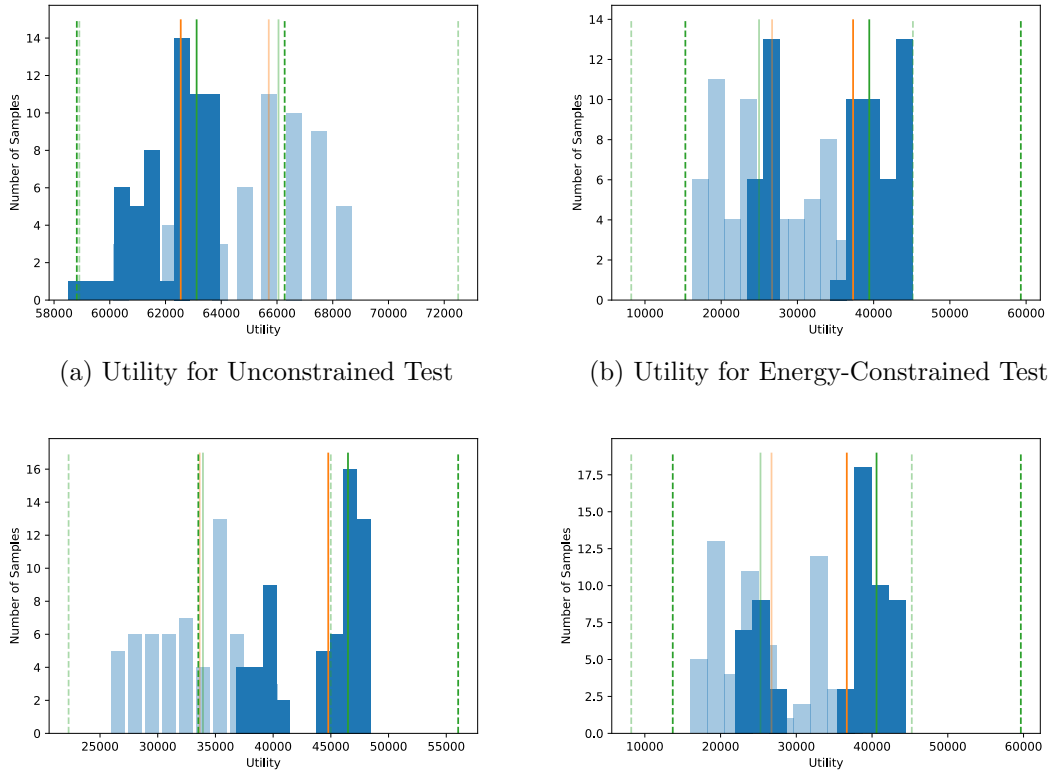
Fig. 3.14: Visits Histograms for Each Test. The vibrant colors represent the ACO results and the muted colors represent the EHGA results. The mean is represented in orange, the median is represented in green, and the dotted lines represent 3σ from the mean.

Utility Histograms

The utility histograms are shown in Figure 3.15. For the unconstrained case (Figure 3.15a), the EHGA generally outperformed the ACO algorithm. This is because the EHGA generally used less energy while visiting close to the same number of vertices. For the constrained cases, the ACO generally had a higher utility.

There are several observations to be made about Figure 3.15. Figure 3.15a shows the utility of the two approaches in the unconstrained test. Note that given that the number of visits remained the same for the ACO approach, the utility histogram of the ACO approach looks like a mirrored image to its energy histogram (Figure 3.13a). Furthermore, note that the EHGA outperformed the ACO in the unconstrained case due to its energy

savings. Figure 3.15b shows the results of the energy-constrained test. In this case, the ACO outperformed the EHGA with a significant overlap between the two approaches. Figure 3.15c shows the results of the time-constrained test. In this case, the ACO outperformed the EHGA with much less overlap between the two approaches. Figure 3.15d shows the results of the time-and-energy-constrained test. On average, the ACO outperforms the EHGA with some overlap between the two approaches.



(a) Utility for Unconstrained Test

(b) Utility for Energy-Constrained Test

(c) Utility for Time-Constrained Test

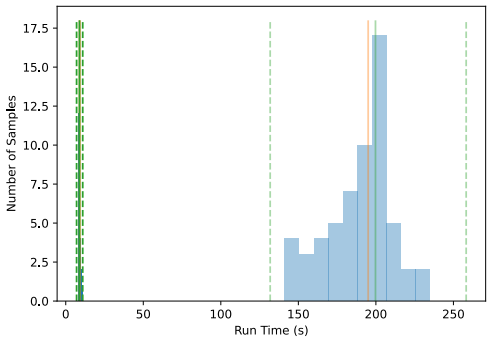
(d) Utility for Time-and-Energy Test

Fig. 3.15: Utility Histograms for Each Test. The vibrant colors represent the ACO results and the muted colors represent the EHGA results. The mean is represented in orange, the median is represented in green, and the dotted lines represent 3σ from the mean.

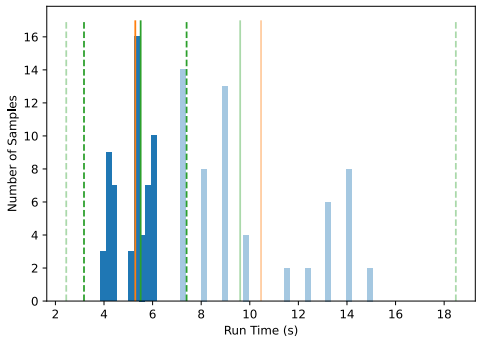
Run Time Histograms

Figure 3.16 shows a series of histograms concerning the run time, in seconds, for the two algorithms. In general, the ACO algorithm performed faster than the EHGA and performed with much less variance. Note that for the tests with energy constraints (Figures 3.16b and 3.16d) there is more of an overlap of run times for the two algorithms, suggesting that energy constraints have a direct correlation with the run time of the EHGA. The ACO algorithm was generally unaffected by the different tests in terms of speed.

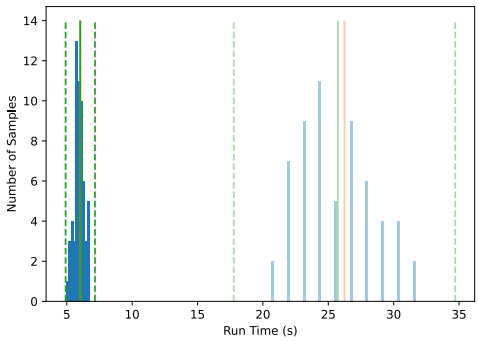
Figure 3.16a shows the run time of the unconstrained test. Although the EHGA had higher utilities on average, the EHGA also has a much longer run time. Figure 3.16b shows the run time of the energy-constrained test. There is some overlap between the two approaches, implying a correlation between an energy constraint and running time. The ACO approach tends to have much less variance in terms of running time. Figure 3.16c shows the run time of the time-constrained test. Although the separation between the two approaches is less exaggerated than the separation in the unconstrained case, there is a large difference in running time between the two approaches. Figure 3.16d shows the run time of the time-and-energy-constrained test. Again, the characteristics of the energy-constrained test dominate the time-window constraints.



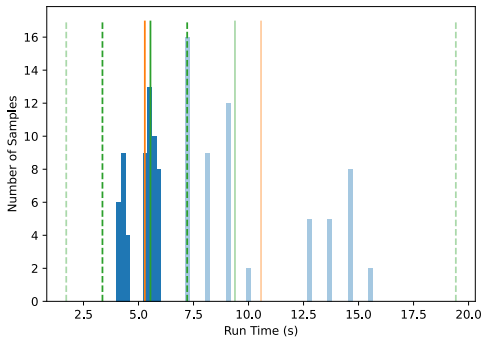
(a) Run Time for Unconstrained Test



(b) Run Time for Energy-Constrained Test



(c) Run Time for Time-Constrained Test



(d) Run Time for Time-and-Energy Test

Fig. 3.16: Run Time Histograms for Each Test. The vibrant colors represent the ACO results and the muted colors represent the EHGA results. The mean is represented in orange, the median is represented in green, and the dotted lines represent 3σ from the mean.

3.6 Using Generated Routes in a Scheduling Paradigm

Now that the ACO approach has been developed and discussed throughout this chapter, the final result of this chapter is to demonstrate ACO-generated routes in a scheduling paradigm. This section covers the implementation details of converting open-schedule problems into flexible-schedule problems by creating routes with ACO and then using the generated routes in a scheduling paradigm.

3.6.1 Creating Routes with EVRPE

Simply put, an open-schedule problem scenario consists of locations to visit, vehicles to visit the locations, roads between locations, and a way for the vehicles to refuel. Creating an open-schedule problem scenario entails choosing locations for the vehicles to visit and finding energy-aware paths between any two locations.

Each component of the open-schedule problem can be simulated using software tools. One software tool that uses realistic road and vehicle models is the Electric Vehicle Range and Prediction Estimator (EVRPE) [68] which builds on the FASTSim tool developed by the National Renewable Energy Lab (NREL) [69]. FASTSim is a tool that models vehicles both quickly and accurately [69].

The EVRPE takes in road information from Google's API along with a user-defined vehicle model to estimate the energy usage of the vehicle as it travels on defined road segments. The estimated energy usage is saved in a graph with the edges representing the road segments with weights for distance, time, and energy. After creating the model, the Bellman-Ford algorithm [70] is run to find the most energy-efficient route between two given locations and displays the route on a map [68].

Given that the EVRPE software finds the most energy-efficient route between two locations, it is very useful in creating an open-schedule problem scenario. Several locations may be selected for the open-schedule scenario, and an energy-efficient route to get from one location to any other location may be created for the open-schedule problem scenario.

Since the open-schedule problem can represent package delivery, a vehicle model representative of package delivery was created. FedEx, which delivers packages, has adopted the

2022 Ford E-Transit in a pilot program for electric vehicle fleets [71]. Therefore, a model based on the 2022 Ford E-Transit was created. At the time of writing, there was no model of the 2022 Ford E-Transit on the National Renewable Energy Lab's repository of vehicle models [72]. However, an E-Transit model can be created from the 2022 Ford Transit model found in the repository and using online sources detailing the vehicle specifications, namely [67] and [73].

The ACO algorithm was run on arbitrarily-chosen places in Ogden Valley using 5 vehicles, 25 ants, and 300 iterations, as well as a 15% capacity battery constraint and a 1.5-hour time constraint. Table 3.7 shows the characteristics of the route, including the number of visits, the energy used by the vehicles, and the time to run each route. Figure 3.17 shows the Ogden Station as the depot, marked in blue, and the destinations marked in red on a map. A more simple visual representation of the routes is shown in Figure 3.18. Notably, given ample time and enough battery charge, every vertex is visited. The convergence of the EVPRE results is shown in Figure 3.19, and data regarding the routes is shown in table 3.7.

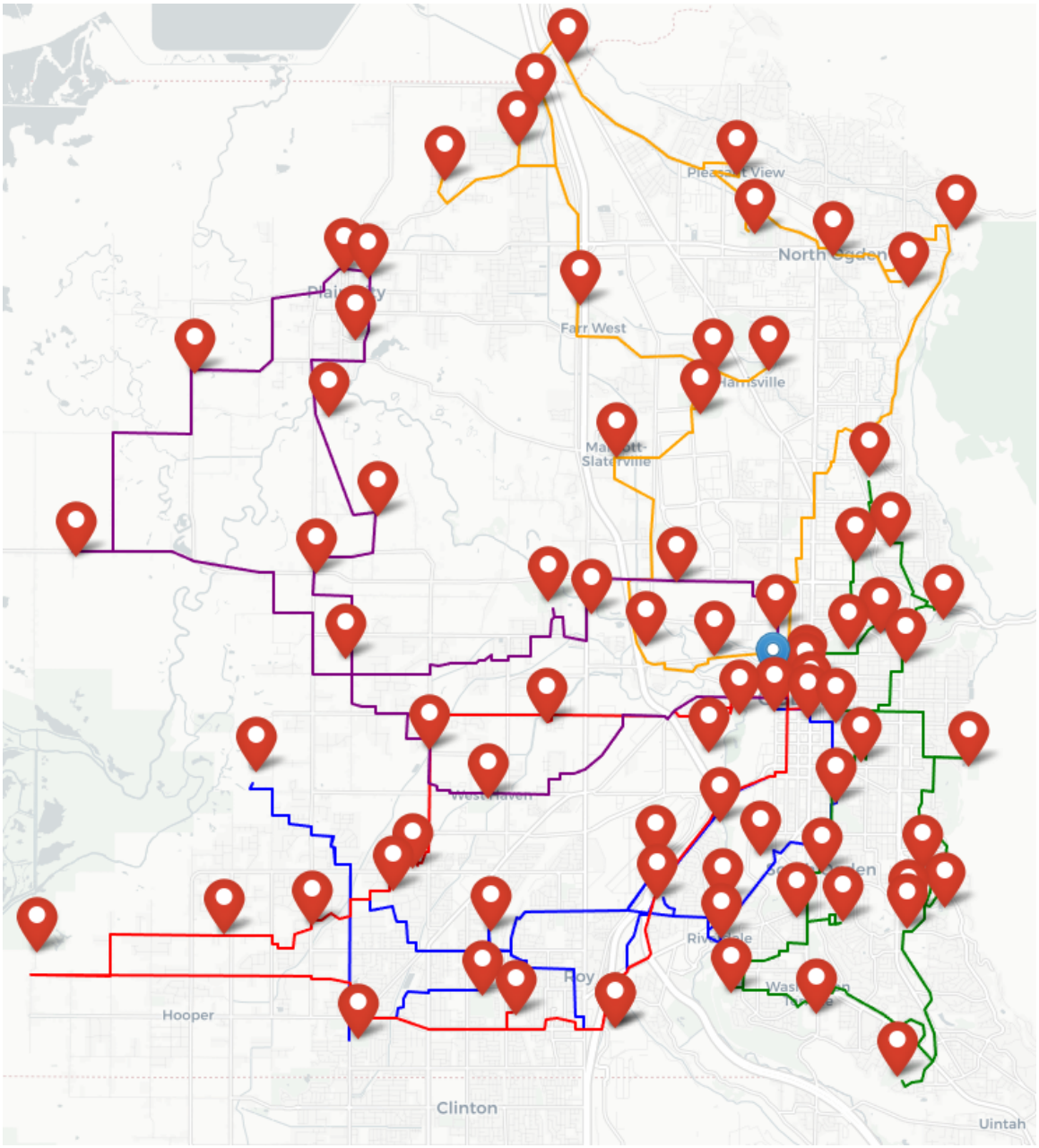
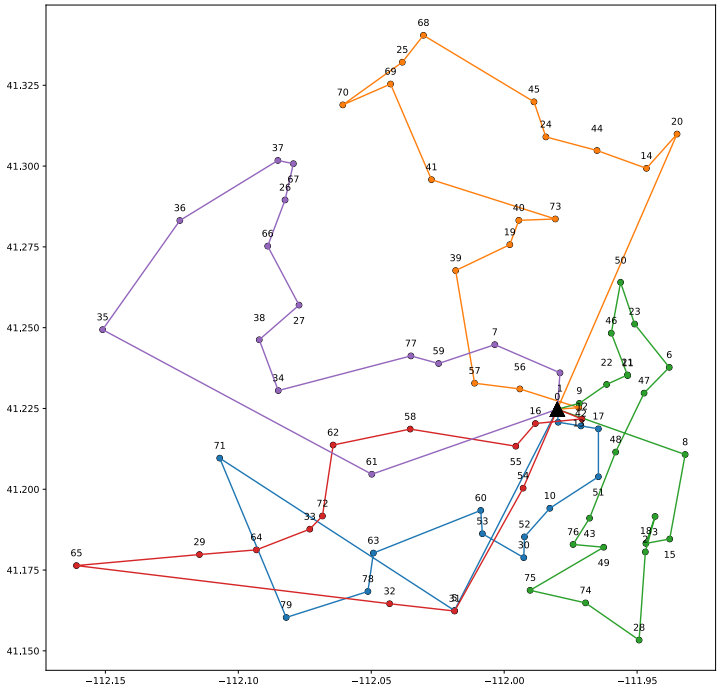
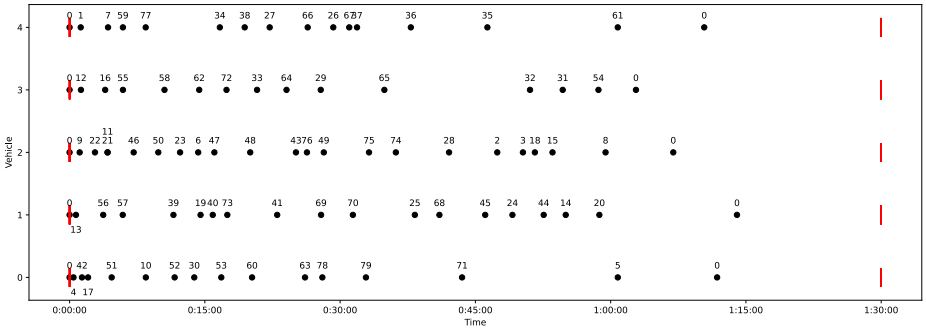


Fig. 3.17: A Visual Representation of the Ogden Routes, where the Ogden bus station is shown with a blue marker, and points of interest are shown with red markers. The lines between any two markers are the most energy-efficient path between the two markers.



(a) A Simple Visual Representation of Routes Generated by ACO. The depot is depicted as a black triangle, and the dots are the vertices of the problem. The routes for vehicles 0, 1, 2, 3 and 4 are depicted in blue, orange, green, red, and purple, respectively



(b) A Visual Representation of the Timing of Routes Generated by ACO. The red bars show the beginning and end of the time window. The labeled dots show the order in which the labeled vertices are visited

Fig. 3.18: Visual Representation of EVPRE Results

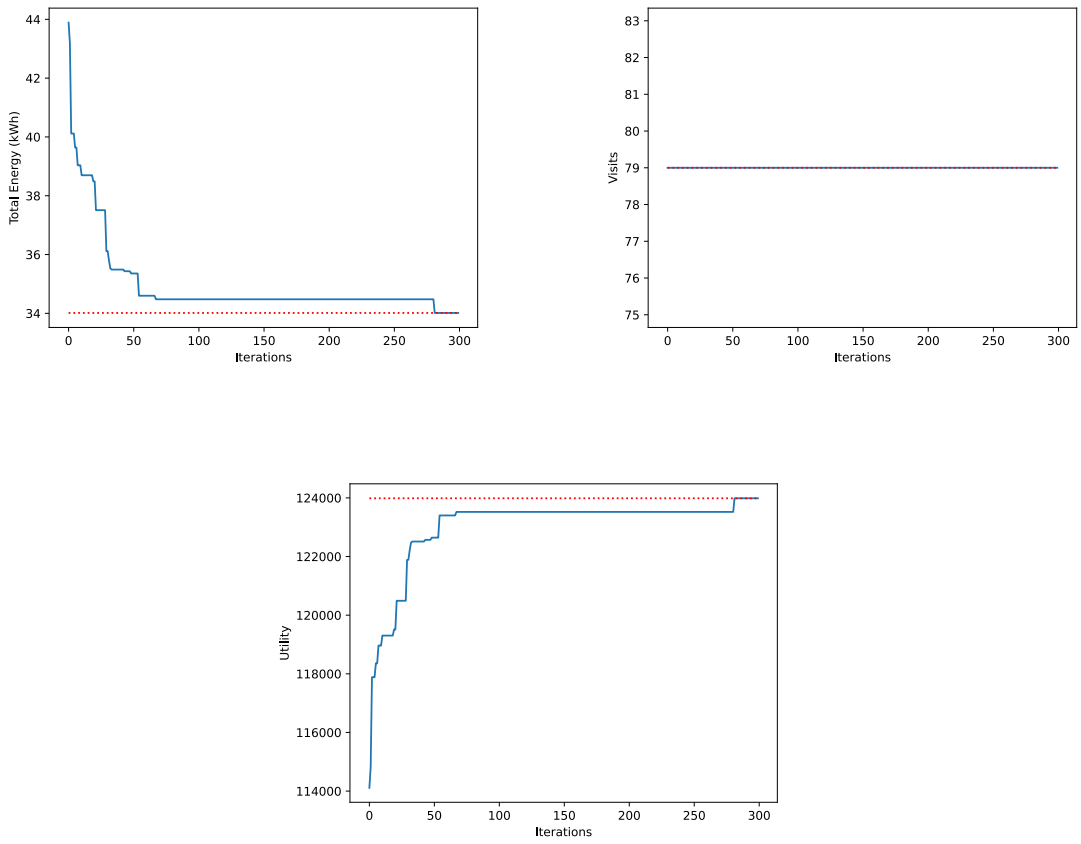


Fig. 3.19: Convergence Results of EVPRE Scenario

Route	Visits	Energy (kW)	Time
0	14	6.52	1:11:48.30
1	17	7.74	1:14:00.40
2	21	6.93	1:06:56.60
3	13	5.89	1:02:48.30
4	14	6.93	1:10:22.50

Table 3.7: Route Metrics for Ogden Scenario

3.6.2 Using Generated Routes in a Scheduling Paradigm

The routes generated in this section were used in the bin-packing scheduling approach used in [42]. The peak charging times were from 7:00 am to 11:00 pm and the charger used in the simulation had a capacity of 100 kW. There were five vehicles used in the simulation, all of which started and ended at 95% SOC. Each of the five routes had to be run a total of three times, totaling 15 routes for the five vehicles. Figure 3.20 shows the state of charge for all of the vehicles throughout the day.

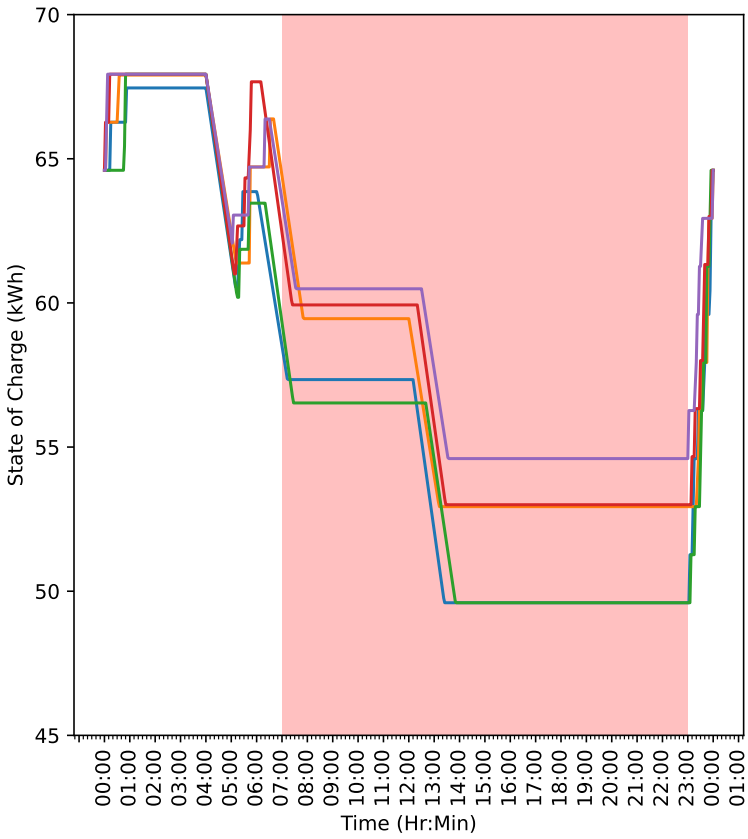


Fig. 3.20: The SOC of the EVs Going on ACO-Generated Routes

3.7 Conclusion

The routing problem that is studied in this chapter is the Energy-Aware, Team Orienteering Problem with Time Windows and Energy Constraints (EATOP). Two approaches to solving the EATOP are presented in detail and a comparison between the two approaches is conducted. The main purpose of the tests was to determine how the time and energy constraints affected the behavior of the algorithms. It was noted that the characteristics from the energy constraint dominate when both constraints are at play. The ACO approach generally uses more energy, but also generally visits more destinations and has a faster run time. The EHGA outperforms the ACO approach in terms of utility when neither constraint inhibits the visitation of all destinations. The ACO approach outperforms the EHGA when any constraint inhibits visits. To show the conversion from an open-schedule problem to a flexible-schedule problem, the chapter concludes by using routes generated by the Ant Colony in a route-scheduling paradigm.

CHAPTER 4
SCHEDULING

The main objective of this research is to create charging schedules that aim to minimize the cost of operating a heterogeneous vehicle fleet with heterogeneous constraints. This chapter covers the formulation of the EV-charge scheduling problem as an orienteering problem, and the solution technique of using Ant Colony Optimization to solve the scheduling problem. This method is compared against other methods found in the literature, then implementation details for a realistic scenario are presented. This chapter ends by showcasing the results of using the realistic scenario.

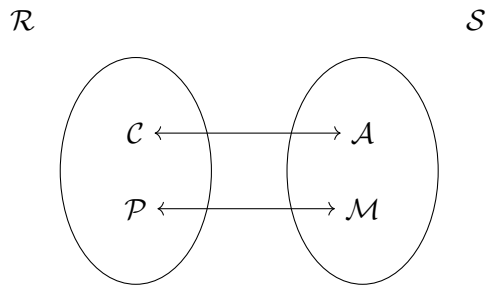
4.1 Scheduling as a Routing Problem

In order to use a routing approach to solve a scheduling problem, it is necessary to define the scheduling problem as a routing problem. A routing problem is generally formulated using a graph, where the vertices represent destinations for vehicles to visit, and the edges represent the travel between said destinations. A destination can use coordinates, $d = (x, y)$, for their location, with a given path $p = (d_i, d_j)$ between two destinations, d_i and d_j. A scheduling problem, on the other hand, has an activity for a vehicle to perform and a time to perform the activity, which can be organized into an ordered pair of a time and activity, (t, a). Switching between activities is called a *movement*. The movement between activities a_i and a_j is denoted by $m = ((t_i, a_i), (t_j, a_j))$. This section discusses the isomorphism between routing and scheduling problems and the technique used to formulate a scheduling problem as a routing problem.

Let $\mathcal{R}$ denote the set of objects that make up a routing problem with n destinations. Note that the set of (x, y) coordinates of the destinations of the problem, $\mathcal{C}$, along with the set of physical paths between them, $\mathcal{P}$, are elements of $\mathcal{R}$. Furthermore, let $\mathcal{S}$ denote the set of objects that make up a scheduling problem with n possible activities. Note that a set

of task-activity ordered pairs, $\mathcal{A}$, along with movements between activities $\mathcal{M}$ are elements of $\mathcal{S}$.

Consider a mapping function $f : \mathcal{A} \to \mathcal{C}$ such that for every pair $(t, a) \in \mathcal{A}$, there exists a unique pair $(x, y) \in \mathcal{C}$, or $f((t, a)) = (x, y)$. Furthermore, consider another mapping function, $g : \mathcal{M} \to \mathcal{P}$ such that for every movement $m = ((t_i, a_i), (t_j, a_j))$ there exists a unique physical path, $p = ((x_k, y_k), (x_l, y_l))$, or $g(m) = p$. Note that given the cardinalities of $\mathcal{R}$ and $\mathcal{S}$ are equal and that every element of $\mathcal{R}$ maps to a unique element in $\mathcal{S}$, these two functions are one-to-one, invertible, and therefore perform a bijection. This is visualized as follows:



Given the bijection above, the scheduling problem can be modeled as a routing problem with the routes produced by the routing problem corresponding directly to the schedules of the scheduling problem. Since the team orienteering problem starts at a depot, goes to a series of vertices, gathering scores by visiting the vertices, and returns to the depot, the flexible-schedule problem can be mapped onto a variation of the team orienteering problem with vertices representing the depot at a given time or a charger at a given time and edges representing the choices of moving between these activities. In this variation, there is no return to the depot. Each vertex cannot be visited more than once. The scheduling problem is formally defined using a directed, weighted graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Each activity in the set $\{\text{On Route}, \text{In Station}, \text{Charging}\}$, is associated with a specific discrete time with steps $0, \delta_t, 2\delta_t, \ldots, t_f$, where t_f is the length of the planning horizon and δ_t is the length of the time step. Each time-activity pair is represented as a vertex, $v \in \mathcal{V}$, and the ability to move from one activity to another is represented by an edge, $e = (i, j) \in \mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. In

the scheduling problem, it is not assumed that the graph is fully connected. Rather, it is assumed that the scheduling problem follows a predefined structure.

4.1.1 Scheduling Graph Structure

In formulating the scheduling problem as a routing problem, a schema of interpretation is necessary. In lieu of using x and y coordinates for vertices as in a traditional routing problem, the scheduling problem uses coordinates in terms of time and an activities. Thus, the vertices represent the vehicle performing one of the following activities: being in the station, going on a route, and charging.

The edges represent five transitory states: a vehicle going on a route, the start of a charging session, a continuation of a charging session, the end of a charging session, and a duration of time in a station or depot without charging. The vertices that represent charging sessions consist of two disjoint subsets that represent on-peak and off-peak charging sessions, respectively. The depots of each vehicle correspond to a station vertex at the vehicle's starting time. Figure 4.1 shows a visual representation of the different types of transitions that can occur in the first four time steps of the simulation.

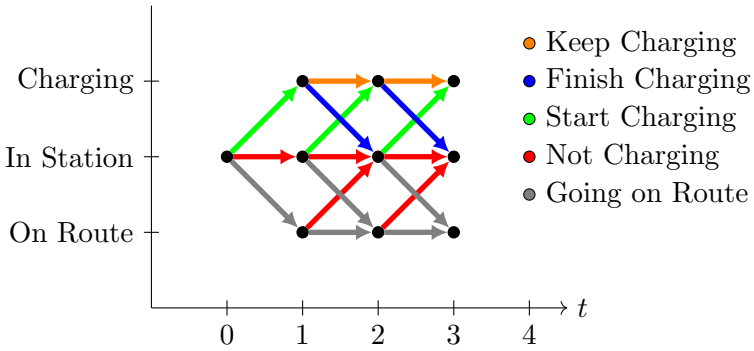


Fig. 4.1: The Types of Possible Transitions in the First 4 Timesteps

Labeling the Adjacency Matrix

Given the structure of the graph, the labeling of vertices is arbitrary, yet necessary, to create an adjacency matrix. Thus, a schema for interpretation is defined in this section, first in a single-vehicle case, then in a multi-vehicle case.

In order to understand the multi-vehicle case, it is important to describe the labeling in a single-vehicle case. Let $(0, t_f)$ be the time horizon for a vehicle k. The total number of in-station vertices for vehicle k is $\frac{t_f}{\delta_t}$, as each vehicle starts and ends in the station. Given that the vehicle starts and ends in the station, both the total number of charging vertices and the total number of on-route vertices is $\frac{t_f}{\delta_t} - 2$. Thus, the total number of vertices that vehicle k can visit is $3\frac{t_f}{\delta_t} - 4$. The range of labels, $(0, 3\frac{t_f}{\delta_t} - 5)$ are defined to be the index range, $\mathcal{I}_k$, for the vehicle k. A visualization of a single vehicle implementation is shown in Figure 4.2.

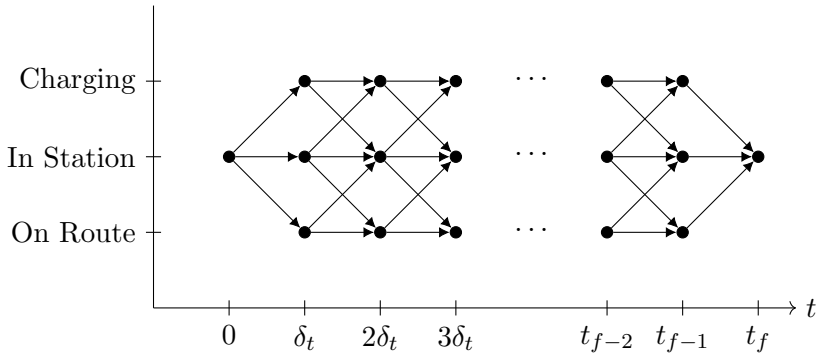


Fig. 4.2: A Visual Representation of a Generic Vehicle Horizon

Given that multiple vehicles can be in the station or on route, phantom vertices for the station and route vertices can be added to the graph's lattice structure, ensuring that multiple vehicles can be in the station or on route at the same time, while ensuring that two vehicles are not able to charge simultaneously at any given time. Thus, a scenario with two vehicles is shown in Figure 4.3.

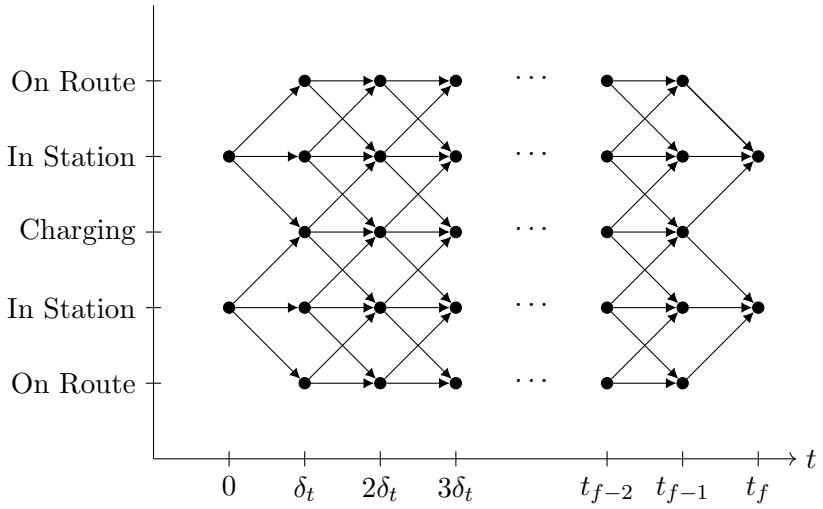


Fig. 4.3: A Visual Representation of a Generic Vehicle Horizon with Two Vehicles

Given that multiple vehicles may be in the station or on route at the same time but cannot charge at the same time, nor can any vehicle take another vehicle's vertices, the ranges of labels or *index ranges* of the vehicles are defined. Given that the first vehicle ranges from 0 to $i_1 = 3\frac{t_{f,1}}{\delta_t} - 5$, and that any subsequent vehicle k has two unique choices (going on route and staying in the station), the indices for vehicle k range from $i_{k-1} + 1$ to $i_{k-1} + 2\frac{t_{f_k} - t_{f,k-1}}{\delta_t} - 1$. These index ranges play an important role in the schedule-generating algorithm, Algorithm 28.

4.2 Formulation of the Scheduling Problem

As mentioned previously, the scheduling problem can be formulated as a variation of an orienteering problem, where the agents do not return to the depot. As described in Section 4.1, the vertices represent a time and an activity, the edges represent movements between activities, and the generated routes correspond to the schedules for each vehicle.

Given the agents, vertices, and edges, the scheduling problem can be constructed with an objective and constraints. The objective of this orienteering problem is to minimize the cost of energy consumed by charging the vehicles. The cost is scaled according to the time-of-use price strategy determined by the electric utility company [3]. Therefore, the

on-peak and off-peak costs are scaled by their respective prices. The constraints of the problem include intermediate and terminal bounds on the SOC of each vehicle and timing constraints for both vehicles and routes.

The solution to the scheduling problem herein is a set of activities for each vehicle, which form a schedule for all of the vehicles. The schedule with the lowest cost aims to minimize the amount of energy consumed during peak hours. The problem herein is titled the Team Orienteering Problem for Scheduling Electric Vehicles (TOPSEV). The remainder of this section covers the constraints and the TOPSEV in greater detail.

4.2.1 Time, SOC, and other Constraints

Given that the realities of scheduling vehicles include time-sensitive activities, as well as the vehicles themselves being constrained by their batteries, it is imperative to have constraints within the TOPSEV. These constraints include the timing of vehicles and routes and the state of charge (SOC) of vehicle batteries. This subsection details these and other important constraints.

In a scheduling paradigm, timing is everything, from the deadlines imposed by customers in a package-delivery scenario to the rigid adherence of periodic bus departures and arrivals, to the shifts of employees driving the vehicles. Given these constraints, the time windows for a vehicle to complete a route i are denoted as an ordered pair, $(\tau_{e,i}, \tau_{l,i})$, where the first entry is the earliest time a vehicle is allowed to depart on route i and the second entry is the latest time a vehicle is allowed to return from route i. Each vehicle also has a time window referred to as an *active window* such that every route completed by the vehicle must be completed during the active window of the vehicle. The active window for each vehicle k is denoted as an ordered pair, $(\tau_{e,k}, \tau_{l,k})$, where the first entry is when a vehicle k starts its operation and the second entry is when a vehicle k ends its operation.

In order to properly reflect the constraint of an EV battery, as well as to have continued operations of an EV fleet, the SOC of each vehicle must be considered. The SOC constraints are bounded above and below to ensure that the physical limitations of the battery are considered. The upper bound, s_u, may be the maximum capacity of the battery, but may

also be a lower value to increase the longevity of the battery. Likewise, the lower bound, s_l, may be the minimum capacity of the battery, but it can be a higher value to take the battery's health into account.

Other constraints exist for the convenience of a fleet operator, such as having only one charging session during a stay at the station. This constraint helps eliminate the unnecessary complexities of charging two or more vehicles by constantly switching between vehicles instead of charging them one at a time.

4.2.2 The Optimization Problem

Given the objective of minimizing the cost of operation and constraints such as timing, battery, and logistical constraints, it is now relevant to describe how each of these pieces fits together in creating the optimization problem. This subsection mathematically describes the optimization problem and introduces some of the notation used in the solution approach.

Much of the notation and definitions originate from dividing the vertex set into subsets. Given the problem graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, there exist subsets of the vertex set, $\mathcal{V}$, that represent the different activities of the vehicles. Let $\mathcal{V}_{r_i}$ be the vertices that represent the state of a vehicle being on a route i, let $\mathcal{V}_s$ be the vertices that represent the vehicle in the station, and let $\mathcal{V}_{\text{offpk}}$ and $\mathcal{V}_{\text{onpk}}$ be the sets of vertices that represent charging off-peak and off-peak, respectively. Given a vehicle k, its finite sequence or *list* of activities is denoted as $\mathcal{A}_k$ and consists of the labels of each vertex corresponding to each activity of the vehicle in the schedule. Therefore, a list $\mathcal{A}_k$ contains subsets of $\mathcal{V}_{r_i}$ for each route i taken, $\mathcal{V}_s$, $\mathcal{V}_{\text{offpk}}$, and $\mathcal{V}_{\text{onpk}}$, each denoted as $\mathcal{V}^k_{r_i}$, $\mathcal{V}^k_s$, $\mathcal{V}^k_{\text{offpk}}$, and $\mathcal{V}^k_{\text{onpk}}$, respectively.

More notation addresses the ideas necessary for constraints. Let $a(v)$ be the function that returns the activity of v, and let $t(v)$ be the function that returns the time of v of any vertex $v \in \mathcal{V}$. Furthermore, let m be the number of vehicles and n_r be the number of routes in the problem. The function $s_k(t)$ returns the SOC of a vehicle k at time t, where t is a multiple of δ_t between t_0 and t_f. The function $t(i)$ returns the time it takes to complete the route i.

As an optimization problem, the TOPSEV is defined as a minimization problem which

minimizes the cost of operating an electric fleet of vehicles. It adds the number of vertices from on-peak and off-peak charging, scaled by power and pricing factors. Given that each vertex in $\mathcal{V}$ has a time associated with it, the number of vertices that represent an activity is equivalent to the amount of timesteps spent on that activity. The charging rate in kilowatts, r, is multiplied by the sum of on-peak and off-peak time spent charging, with both scaled by a pricing factor. Both the on-peak and off-peak pricing factors, $\bar{\alpha}$ and $\bar{\beta}$, respectively, are measured in dollars per kilowatt-timestep.

The objectives and constraints are defined in Equation 4.1.

$$
\begin{aligned}
&\text{Minimize } r \sum_{k=0}^{m} \bar{\alpha}|\mathcal{V}^k_{\text{offpk}}| + \bar{\beta}|\mathcal{V}^k_{\text{onpk}}| \\
&\text{subject to} \\
&\quad (1) \quad \left| \bigcup_{k=0}^{m} \mathcal{V}^k_r \right| = \frac{1}{\delta_t} \sum_{i=0}^{n_r} t(i) \\
&\quad (2) \quad |\mathcal{A}_k| = \frac{t^k_f - t^k_0}{\delta_t} \quad \forall k \in \{0, \ldots, m\} \\
&\quad (3) \quad \tau_{e,k} \leq t(v) \leq \tau_{l,k} \quad \forall v \in \mathcal{A}_k,\ \forall k \in \{0, \ldots, m\} \\
&\quad (4) \quad \tau_{e,i} \leq t(v_i) \leq \tau_{l,i} \quad \forall v_i \in \mathcal{V}^k_{r_i},\ \forall k \in \{0, \ldots, m\},\ \forall i \in \{0, \ldots, n_r\} \\
&\quad (5) \quad s_l \leq s_k(t) \leq s_u \quad \forall t \in \{t_0, \ldots, t_f\},\ \forall k \in \{0, \ldots, m\} \\
&\quad (6) \quad s_k(0) = s_{k_0} \quad \forall k \in \{0, \ldots, m\} \\
&\quad (7) \quad s_k(t_f) \geq s_{t_f} \quad \forall k \in \{0, \ldots, m\}
\end{aligned}
\tag{4.1}
$$

Constraint 1 ensures that each route is completed by counting the total number of route vertices in a solution and comparing that count to the total time taken to complete the routes divided by the time step δ_t. Constraint 2 ensures that for each activity list, there are as many activities as there are time steps in the planning horizon. Constraint 3 ensures that every vertex visited by a vehicle is visited within the vehicle's active window. Constraint 4 ensures that the route is taken during its time window. Constraints 5-7 add constraints to the SOC of each vehicle. Constraint 5 ensures that the SOC of any vehicle at any time

remains within the predefined SOC limits. Constraint 6 is the initial SOC constraint, where each vehicle has a predefined starting SOC, s_{k_0}. Constraint 7 is the final SOC constraint, where each vehicle must end the planning period with an SOC value that is greater than or equal to s_{t_f}.

4.3 Solving the Scheduling Problem using an Ant Colony Approach

Now that the problem has been defined, the next step is to define a solution technique for the problem. The proposed solution technique is Ant Colony Optimization (ACO). ACO is a metaheuristic technique that is used to solve routing problems. It is based on the food-foraging behavior of ant colonies in nature. Ants venture out of their nest and explore their surrounding space in random directions for food, with each individual ant leaving a pheromone behind it for other ants to follow. In the event that an ant finds food, it brings the food back to the nest and leaves a strong pheromone to lead other ants to the food source. As more ants go to the food source, the pheromone grows stronger, creating a feedback loop. In the event that a food source is depleted, the ants gather to other food sources, and the pheromone evaporates.

Virtual ants in ant colony optimization explore the solution space for a good solution. These virtual ants generate solutions, which is the equivalent of finding food, and they leave pheromones on the edges of the graph of the problem. If better solutions are found, the pheromones increase the likelihood that future ants will select the same, or similar, paths. Ant colony approaches generally repeat until the quality of the solution stagnates or a predefined number of iterations have been completed. The contributions of this section include an ACO technique to solving the TOPSEV. The remainder of this section covers the notation of the ACO approach, the supporting algorithms for the ACO algorithm, and the ACO algorithm used to solve the TOPSEV.

4.3.1 Notation for the ACO Approach

Tables 4.1 and 4.2 include the notation used in the ACO scheduling process. In general, the l and u subscripts denote lower and upper, while c and n denote current and next.

Name	Description	Name	Description
Structures and Variables			
P	The probability matrix	c	A cost variable
C	The charging heuristic matrix	w_s	The start of an on-peak window
H	The pheromone matrix	w_e	The end of an on-peak window
Δ_H	This is a matrix which helps the update process of the pheromone matrix H	$\mathcal{W}$	The list of on-peak windows
		$\mathcal{R}$	The set of routes assigned to the vehicles
$\vec{w}$	The weights vector	$\mathcal{Q}$	The index list of integers which correspond to a specific entry in a vector
$\vec{t}$	The vector of "current times"		
$\vec{v}$	The vector of "current vertices"		
$\vec{s}$	The vector of "current SOCs"	$\mathcal{J}$	The starting indices for each vehicle
$\vec{a}$	The availability vector	$\mathcal{A}_r$	The list of available routes
t	A time variable	$\mathcal{P}_k$	The time-activity pairs of vehicle k
a	An activity variable	$\mathcal{R}_k$	The routes assigned to vehicle k
s	An SOC variable	$\mathcal{I}_k$	The index range of k
s_p	The projected SOC		
Functions			
$a(v)$	The activity of vertex v	$t(i)$	The time it takes to run route i
$t(v)$	The time of vertex v	$c(k)$	The battery capacity of vehicle k
$s_k(t)$	The SOC of vehicle k at time t	$e(i)$	The energy consumption of route i

Table 4.1: Structures, Variables, and Functions in the ACO Scheduling Process

Name	Description	Name	Description
Parameters			
m	The number of vehicles in the problem	$\tau_{l,k}$	The end of vehicle k's active window
n	The number of vertices in the problem	δ_t	The discrete timestep
n_r	The number of routes in the problem	δ_e	The change in energy
s_u	The SOC upper bound	r	The charging rate
s_l	The SOC lower bound	n_a	The number of ants
s_{k_0}	The starting SOC for vehicle k	N	The number of iterations to run the algorithm
s_{t_f}	The minimum final SOC value	α	The power on the pheromone matrix
t_0	The starting time	β	The power on the charging heuristic matrix
t_f	The final time	$p_{\min}$	The minimum value for the pheromone
$\tau_{e,i}$	The earliest time route i can be started	$p_{\max}$	The maximum value for the pheromone
$\tau_{l,i}$	The latest time route i must be completed	ρ	The forgetting factor
$\tau_{e,k}$	The beginning of vehicle k's active window		
Indices and Iterators			
i	An index generally depicting a route	k	An index depicting a vehicle
j	An index generally depicting a vertex	p	A generic iteration variable
Graph and Optimization Variables			
$\mathcal{G}$	The graph used in the problem	$\bar{\alpha}$	The off-peak pricing factor in terms of dollars per kilowatt timestep
$\mathcal{E}$	The edge set of the graph.		
$\mathcal{V}$	The vertex set of the graph, representing the destinations	$\bar{\beta}$	The on-peak pricing factor in terms of dollars per kilowatt timestep
x_{ij}	A decision variable, which equals one if the edge (i, j) is used and is zero otherwise		
Operators			
$\oplus$	An operator where $A \oplus b$ appends element b to (the end of) a set or list A	$\otimes$	An operator where $A \otimes B$ performs element-wise multiplication
$\ominus$	An operator where $A \ominus b$ removes element b from a set or list A		

Table 4.2: Parameters, Indices, Iterators, Optimization Variables and Operators used in the ACO Scheduling Process

4.3.2 Supporting Algorithms for the ACO Algorithm

The ACO approach uses supporting algorithms to create and analyze solutions. This section details the supporting algorithms for the ACO approach, mainly the schedule-generating algorithm and its supporting algorithms. The supporting algorithms for the ACO process include protocols for when route vertices and charging vertices are selected in the schedule-generating process, the schedule-generating process itself, and a cost calculation algorithm used for the ACO approach. The remainder of this section discusses each of these processes in detail.

Route Protocol

The schedule-generating process includes randomly selecting vertices, and depending on the type of vertex that is chosen, the algorithm should behave in a certain way. This section details two algorithms which are used when a route vertex is chosen, including a validity check and route protocol. The validity check in Algorithm 23 ensures that no time or energy constraints are violated. First, it checks that the time associated with the vertex is within both the active window of the chosen vehicle and the time window of the route. The validity check also ensures that the route can end within the time window and that the vehicle has sufficient SOC for the vehicle to complete the route.

The route protocol algorithm, Algorithm 24, uses Algorithm 23 to check the validity of the movement and adds the appropriate route indices to the vehicle's activity list. If the movement is not valid, the algorithm removes the probability of reselection and returns the current SOC and vertex for the vehicle.

The validity check is described in Algorithm 23. Algorithm 23 goes through a series of checks to see if adding a route i to a vehicle k's specified route violates any constraints. If any constraints are violated, the algorithm returns `false`. The first check is on line 2, which verifies that the next timestep, t, is within the vehicle's active window. The second check is on line 6, which verifies that the next timestep is at or after the beginning of the route's time window. The third check is on line 10, which ensures that there is enough time to complete the route by adding the route duration to the next time step and verifying that

the value is within the route's time window. The final check is to see if SOC constraints are violated. First, the projected SOC, s_p, is found on line 14 by subtracting the projected percentage drop from the current SOC, s. The percentage drop is calculated using the energy needed for the route, $e(i)$, in kWh, along with the battery capacity in kWh of the vehicle, $c(k)$. Line 15 ensures that the projected SOC is within the SOC bounds. If all tests pass, then the algorithm returns `true`.

Algorithm 23 `bool`← `route_valid`()

```
1:  // See if vehicle is active
2:  if τ_{e,k} ≤ t ≤ τ_{l,k} do
3:      return false
4:  end if
5:  // Check starting time constraint
6:  if t < τ_{e,i} do
7:      return false
8:  end if
9:  // Check route timing constraints
10: if not τ_{e,i} ≤ t + t(i) ≤ τ_{l,i} do
11:     return false
12: end if
13: // Check SOC constraints
14: s_p ← s − (e(i)/c(k))
15: if not s_l ≤ s_p ≤ s_u do
16:     return false
17: end if
18: return true
```

Given that the validity check is defined, the route protocol may also be defined. Algorithm 24 defines the protocol for what the schedule-generating algorithm should do when a route vertex is selected. If routes are available and the move is valid, then the vertices corresponding to the completion of the route are added to the schedule. Otherwise, no move is made, and the current time and SOC are returned from the algorithm.

The first part of the algorithm is the initialization step. Line 1 initializes the current time, t_c, and next time, t_n. Line 2 finds the number of available routes by adding the components of the availability vector, $\vec{a}$. If there are no available routes, then the probability

of re-selecting a route vertex is removed, and line 6 returns the current SOC and vertex. Line 8 initializes the set of available routes, and the route with the earliest starting time is chosen on line 9. Line 12 creates a subset of valid vertices for the vehicle.

After the initialization step, the validity check, algorithm 23, is run on line 13 and if the movement is valid, then the route is assigned to the vehicle. Line 15 adds the route's ID to vehicle k's route set. The route is then marked as unavailable on line 16. The vertices representing the route are then added to the schedule. Line 18 calculates the ending time for the route, and the **while** loop on line 19 adds route vertices to the schedule until the ending time for the route has been reached. Line 20 finds the next vertex associated with being on route, and shifts the indices to match the vertex set. Line 21 adds the vertex to vehicle k's schedule, and line 22 increments the time. Line 23 ensures that the chosen vertex cannot be revisited. After the **while** loop has run, a return-to-station vertex is selected, added, and marked as visited in a similar fashion. The SOC is updated on line 30 and the next SOC and vertex are returned on line 31. If the route is not valid, then the probability of selecting the same movement becomes zero and the current SOC and vertex are returned on line 34.

Algorithm 24 `route_protocol`()

$t_c \leftarrow t(v_c)$; $t_n \leftarrow t(v_n)$
$a \leftarrow \sum_{i=0}^{n_r} \vec{a}_i$
if not a **do**
    // Remove probability of re-choosing
    $P^*[v_c][v_n] \leftarrow 0$
    return s_c, v_c
end if
$\mathcal{A}_r \leftarrow \{r_i \in \mathcal{R} \mid \vec{a}_i \neq 0\}\ \forall i \in \{0, \ldots, n_r\}$
$i \leftarrow$ `argmin`$(\{\tau_{e,i} \mid i \in \{0, \ldots, n_r\}\})$
// Use only a subset of points
$(i_l, i_u) \leftarrow \mathcal{I}[k]$
$\mathcal{P}_k \leftarrow \{v \mid i_l \leq v \leq i_u\}$
if `route_valid`() **do**
    // Assign route
    $\mathcal{R}_k \oplus i$
    $\vec{a}[i] \leftarrow 0$
    // Add vertices from route
    $t_f \leftarrow t_c + t(i)$
    while $t_c < t_f$ **do**
        $v_n \leftarrow i_l + \mathcal{P}_k$`.index`$((t_c + \delta_t,$ `"On Route"`$))$
        $\mathcal{S}[k] \oplus v_n$
        $t_c \leftarrow t_c + \delta_t$
        $P^*[:, v_n] \leftarrow 0$
    end while
    // Return to station
    $v_n \leftarrow i_l + \mathcal{P}_k$`.index`$((t_c + \delta_t,$ `"In Station"`$))$
    $\mathcal{S}[k] \oplus v_n$
    $P^*[:, v_n] \leftarrow 0$
    // Update SOC
    $s_n \leftarrow s_c - e(i)/c(k)$
    return s_n, v_n
end if
$P^*[v_c][v_n] \leftarrow 0$
return s_c, v_c

Charging Protocol

Similarly to the route protocol, the schedule-generating algorithm should behave in a certain way when a charging vertex is chosen. This section details two algorithms which are used when a charging vertex is chosen, including a validity check and a charging protocol. The validity check in Algorithm 25 ensures that a charging session does not occur when the SOC is above the no-charge threshold or if the projected SOC increase of having the charging session goes outside of the predefined SOC bounds. Algorithm 25 also ensures that a one-charge-per-visit rule is observed.

The charging protocol in Algorithm 26 uses Algorithm 25 to ensure that the charging movement is valid. If charging is valid, then a charge session occurs. The vertex is added to the vehicle's activity list, and the probability matrix, current vertex, and SOC are all updated.

The validity check is described in Algorithm 25. Algorithm 25 runs a series of checks to see if a charging movement is valid. The first check is on line 2. If the `able_to_charge` boolean variable is not set, then the algorithm returns `false`. The final check is to ensure that a charge session does not violate the SOC bounds. The projected SOC is calculated on line 6 by adding the energy gained from charging in terms of the SOC of the battery. If the projected SOC is outside the bounds, then the algorithm returns `false`. If all checks pass, then line 10 returns `true`.

Algorithm 25 `bool`← `charging_valid`()

```
1:  // Observe one charge per visit
2:  if not able_to_charge do
3:      return false
4:  end if
5:  // Observe SOC thresholds
6:  s_p ← s_c + (δ_e/c(k))
7:  if not s_l ≤ s_p ≤ s_u do
8:      return false
9:  end if
10: return true
```

Given the validity check, the charging protocol can be defined in full. Algorithm 26 states the protocol for when a charging vertex is selected. If the movement is valid, then the vertex is added to the schedule on line 3, the probability matrix is updated on line 4, the SOC is updated on line 6, and the new SOC and vertex are returned on line 7. If the charging movement is not valid, then the probability matrix is updated on line 9 and the current SOC and vertex are returned on line 10.

Algorithm 26 `charging_protocol`()

```
1: if charging_valid() do
2:     // Start charging
3:     S[k] ⊕ v_n
4:     P*[:, v_n] ← 0
5:     // Update SOC
6:     s_n ← s_c + (δ_e/c(k))
7:     return s_n, v_n
8: end if
9: P*[v_c][v_n] ← 0
10: return s_c, v_c
```

Schedule Generation

Given the route and charging protocols, the schedule-generating algorithm can now be fully defined. The schedule-generating algorithm is divided into two stages: the initialization stage (Algorithm 27), and the assignment stage (Algorithm 28). Together, the two stages are analogous in operation to the route-generating algorithm, Algorithm 18 from Chapter 3, although structurally they are very different. The schedule generation process starts by finding the starting indices of the vehicles, and iteratively picks a vehicle and vertex, checks constraint violations, and if constraints are observed, updates the activity list for each vehicle in the schedule. After the probability of a movement is zero, the iterative process ends and the algorithm returns the schedule.

The first part of Algorithm 27 initializes the variables and finds the initial vertex set. The schedule is created on line 1, and the probability matrix is copied on line 2. The next step is to find the index of the starting vertex of each vehicle. The **for** loop which starts on

line 5 iterates through each of the vehicles in the problem to find the starting indices for each vehicle. Line 7 initializes a variable p which stores the starting index. The inner **for** loop that starts on 8 iterates through each time-activity pair to find the potential starting vertices. Line 9 extracts the time and activity pair. Line 10 finds the conditions of a vertex being a potential starting point, namely if the vertex represents if the vehicle being in the station at the start of the vehicle's active window and if the index is not already in $\mathcal{J}$. If the conditions are true, then the index corresponding to the vertex is assigned to p on line 11 and the algorithm breaks the loop. Once the loop is broken, the starting index is added to the starting indices list on line 15 and the schedule for vehicle k is started on line 16.

The next part of Algorithm 27 creates index ranges and initializes storage variables. The index ranges, $\mathcal{I}$, are initialized on line 18. The **for** loop on line 19 populates $\mathcal{I}$. Line 20 adds the starting and ending vertex indices for each vehicle k and line 22 adds the starting and ending vertex indices for the last vehicle. Line 24 makes a copy of the starting indices to track the current indices. Line 25 initializes a variable to track the current time of each vehicle. Likewise, line 26 initializes a variable to track the current SOC of each vehicle. The `able_to_charge` variable, which tracks each vehicle's ability to charge, is initialized on line 27, and the `route_availability`, which tracks each route's availability, is initialized on line 28.

Algorithm 27 `init_schedule`

1: $\mathcal{S} \leftarrow []$ // Initialize the schedules
2: $P^* \leftarrow P$ // Copy the probability matrix
3: // Find the starting indices and start schedules
4: $\mathcal{J} \leftarrow []$ // Initialize starting indices variable
5: **for** $k = 0, \ldots, m$ **do** // Iterate through vehicles
6: // Find the starting point of each vehicle
7: $p \leftarrow 0$ // Store potential starting index
8: **for** $j = 0, \ldots, n$ **do** // Iterate through time-activity pairs
9: $(t_j, a_j) \leftarrow \mathcal{P}[j]$
10: **if** $a_j =$ "In Station" **and** $t_j = t_0^k$ **and** $j \notin \mathcal{J}$ **do**
11: $p \leftarrow j$ // Add potential vertex
12: **break**
13: **end if**
14: **end for**
15: $\mathcal{J} \oplus p$ // Add starting index to starting indices
16: $\mathcal{S} \oplus [p]$ // Start schedule for vehicle k
17: **end for**
18: $\mathcal{I} \leftarrow []$ // Define index ranges for each vehicle
19: **for** $i = 0, \ldots, m-2$ **do** // Populate $\mathcal{I}$
20: $\mathcal{I} \oplus (\mathcal{J}[i], \mathcal{J}[i+1] - 1)$
21: **end for**
22: $\mathcal{I} \oplus (\mathcal{J}[|\mathcal{J}| - 1] + 1, n - 1)$
23: // Initialize storage variables
24: $\vec{v} \leftarrow \mathcal{J}$ // Current vertices
25: $\vec{t} \leftarrow [\tau_{e,k}]\ \forall k$ // Current times
26: $\vec{s} \leftarrow [s_{k_0}]\ \forall k$ // Current SOCs
27: `able_to_charge` $\leftarrow$ [`true`] $\forall k$
28: `route_availability` $\leftarrow [1]\ \forall i \in \{0, \ldots, n_r\}$

The schedule generator in Algorithm 28 creates schedules using the route and charging protocols in an iterative process. After the initialization step on line 1, and initializing an index set to choose vertices at random, the iterative process begins. The iterative process contained in the **while** loop on line 3 starts by selecting the vehicle k with the earliest current time. After the vehicle is selected, the index range for the vehicle is extracted on line 5. Line 6 creates the weights for the next movement. If the weights are all zero, then the assignment process is over and the **while** loop breaks. Otherwise, a random vertex, j is selected from the available indices given the weights using the `random_choice` function on line 10. After the vertex is selected, its time and activity are extracted on line 11. If j

is not in the vehicle's index range and also is not a charging vertex, then the movement is rejected and the loop restarts.

The next part of the schedule assignment process is the use of protocols for the different types of vertices. If the activity of j is "On Route," then the route protocol, Algorithm 24 updates the SOC and the current vertex of vehicle k on line 17. After running the route protocol, line 18 marks vehicle k as able to charge. If there are no available routes, then the probability of selecting an "On Route" vertex becomes zero using the **for** loop on line 20. If j is a "Charging" vertex, then the charging protocol is used to update the SOC and the current vertex of vehicle k using Algorithm 26 on line 28. If the activity of j is "In Station," then a small station protocol is enacted. First, if the vehicle is currently charging and there are still routes available, then the vehicle is not allowed to start another charging session. Vertex j is added to the schedule for vehicle k on line 33, marked as visited on line 34, and updated as the current vertex on line 35. The time is updated on line 37.

The final part of the schedule assignment process is to check for constraint violations. The number of available routes is counted and added to a `violations` variable. The current SOCs of each vehicle are compared against the final, minimum-SOC constraint. If the SOC of the vehicle is below the constraint, then the number of violations increases by one. The schedule and violations are returned on line 45.

Algorithm 28 $\mathcal{S} \leftarrow$`generate_schedules`

`init_schedule`() // Initialize schedule using Algorithm 27
$\mathcal{Q} \leftarrow \{0, \ldots, n-1\}$
while `true` **do**
    $k \leftarrow$ `argmin`$(\vec{t})$ // Prioritize earliest vehicle
    $(i_l, i_u) \leftarrow \mathcal{I}[k]$ // Extract index range for vehicle
    $\vec{w} \leftarrow P[\vec{v}[k]]$ // Create weights for next movement
    if $\vec{w} = 0$ **do**
        break // Assignment process is over
    end if
    $j \leftarrow$ `random_choice`$(\mathcal{Q}, \vec{w})$ // Select next vertex j
    $(t_j, a_j) \leftarrow \mathcal{P}[j]$ // Extract time and activity of vertex j
    if not $i_l \leq j \leq i_u$ **and** $a_j \neq$"Charging" **do**
        $P^*[\vec{v}[k], j] \leftarrow 0$ // Eliminate probability of reselection
        continue
    end if
    if $a_j =$ "On Route" **do**
        $\vec{s}[k], \vec{v}[k] \leftarrow$ `route_protocol`() // Run route protocol (Algorithm 24)
        `available_to_charge`$[k] \leftarrow$ `true` // Mark vehicle k as able to charge
        if `sum(routes_available)` $= 0$ **do**
            for $i = 0, \ldots, |\mathcal{P}|$ **do**
                $(t_i, a_i) \leftarrow \mathcal{P}[i]$
                if $a_i =$ "On Route" **do**
                    $P[:, i] \leftarrow 0$ // Remove probability of selecting routes
                end if
            end for
        end if
    else if $a_j =$ "Charging" **do**
        $\vec{s}[k], \vec{v}[k] \leftarrow$ `charging_protocol` // Run charging protocol (Algorithm 26
    else do
        if $a(\vec{v}[k]) =$"Charging" and `sum(route_availability)` > 0 **do**
            `able_to_charge`$[k] \leftarrow$ `false` // Mark vehicle as unable to charge
        end if
        $\mathcal{S}_k \oplus j$ // Add vertex to schedule
        $P[:, j] \leftarrow 0$ // Mark as visited
        $\vec{v}[k] \leftarrow j$ // Update current vertex
    end if
    $\vec{t}[k] \leftarrow t(\vec{v}[k])$
end while
`violations` $\leftarrow$ `sum(route_availability)` // Run a validation check
for $k = 0, \ldots, m-1$ **do**
    if $\vec{s}[k] < s_{t_f}$ **do**
        `violations` $\leftarrow$ `violations` $+ 1$
    end if
end for
return $\mathcal{S}$, `violations`

Cost Calculation

In order to compare different solutions generated by Algorithm 28, it is necessary to calculate the cost of each solution. Algorithm 29 computes the cost of a solution by evaluating the objective function in Equation 4.1

The cost calculation algorithm, Algorithm 29, calculates the cost of a given schedule and returns infinity if there are any violations. Line 1 sees if there are any violations. If any constraint has been violated, then an infinite cost is returned from the algorithm. Otherwise, variables that store the amount of time spent charging on peak and off peak are initialized on lines 4 and 5, respectively. The algorithm iterates through the schedule of each vehicle in the overall schedule as well as each vertex in the schedule using the **for** loops on lines 6 and 8, respectively. If a vertex represents a charging vertex, then the **for** loop on line 11 iterates through the peak time windows to determine whether the charging vertex occurs within the time window or not. If the vertex is within the peak time window, then the on-peak time is increased by the time step on line 14. Otherwise, the off-peak time increases on line 16. The total cost is calculated using the following equation:

$$c = r(\bar{\alpha} \cdot t_{\text{offpk}} + \bar{\beta} \cdot t_{\text{onpk}}) \tag{4.2}$$

where r is the charging rate, $\bar{\alpha}$ is the off-peak cost in dollars per kilowatt-timestep, t_{offpk} is the time spent charging during off-peak hours, in timesteps, $\bar{\beta}$ is the on-peak cost in dollars per kilowatt-timestep, and t_{onpk} is the time spent charging during on-peak hours, in timesteps. This cost is returned on line 23.

Algorithm 29 `calculate_cost`

1: **if** `violations`
2: **return** ∞
3: **end if**
4: $t_{\text{onpk}} \leftarrow 0$
5: $t_{\text{offpk}} \leftarrow 0$
6: **for** $k = 0, \ldots, |\mathcal{S}|$ **do**
7: $\mathcal{S}_k \leftarrow \mathcal{S}[k]$
8: **for** $j = 0, \ldots, |\mathcal{S}_k|$ **do**
9: $(t_j, a_j) \leftarrow \mathcal{P}[j]$
10: **if** $a_j =$ "Charging" **do**
11: **for** $i = 0, \ldots, |\mathcal{W}|$ **do**
12: $(w_s, w_e) \leftarrow \mathcal{W}[i]$
13: **if** $w_s \leq t_j \leq w_e$ **do**
14: $t_{\text{onpk}} \leftarrow t_{\text{onpk}} + \delta_t$ // Increase on-peak time
15: **else do**
16: $t_{\text{offpk}} \leftarrow t_{\text{offpk}} + \delta_t$ // Increase off-peak time
17: **end if**
18: **end for**
19: **end if**
20: **end for**
21: **end for**
22: $c \leftarrow r(\bar{\alpha} \cdot t_{\text{offpk}} + \bar{\beta} \cdot t_{\text{onpk}})$ // Calculate cost using Equation 4.2
23: **return** c

4.3.3 The Ant Colony Optimization Algorithm

Given the schedule generation process as well as a way to calculate the operating cost, the ACO approach can be fully defined. The Ant Colony Optimization Algorithm, Algorithm 30, is very similar to the ACO algorithm in 22. The first part of the algorithm is the initialization step. First, the probability matrix is initialized on line 2 to be the charging heuristic matrix C, where c_{ij} is defined to be either 10, 0.1, 1, or 0. If j represents a charging vertex during off-peak hours or if j represents a route vertex during the route completion window, then c_{ij} is 10. If j represents a charging vertex during on-peak hours or if j represents a route vertex outside the route completion window, then c_{ij} is 0.1. If j represents a station vertex, then c_{ij} is 1, otherwise $c_{ij} = 0$. After the probability matrix is created, the pheromone matrix is initialized on 3. Using Algorithm 28, line 4 creates an initial solution. Line 5 calculates the cost of the initial solution using Algorithm 29.

After the initialization step, the iterative process creates subsequent schedules and updates the pheromones. This process is contained in the **for** loop that starts on line 6. Line 7 updates the pheromone matrix in a way similar to Equation 3.5, with the main difference being the use of a different heuristic matrix. Line 8 initializes the pheromone update matrix as zeros. The iterative process of ant simulation is contained in the **for** loop which starts on line 9. For each ant, a new schedule is created using Algorithm 28 on line 10 and the cost of the new schedule is calculated using Algorithm 29 on line 11. The cost of the new solution is compared against the lowest cost on line 12. If the cost is lower, then the current solution replaces the old solution on line 13. The pheromone update is then calculated using Algorithm 19 on line 15 and the inner **for** loop ends. Line 17 calculates the pheromone update using the best ant's route using Algorithm 19. Line 18 updates the pheromones using Equation 3.4. The pheromones are then saturated on line 19. After the outer **for** loop is complete, the algorithm returns the best-performing schedule on line 21.

Algorithm 30 $\mathcal{S}_{\text{best}} \leftarrow$ aco()

```
// Initialize ACO variables
P ∈ ℝ^{n×n} ← C // Initialize probability matrix P
H ∈ ℝ^{n×n} ← [p_max] // Initialize pheromone matrix H
// Create initial solution using Algorithm 28
S_best ← generate_schedules()
c_min ← calculate_cost(E, S_best) // Get the cost via Algorithm 29
for i = 0, ..., N do
    P ← [H]^α ⊗ [C]^β // Update the probabilities via Equation 3.5
    Δ ∈ ℝ^{n×n} ← [0] // Initialize pheromone update matrix as zeros
    for j = 0, ..., n_a do // Run the ants
        // Create new schedules using Algorithm 28
        S ← generate_schedules()
        c ← calculate_cost() // Get the cost via Algorithm 29
        if c < c_min do // If the solution is better, record it
            S_best ← S; c_min ← c
        end if
        // Calculate the pheromone update given the ant's route via Algorithm 19
        Δ ← update_del_pheromone(c, Δ, S)
    end for
    // Calculate the pheromone update given the ant's route via Algorithm 19
    Δ ← update_del_pheromone(c_min, p_max, Δ, S_best)
    H ← (1 − ρ)H + Δ // Calculate Pheromones with Equation 3.4
    // Saturate the pheromones to minimum and maximum allowed values
    H ← min(H, p_max); H ← max(H, p_min)
end for
return S_best
```

4.4 The Bus Model for Fixed Schedules

Now that the ACO scheduling process has been defined, the focus shifts to modeling the vehicles that are being scheduled. The model for open- and flexible-schedule vehicles is of a Ford E-Transit and is defined in Section 3.6. The model for fixed-schedule vehicles is that of a battery-electric bus. Since fixed-schedule vehicles are modeled as buses, the construction of an electric bus charging and discharging model was necessary. This section outlines the techniques and results of creating a bus model to be used in a scheduling paradigm.

Given that the discharging rate depends on the route the bus travels and the charging rate depends on the charger that charges the bus, it was necessary to collect and analyze both the raw state-of-charge (SOC) data and the GPS data of the buses. It was not known

which bus was assigned to which route on each day, so it was necessary to examine the two datasets to separate the SOC data by route.

The process of collecting and filtering data includes multiple steps. Data for each day was collected and stored over a period of several months. The GPS and SOC data was then matched to find the location of the bus when it was charging and discharging, and, using that information, the charge and discharge rates were calculated. Figure 4.4 shows the process used to model the charging and discharging of the buses. The remainder of this section includes the methods used to process the SOC data, the methods used to process the GPS data, the method used to match the two datasets, and the results of the battery model.

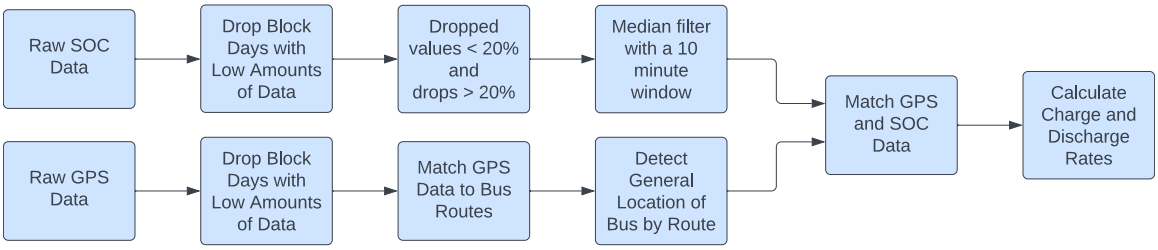


Fig. 4.4: Bus Charge and Discharge Modeling Process

4.4.1 Processing the SOC Data

In order to create a charging and discharging model for a battery-electric bus, the SOC data was recorded and processed. SOC data of the buses in Utah Transit Authority's electric bus fleets in Ogden and Salt Lake City was recorded over a period of several months. The amount of recorded data varied widely by day, and it became an area of interest to see how many block-days[1] had low amounts of SOC samples. The block-days that had low amounts of SOC samples would significantly skew the model towards lower charging and discharging rates due to interpolation. It was decided that around 720 samples (1 sample for every 2 minutes, on average) would be the cutoff for inclusion in calculating charge and discharge rates.

[1] A block-day is defined to be an instance of a bus on its assigned route on a given day

Further filtering of the SOC data included dropping points that were not valid. Utah Transit Authority (UTA) stated that they do not let the SOC of any bus drop below 20%, but there were several instances in the data where the SOC of a bus would drop to 0%, or another very low value, and then immediately go back to its previous level. Other such occurrences would occur with higher levels of SOC, and therefore, for drops more than 20% the rogue data point would not be considered in the calculation.

Given that the SOC data itself was very discretized and also very noisy, a median filter was used to smooth out the data and paint a more realistic picture. It was found that a ten-minute window smoothed out major anomalies. After these processing steps, the SOC data was then interpolated to match the time stamps of the GPS data so that the two sets could be matched together.

4.4.2 Processing the GPS Data

Along with the SOC data, GPS data of the buses in UTA's electric bus fleets in Ogden and Salt Lake City was recorded over the same period of several months. Similarly to the SOC data, 720 samples (1 sample for every 2 minutes, on average) was the number of samples needed to use the GPS data for a given block-day in the model.

The GPS data was matched to routes by hand, where it was placed into a KML file and visually matched with a route description from either a KML file provided by UTA or by verifying that the route follows the stops specified on UTA's website. After separating the block-days by route, each block-day was examined to detect the general location of the bus on a given route. This helped to find whether the bus was in the depot, at the station, at the terminal of the route, or on the route. Figure 4.5 shows the implementation details for detecting the general location of the bus.

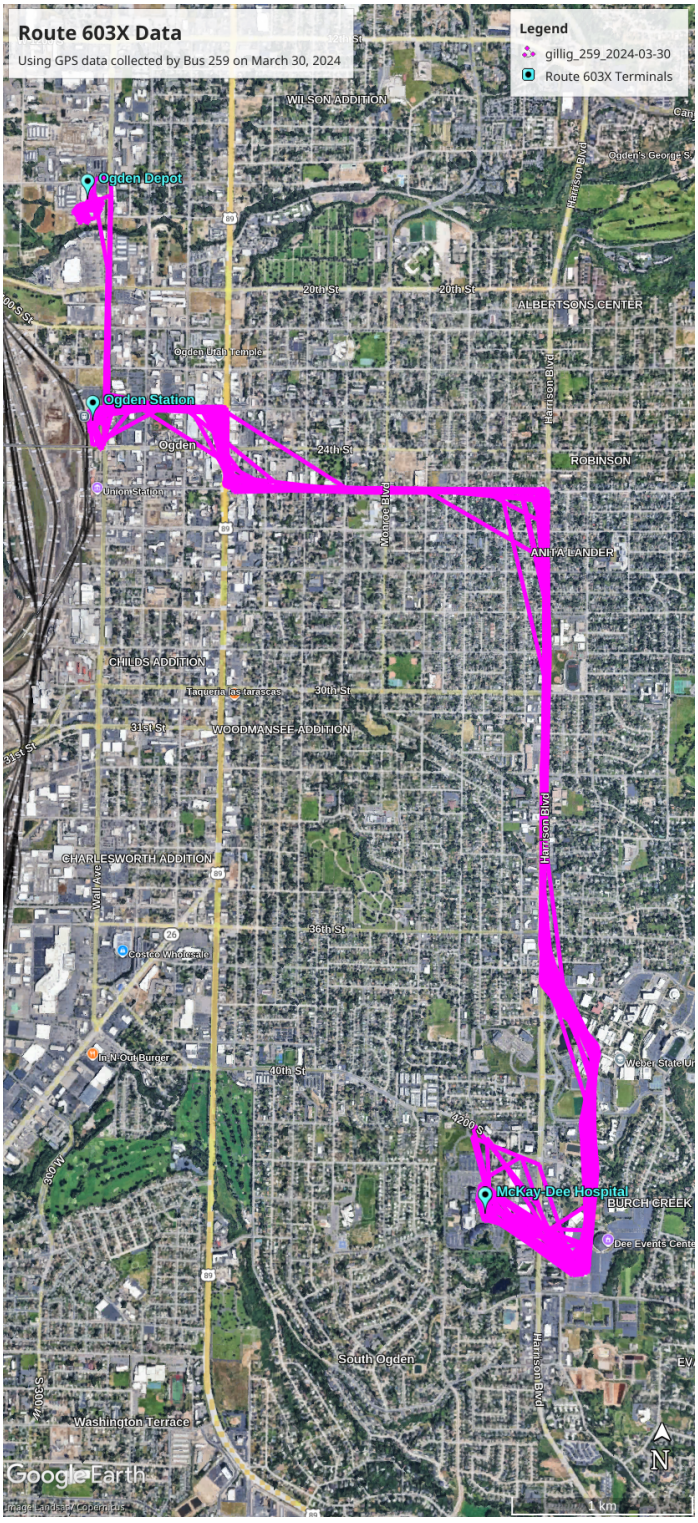


Fig. 4.5: If the bus is outside a radius of 125 yards (for terminals) or 250 yards (for a station or depot), then the bus was assumed to be on route. The stray lines are from interpolation of location data.

4.4.3 Matching GPS and SOC Data

Using the general location of the bus along with the interpolated SOC values, the two datasets were used to find the charging and discharging rates of the buses. The outbound discharge rate for each route was calculated by finding the average change of SOC as a bus traveled from the station to the route terminal. The inbound discharge rate was calculated by finding the average change in SOC on the return trip. These values were added together to find the overall rate from a single run.

Figure 4.6 shows a visual representation of the mapping of GPS values to SOC values. It shows the SOC and GPS data of a bus in Salt Lake City over a period of 24 hours. The SOC is represented by green dots (A), and the GPS data is visualized in the background as vertical bars. The darker color (B) shows that the bus is in the station at the given time. The lighter color (C) shows the times the bus was at the route's terminal; in this case, the terminal was in West Valley City, Utah, and had the code WVCENTRAX. The thick, white bars (D) indicate that the bus was on route during those times. The thin, white bars (E) indicate noise and data loss.

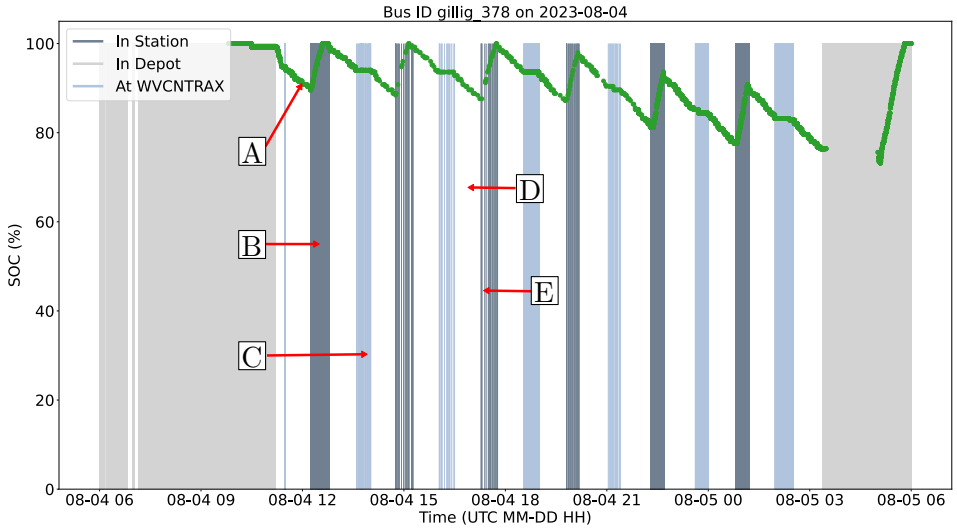


Fig. 4.6: A Visual Representation of Mapping GPS Values to SOC Values

4.4.4 Results and Implementation

The charge and discharge models of the bus were created from processing the SOC and GPS data. The charge and discharge rates for the Ogden 603X route are shown in Table 4.3. The outbound and inbound discharge rates were used as the energy costs in the Ant Colony simulation. A histogram showing discharge rate data for Ogden route 603X is shown in Figure 4.7.

Metric	Inbound Rate	Outbound Rate	Overall Rate	Inbound Discharge	Outbound Discharge
Average	-14.36	-26.54	-19.89	-7.55	-13.23
Median	-14.63	-27.75	-16.70	-8.00	-13.54
Standard Deviation	5.96	13.37	11.71	3.04	6.46

Table 4.3: The Discharge Rates and Bus Discharge for Ogden Route 603X

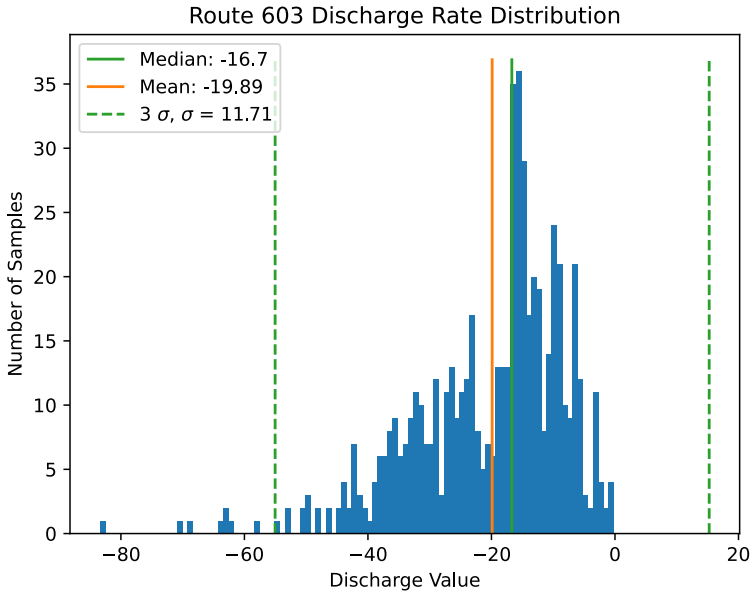


Fig. 4.7: A Histogram Showing Discharge Rate Data for Ogden Route 603X

4.5 Method Comparison Against Exact Approaches

Given the novel approach from Section 4.3, the Ford E-Transit model from Section 3.6, and the bus model from Section 4.4, a comparison between the novel approach and existing approaches can be made. The two approaches used are the bin-packing approach from [42] for flexible scheduling, the network-flow approach from [41] for fixed scheduling, and the combined fixed-and-flexible scheduling approach from [2] for a combined approach. Each optimization technique was implemented in Python and run on an Intel(R) Core(TM) i7-10700 CPU with 32 GB of RAM. The remainder of this section includes background information on each approach and the results of comparing the novel ACO approach against the two exact approaches.

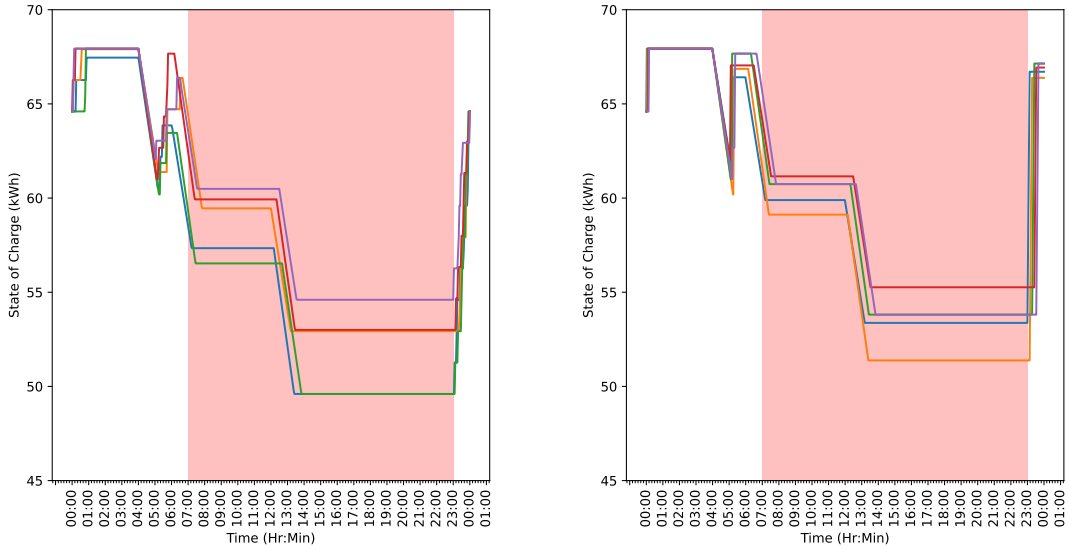
4.5.1 Bin Packing and Flexible Scheduling

Bin-packing problems are a family of combinatorial optimization problems that involve placing items into containers, called bins. The items and bins of the bin-packing approach in [42] are the activities and schedules of a bus fleet. This subsection details the bin-packing approach to scheduling electric buses, as well as a comparison between the ACO and bin-packing approaches.

The bin-packing approach in [42] minimizes the cost of charging an electric bus fleet. It integrates time-of-use and demand costs into one objective function and takes uncontrolled loads into account. The authors of [42] formulate a mixed-integer, linear program to solve the flexible-schedule problem. The variables of optimization include the times that the charging session starts and ends (both continuous variables), as well as a continuous SOC variable, discrete charging session starts and ends, uncontrolled loads, the average power drawn from charging the buses, and a slack variable that prevents two buses from being assigned to the same charger at the same time. This bin-packing approach was used without demand costs as well as no uncontrolled loads for its comparison with the ACO implementation.

Figure 4.8 shows the state of charge for each of the five vehicles throughout the day using the two scheduling methods. Note that the SOC values for the buses in the ACO

approach tend to be higher than those of the bin-packing approach. Furthermore, note that both approaches chose to run routes around the same time given the suggested start time for each route.



(a) Vehicle SOC from Bin Packing Approach

(b) Vehicle SOC from ACO Approach

Fig. 4.8: A Visual Representation of Vehicle SOC from schedules generated by Bin Packing and ACO aproaches. The ACO tends to charge the vehicles to higher levels, resulting in higher costs. The peak windows are shown in red.

Table 4.8 shows the comparison between the two solution approaches. There is an 11.33% gap between the ACO approach and the bin-packing approach. The ACO also suffers from the curse of dimensionality, as more vehicles significantly increase the number of vertices in the graph. Increased graph vertices directly increase the amount of memory needed to store the graph. Given that the bin-packing approach uses continuous time, its peak computational load is much less than that of the ant colony. In terms of time complexity, the ant colony provides a solution much faster than the bin-packing approach. This is due to the heuristics used in the technique. In general, the ACO technique is effective in providing a good solution in a small amount of time at the expense of using more memory.

	Bin Packing	Ant Colony
Solution Cost ($)	3.02	3.36
Peak Computational Load (GB)	6.784	15.671
Time Complexity (s)	270.564	45.488

Table 4.4: A Comparison between Bin Packing and ACO approaches

4.5.2 Network Flow and Fixed Scheduling

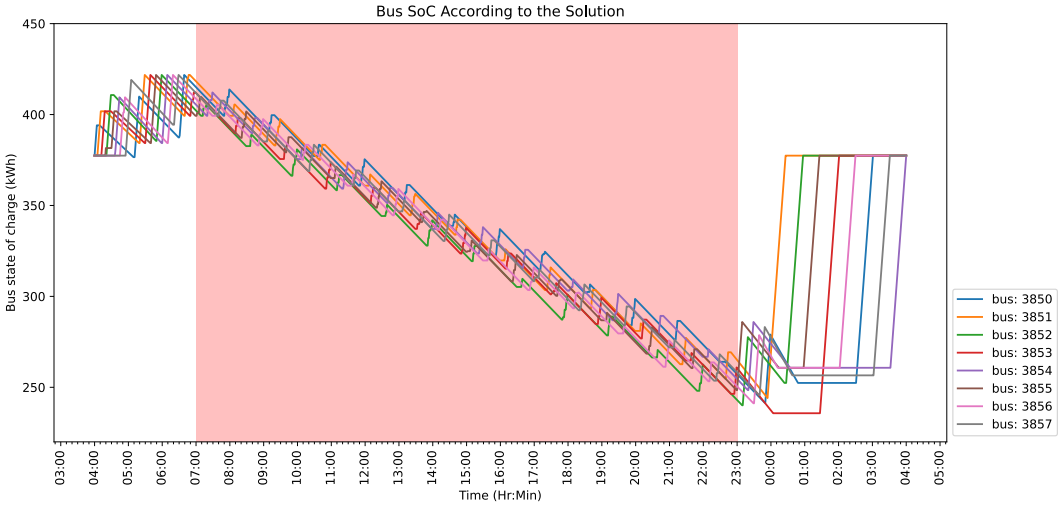
In a network-flow problem, a network is a directed graph with edges that have some predefined maximum edge weight, or *capacity*. The edges of the graph also have a flow value, which shows how much flow is passing through an edge. The vertices of the graph include two special types of nodes: source nodes and sink nodes. The source nodes generate the flow and the sink nodes capture the flow. All other nodes are intermediary points that must have the same amount of flow entering and exiting the node. The network-flow graph in [41] models the action space of the chargers in an electric bus station. This subsection details the network-flow approach to scheduling electric buses as well as a comparison between the ACO and network-flow approaches.

The network-flow approach in [41] provides a Mixed-Integer, Linear Program (MILP) with several bus-charging attributes. The MILP accounts for a nonlinear charging profile, partial charge modeling, charger availability considerations, multiple charger types, and time-of-use costs. In this work, the nonlinear charging profile is replaced by its linear counterpart.

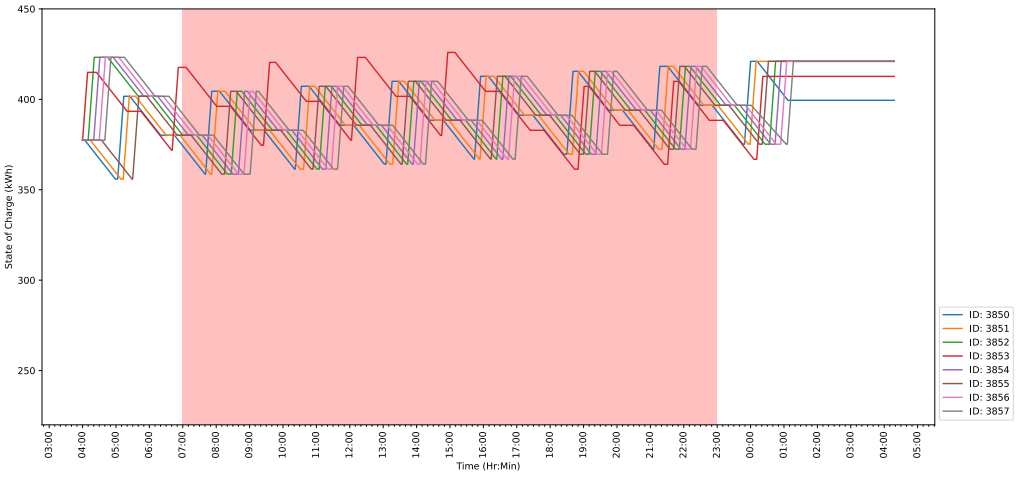
The network-flow approach was used to generate charging schedules of Utah Transit Authority's electric bus fleet in Ogden, Utah, using the model from Section 4.4. Figure 4.9a shows the projections for the SOC of each bus that runs route 603X in Ogden, given that the buses follow the schedule.

The ACO approach used the same schedule and model as the network-flow approach, but initially struggled to find a feasible solution. The charging heuristic matrix was updated to be the adjacency matrix of the graph, and a realistic heuristic of having ten-minute

charging sessions was implemented in order to create a feasible solution. The struggle with feasibility originates from the use of soft constraints to comply with the terminal SOC constraint. Figure 4.9b shows the projections for the SOC of each bus given the ACO-generated schedule.



(a) Results of the Network Flow Scheduler



(b) Results of the ACO Scheduler

Fig. 4.9: A Visual Representation of Bus SOCs for Ogden Route 603X Given Schedules Generated by Network Flow and ACO Approaches. The on-peak window is shown in red.

Table 4.5 shows the comparison between the two solution approaches. Again, the ACO suffers from the curse of dimensionality, as more vehicles significantly increase the number of vertices in the graph. Increased graph vertices directly increase the amount of memory needed to store the graph. In terms of time complexity, the ant colony provides a solution much faster than the network-flow approach. This is due to the heuristics used in the technique. In this case, the ACO technique is not as effective in providing a good solution compared to the flexible-schedule approach, although it generates a solution in a small amount of time at the expense of using more memory.

	Network Flow	Ant Colony
Solution Cost ($)	105.06	149.57
Peak Computational Load (GB)	0.141	26.842
Time Complexity (s)	302.096	77.357

Table 4.5: A Comparison between Network Flow and ACO approaches

4.6 Memory Usage of the Ant Colony Optimization Process

As one of the pitfalls of the ACO scheduling algorithm is the memory usage of the algorithm, it is important to consider which aspects of the algorithm caused such memory usage. Further study into the memory usage of the algorithm concluded that the main contributor to the intense memory usage was in the graph creation process. Given that there is a vertex representing each minute of every possible activity for each vehicle, the adjacency matrix of the graph containing all possible activities in the time horizon for all of the vehicles can be massive. Thus, a pruning and merging process occurs where the vertices representing charging sessions are combined and the graph rewired. After the pruning and merging process, the graph has a much smaller size. Figure 4.10 shows a visual representation of a graph of two vehicles before and after the rewiring process.

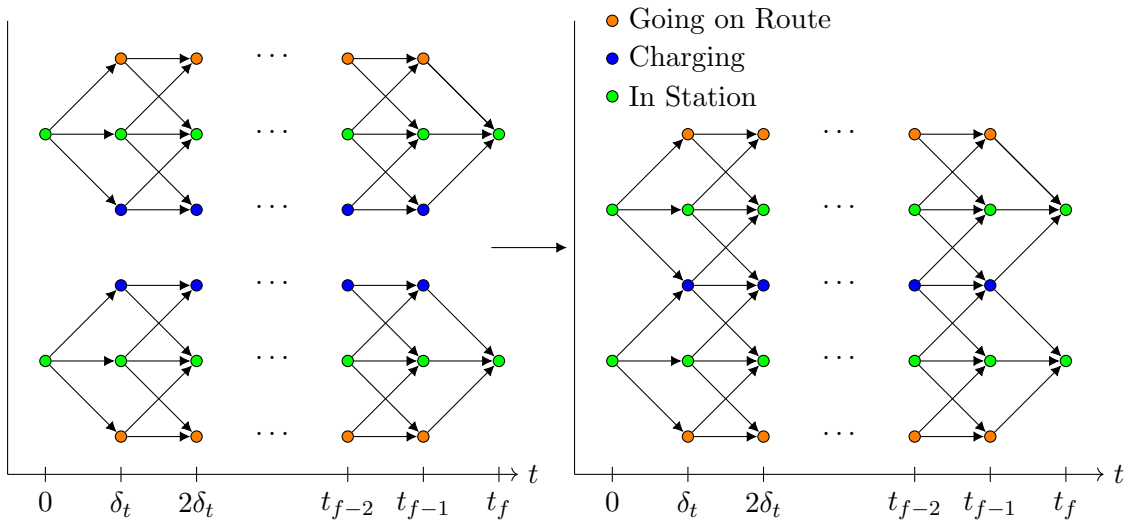


Fig. 4.10: A Visual Representation of a Graph of Two Vehicles before and after the Rewiring Process

4.7 Results Using the Bus Model and EVRPE for Heterogeneous Scheduling

This section is the culmination of every previous section in this work. In this section, a scheduling scenario was created using open, flexible, and fixed scheduling constraints using both the EVPRE model and the bus model. The solution of this scenario is referred to as the *full schedule* and uses two stages to solve the problem. Figure 4.11 shows the flow of data in creating the full schedule. First, previously-selected coordinates of trivial locations in the Ogden area are fed into the EVPRE (from Section 3.6.1) and the routing ACO (Section 3.3) to create flexible-schedule routes from an open-schedule problem. Next, the bus model (Section 4.4) and the bus schedule are fed into the ACO implementation for fixed schedules, referred to as the *fixed scheduler* (Section 4.5.2). The full scheduler then uses the information from the fixed scheduler and combines the newly-created flexible routes with previously-generated flexible routes from section 3.6 to create the full schedule.

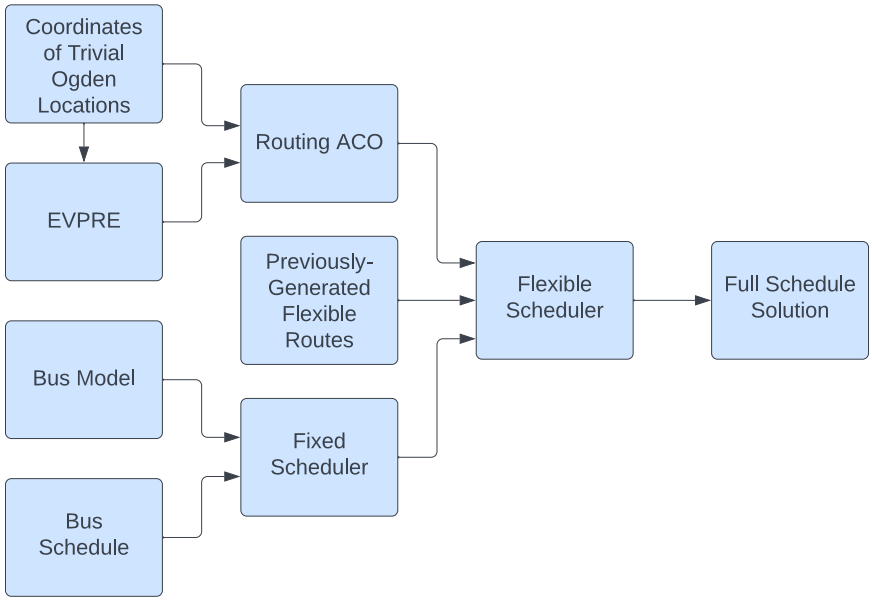


Fig. 4.11: Overall Process of the Full Schedule

The two scheduling stages, namely the fixed scheduler and the flexible scheduler, exist for three main reasons. The first reason is that, given the memory-intensive nature of the ACO scheduling algorithm, the algorithm can handle more complex problems when split into two stages. The second reason is that the heuristics for the fixed- and flexible-schedule paradigms are different, with the heuristic differences mainly revolving around the charging heuristic matrix. The third reason is that the fixed schedules and flexible schedules use different vehicle models and chargers.

Given that the charging heuristic matrices are different between the two ACO approaches, information from the fixed scheduler can be fed into the flexible scheduler through the use of the charging heuristic matrix. A list of times in which fixed vehicles are charging is fed into the flexible scheduler, which in turn uses that list of times to decrease the probability of a flexible or open vehicle from charging during those times. This is done by weighting the charging heuristic matrix so that the times at which fixed-schedule vehicles are being charged are set to zero.

Figures 4.12 and 4.13 show the results of the full schedule in terms of the SOC of each vehicle. The fixed schedules are the same schedules used in Section 4.5.2 and are

shown on the top. The flexible schedules are shown below using the flexible schedules from Section 3.20 and three additional newly-generated open schedules. The newly-generated open schedules use a 20% constraint on the battery and three vehicles, yielding larger routes. The open-schedule routes are also run four times throughout the day, while the flexible-schedule routes are run three times during the day.

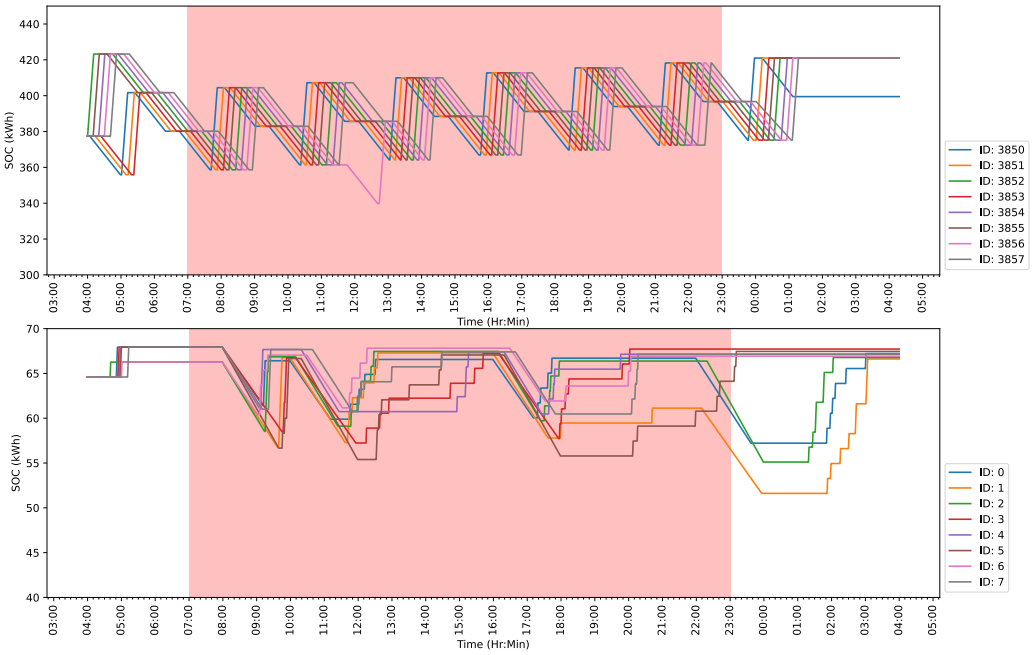


Fig. 4.12: Results of the ACO Scheduler with Heuristic Considerations. The open- and flexible-schedule vehicles (below) tend to charge when the fixed-schedule vehicles (above) are not charging.

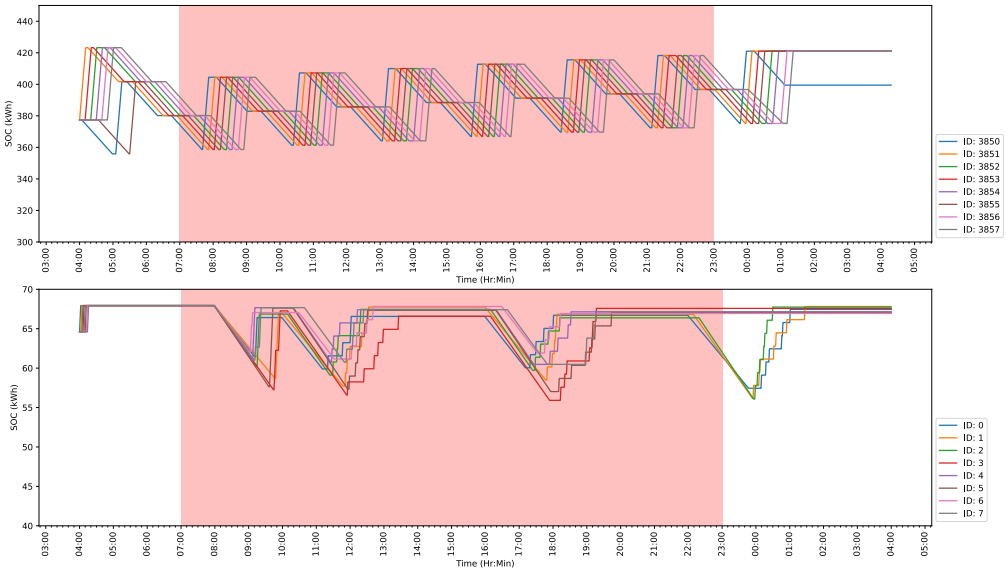


Fig. 4.13: Results of the ACO Scheduler without Heuristic Considerations. Given no heuristic considerations, the schedulers are disjoint and the two vehicle types tend to charge at the same time.

Given that the operational costs depend on the peak power draw in addition to the on-peak and off-peak charging sessions [3], and given that there are multiple chargers in this scenario, it is important to see how much average power the chargers draw over a period of 15 minutes. Figures 4.14 and 4.15 show the interpolated average power over 15 minutes, with the average plotted at the end of each 15-minute window, for using and not using heuristic considerations, respectively. Note that without heuristic considerations, the peak power is higher than that using the heuristic considerations.

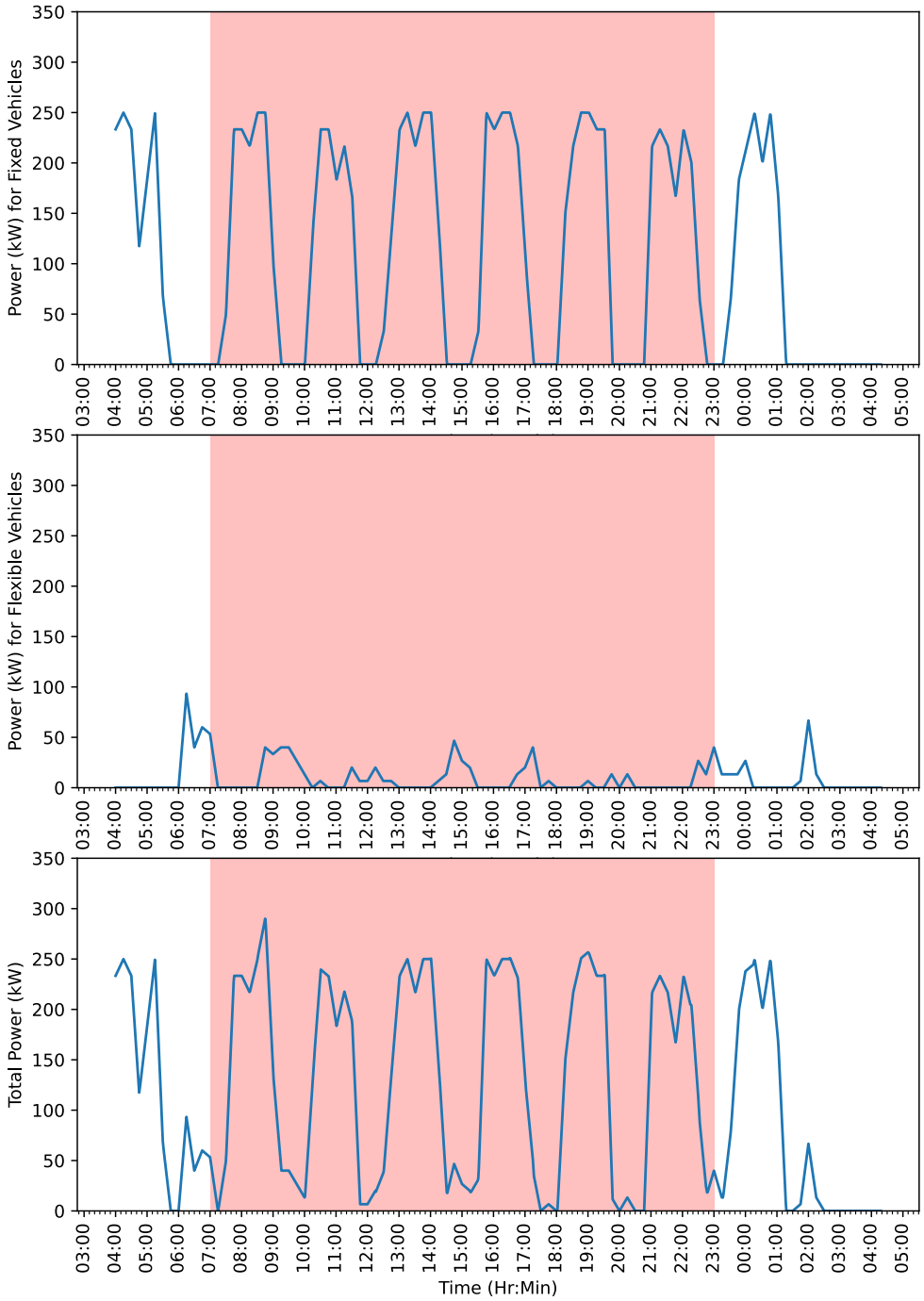


Fig. 4.14: The peak average power draw is around 250 kW using the schedule with heuristic considerations. The power draw of the bus fleet is shown at the top, the power draw from the E-Transit fleet is shown in the middle, and the power draw of the full heterogeneous fleet is shown at the bottom.

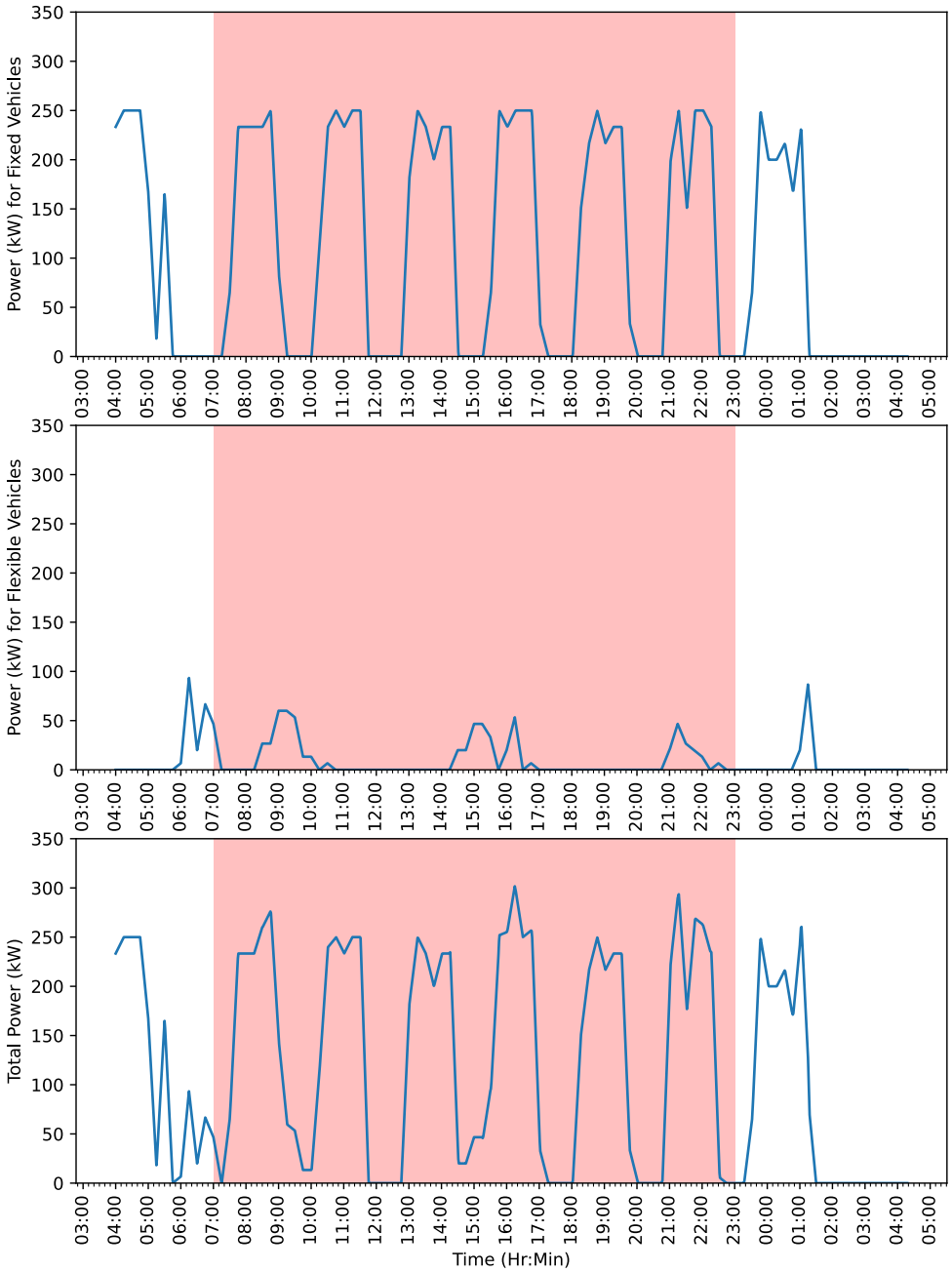


Fig. 4.15: The peak average power draw is around 325 kW using the schedule without heuristic considerations. The power draw of the bus fleet is shown at the top, the power draw from the E-Transit fleet is shown in the middle, and the power draw of the full heterogeneous fleet is shown at the bottom.

4.8 Conclusion

The scheduling problems studied in this chapter include flexible scheduling, fixed scheduling, and heterogeneous scheduling. This chapter includes the graph structure used in the scheduling process, the method of creating the bus model used for fixed schedules, comparisons against exact solvers, and a presentation of the full schedule. The ACO creates good solutions in a relatively short amount of time at the expense of using more memory than the exact approaches. This chapter concludes with an explanation of the process of creating the full schedule and the results of full schedule creation.

CHAPTER 5

CONCLUSION

The contributions and accomplishments of this thesis are two-fold and pertain to routing and scheduling problems. The routing problem in this work is used to create energy-aware routes that can be used in a scheduling paradigm. These routes use real locations and realistic road and vehicle models. The scheduling problem uses the routes and vehicle model created in the routing process, along with a bus model and real routes to schedule the heterogeneous fleet with its heterogeneous scheduling constraints.

Further research in the routing aspect includes a greater exploration of the Pareto front of the problem and hyperparameter tuning, as well as a further exploration of route-generating techniques. Further research in the field includes creating ways to avoid premature convergence and concrete methods to compare metaheuristic methods. Further research in the scheduling aspect includes higher-fidelity charging models as well as the use of more complex pricing strategies and uncontrolled loads. Further research may also include different solution techniques with the graph structure used in this work, different structures for the scheduling graph, and measures to conserve memory.

REFERENCES

[1] Z. H. Ahmed, A. S. Hameed, and M. L. Mutar, "Hybrid genetic algorithms for the asymmetric distance-constrained vehicle routing problem," *Mathematical Problems in Engineering*, vol. 2022, no. 1, p. 2435002, 2022. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1155/2022/2435002

[2] J. Whitaker, "Scheduling charging for fleets of battery electric vehicles: Techniques for modeling and real-time operation," Ph.D. dissertation, Utah State University, 2024.

[3] "ROCKY MOUNTAIN POWER ELECTRIC SERVICE SCHEDULE NO. 6A STATE OF UTAH," January 2021. [Online]. Available: https://www.rockymountainpower.net/content/dam/pcorp/documents/en/rockymountainpower/rates-regulation/utah/rates/006A_General_Service_Energy_Time_of_Day_Option.pdf

[4] C. F. Dursunoglu, O. Arslan, S. M. Demir, B. Y. Kara, and G. Laporte, "A unifying framework for selective routing problems," *European Journal of Operational Research*, vol. 320, no. 1, pp. 1–19, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0377221724001693

[5] S. J. G. Dantzig, R. Fulkerson, "Solution of a large-scale traveling-salesman problem," *Operations Research*, 1954. [Online]. Available: https://doi.org/10.1287/opre.2.4.393

[6] J. H. R. G. B. Dantzig, "The truck dispatching problem," *Management Science*, 1959. [Online]. Available: https://doi.org/10.1287/mnsc.6.1.80.

[7] G. Laporte, Y. Nobert, and D. Arpin, "An exact algorithm for solving a capacitated location-routing problem," *Annals of Operations Research*, vol. 6, no. 9, pp. 291 – 310, 1986, cited by: 133. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-0006234051&doi=10.1007%2fBF02023807&partnerID=40&md5=63e9d06eb6bac25c31d4f813af35d8ee

[8] A. M. . M. Speranza, "Vehicle routing problems over time: a survey," *Annals of Operations Research*, 2022. [Online]. Available: https://doi.org/10.1007/s10479-021-04488-0

[9] M. Solomon, "Algorithms for the vehicle routing and scheduling problems with time window constraints," *Operations Research*, 1987. [Online]. Available: http://dx.doi.org/10.1287/opre.35.2.254

[10] G. Laporte, Y. Nobert, and S. Taillefer, "A branch-and-bound algorithm for the asymmetrical distance-constrained vehicle routing problem," *Mathematical Modelling*, vol. 9, no. 12, pp. 857–868, 1987. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0270025587900042

[11] T. Erdelić and T. Carić, "A survey on the electric vehicle routing problem: Variants and solution approaches," *Journal of Advanced Transportation*, vol. 2019, no. 1, p. 5075671, 2019. [Online]. Available: https://doi.org/10.1155/2019/5075671

[12] I. Kucukoglu, R. Dewil, and D. Cattrysse, "The electric vehicle routing problem and its variations: A literature review," *Computers & Industrial Engineering*, vol. 161, p. 107650, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0360835221005544

[13] R. Basso, P. Lindroth, B. Kulcsár, and B. Egardt, "Traffic aware electric vehicle routing," in *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*, 2016, pp. 416–421. [Online]. Available: https://doi.org/10.1109/ITSC.2016.7795588

[14] R. Basso, B. Kulcsár, B. Egardt, P. Lindroth, and I. Sanchez-Diaz, "Energy consumption estimation integrated into the electric vehicle routing problem," *Transportation Research Part D: Transport and Environment*, vol. 69, pp. 141–167, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1361920918304760

[15] R. G. Conrad and M. A. Figliozzi, "The recharging vehicle routing problem," in *Proceedings of the 2011 industrial engineering research conference*, vol. 8. IISE Norcross, GA, 2011.

[16] S. Zhang, Y. Gajpal, S. Appadoo, and M. Abdulkader, "Electric vehicle routing problem with recharging stations for minimizing energy consumption," *International Journal of Production Economics*, vol. 203, no. C, pp. 404–413, 2018. [Online]. Available: https://ideas.repec.org/a/eee/proeco/v203y2018icp404-413.html

[17] B. L. Golden, L. Levy, and R. Vohra, "The orienteering problem," *Naval Research Logistics (NRL)*, vol. 34, no. 3, pp. 307–318, 1987. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/1520-6750%28198706%2934%3A3%3C307%3A%3AAID-NAV3220340302%3E3.0.CO%3B2-D

[18] I.-M. Chao, B. L. Golden, and E. A. Wasil, "The team orienteering problem," *European Journal of Operational Research*, vol. 88, no. 3, pp. 464 – 474, 1996, cited by: 446. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-0030574659&doi=10.1016%2f0377-2217%2894%2900289-4&partnerID=40&md5=1bb7059f7032bc1d9f71da5cabe30f04

[19] Martins L.d.C., Tordecilla R.D., Castaneda J., Juan A.A., Faulin J, "Electric vehicle routing, arc routing, and team orienteering problems in sustainable transportation," *Energies*, 2021. [Online]. Available: https://doi.org/10.3390/en14165131

[20] Y.-W. Wang, C.-C. Lin, and T.-J. Lee, "Electric vehicle tour planning," *Transportation Research Part D: Transport and Environment*, vol. 63, pp. 121–136, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1361920917301049

[21] D. Bezzi, A. Ceselli, and G. Righini, "Dynamic programming for the electric vehicle orienteering problem with multiple technologies," in *16th Cologne-Twente Workshop on Graphs and Combinatorial Optimization, CTW 2018 - Proceedings of the Workshop*, 2019, Conference paper, pp. 21 – 23, cited by: 0. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85075933867&partnerID=40&md5=25dee4c79b05898209e05a054cdfc333

[22] S. S. Perumal, R. M. Lusby, and J. Larsen, "Electric bus planning & scheduling: A review of related problems and methodologies," *European Journal of Operational Research*, vol. 301, no. 2, pp. 395–413, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0377221721009140

[23] M. Alonso, H. Amaris, J. G. Germain, and J. M. Galan, "Optimal charging scheduling of electric vehicles in smart grids by heuristic algorithms," *Energies*, vol. 7, no. 4, pp. 2449–2475, 2014. [Online]. Available: https://www.mdpi.com/1996-1073/7/4/2449

[24] J. Barco, A. Guerra, L. Muñoz, and N. Quijano, "Optimal routing and scheduling of charge for electric vehicles: A case study," *Mathematical Problems in Engineering*, vol. 2017, no. 1, p. 8509783, 2017. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1155/2017/8509783

[25] O. Sassi, W. R. Cherif, and A. Oulamara, "Vehicle Routing Problem with Mixed fleet of conventional and heterogenous electric vehicles and time dependent charging costs," Oct. 2014, working paper or preprint. [Online]. Available: https://hal.science/hal-01083966

[26] E. Yao, T. Liu, T. Lu, and Y. Yang, "Optimization of electric vehicle scheduling with multiple vehicle types in public transport," *Sustainable Cities and Society*, vol. 52, p. 101862, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2210670719318098

[27] M. Wen, E. Linde, S. Ropke, P. Mirchandani, and A. Larsen, "An adaptive large neighborhood search heuristic for the electric vehicle scheduling problem," *Computers & Operations Research*, vol. 76, pp. 73–83, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0305054816301460

[28] H. Wang and J. Shen, "Heuristic approaches for solving transit vehicle scheduling problem with route and fueling time constraints," *Applied Mathematics and Computation*, vol. 190, no. 2, pp. 1237–1249, 2007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0096300307001543

[29] J.-Q. Li, "Transit bus scheduling with limited energy," *Transportation Science*, vol. 48, no. 4, pp. 521–539, 2014. [Online]. Available: http://www.jstor.org/stable/43666940

[30] L. Zhang, S. Wang, and X. Qu, "Optimal electric bus fleet scheduling considering battery degradation and non-linear charging profile," *Transportation Research Part E: Logistics and Transportation Review*, vol. 154, p. 102445, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S136655452100209X

[31] van Kooten Niekerk, M.E., van den Akker, J.M. and Hoogeveen, J.A., "Scheduling electric vehicles," *Public Transp*, 2017. [Online]. Available: https://doi.org/10.1007/s12469-017-0164-0

[32] G.-J. Zhou, D.-F. Xie, X.-M. Zhao, and C. Lu, "Collaborative optimization of vehicle and charging scheduling for a bus fleet mixed with electric and traditional buses," *IEEE Access*, vol. 8, pp. 8056–8072, 2020.

[33] N. Qin, A. Gusrialdi, R. Paul Brooker, and A. T-Raissi, "Numerical analysis of electric bus fast charging strategies for demand charge reduction," *Transportation Research Part A: Policy and Practice*, vol. 94, pp. 386–396, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S096585641630444X

[34] M. Rinaldi, E. Picarelli, A. D'Ariano, and F. Viti, "Mixed-fleet single-terminal bus scheduling problem: Modelling, solution scheme and potential applications," *Omega*, vol. 96, p. 102070, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0305048318314075

[35] A. Bagherinezhad, A. D. Palomino, B. Li, and M. Parvania, "Spatio-temporal electric bus charging optimization with transit network constraints," *IEEE Transactions on Industry Applications*, vol. 56, no. 5, pp. 5741–5749, Sep. 2020.

[36] H. Chen, Z. Hu, Z. Xu, J. Li, H. Zhang, X. Xia, K. Ning, and M. Peng, "Coordinated charging strategies for electric bus fast charging stations," in *2016 IEEE PES Asia-Pacific Power and Energy Engineering Conference (APPEEC)*, Oct 2016, pp. 1174–1179.

[37] Y. He, Z. Liu, and Z. Song, "Optimal charging scheduling and management for a fast-charging battery electric bus system," *Transportation Research Part E: Logistics and Transportation Review*, vol. 142, p. 102056, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1366554520307079

[38] Rajwar, K., Deep, K. & Das, S., " An exhaustive review of the metaheuristic algorithms for search and optimization: taxonomy, applications, and open challenges," *Artif Intell Rev*, 2023. [Online]. Available: https://doi.org/10.1007/s10462-023-10470-y

[39] F. Behnia, B.-A. Schuelke-Leech, and M. Mirhassani, "Optimizing sustainable urban mobility: A comprehensive review of electric bus scheduling strategies and future directions," *Sustainable Cities and Society*, vol. 108, p. 105497, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S221067072400324X

[40] M. Wei, W. Jin, W. Fu, and X. ni Hao, "Improved ant colony algorithm for multi-depot bus scheduling problem with route time constraints," in *2010 8th World Congress on Intelligent Control and Automation*, 2010, pp. 4050–4053.

[41] J. Whitaker, G. Droge, M. Hansen, D. Mortensen, and J. Gunther, "A network flow approach to battery electric bus scheduling," *IEEE Transactions on Intelligent Transportation Systems*, vol. PP, pp. 1–12, 09 2023. [Online]. Available: https://ieeexplore-ieee-org.dist.lib.usu.edu/document/10137347

[42] D. Mortensen, J. Gunther, and G. Droge, "A bin packing approach to minimize charging cost for electric bus fleets," *IEEE Transactions on Intelligent Transportation Systems*, vol. 25, no. 7, pp. 7818–7831, 2024. [Online]. Available: https://ieeexplore-ieee-org.dist.lib.usu.edu/document/10423914

[43] M. Dorigo, "Optimization, learning and natural algorithms," *Ph. D. Thesis, Politecnico di Milano*, 1992.

[44] C. Blum, "Ant colony optimization: A bibliometric review," *Physics of Life Reviews*, vol. 51, pp. 87–95, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1571064524001258

[45] A. Colorni, M. Dorigo, F. Maffioli, V. Maniezzo, G. Righini, and M. Trubian, "Heuristics from nature for hard combinatorial optimization problems," *International Transactions in Operational Research*, vol. 3, no. 1, pp. 1–21, 1996. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1475-3995.1996.tb00032.x

[46] T. Stützle, "Parallelization strategies for ant colony optimization," in *International Conference on Parallel Problem Solving from Nature*. Springer, 1998, pp. 722–731.

[47] B. Bullnheimer, R. Hartl, and C. Strauss, "A new rank based version of the ant system: A computational study," *Central European Journal of Operations Research*, vol. 7, no. 1, pp. 25–38, 1999.

[48] T. Stützle and H. H. Hoos, "Max-min ant system," *Future Generation Computer Systems*, vol. 16, no. 8, pp. 889–914, 2000. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X00000431

[49] M. Guntsch and M. Middendorf, "A population based approach for aco," in *Applications of Evolutionary Computing*, S. Cagnoni, J. Gottlieb, E. Hart, M. Middendorf, and G. R. Raidl, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 72–81.

[50] C. Blum and M. Dorigo, "The hyper-cube framework for ant colony optimization," *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, vol. 34, no. 2, pp. 1161 – 1172, 2004, cited by: 380. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-1842483279&doi=10.1109%2fTSMCB.2003.821450&partnerID=40&md5=bba95332061abad8de0c629ba0a7ea3f

[51] S. C. Negulescu, C. Oprean, C. V. Kifor, and I. Carabulea, "Elitist ant system for route allocation problem," in *Proceedings of the 8th Conference on Applied Informatics and Communications*, ser. AIC'08. Stevens Point, Wisconsin, USA: World Scientific and Engineering Academy and Society (WSEAS), 2008, pp. 62–67.

[52] S. Iredi, D. Merkle, and M. Middendorf, "Bi-criterion optimization with multi colony ant algorithms," in *Evolutionary Multi-Criterion Optimization*, E. Zitzler, L. Thiele, K. Deb, C. A. Coello Coello, and D. Corne, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 359–372.

[53] M. Awadallah, S. N. Makhadmeh, M. Al-Betar, L. Dalbah, A. Al-Redhaei, S. Kouka, and O. Enshassi, "Multi-objective ant colony optimization: Review," *Archives of Computational Methods in Engineering*, vol. 32, 09 2024.

[54] L. Bianchi, L. M. Gambardella, and M. Dorigo, "An ant colony optimization approach to the probabilistic traveling salesman problem," in *Parallel Problem Solving from Nature — PPSN VII*, J. J. M. Guervós, P. Adamidis, H.-G. Beyer, H.-P. Schwefel, and J.-L. Fernández-Villacañas, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 883–892.

[55] M. Schlüter, J. A. Egea, and J. R. Banga, "Extended ant colony optimization for non-convex mixed integer nonlinear programming," *Computers & Operations Research*, vol. 36, no. 7, pp. 2217–2229, 2009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0305054808001524

[56] G. Bilchev and I. C. Parmee, "The ant colony metaphor for searching continuous design spaces," in *Evolutionary Computing*, T. C. Fogarty, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 25–39.

[57] L. M. Gambardella and M. Dorigo, "An ant colony system hybridized with a new local search for the sequential ordering problem," *Informs Journal on Computing*, vol. 12, no. 3, 2000.

[58] C. Blum, "Beam-aco—hybridizing ant colony optimization with beam search: an application to open shop scheduling," *Computers & Operations Research*, vol. 32, no. 6, pp. 1565–1591, 2005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0305054803003599

[59] C. Silva, J. Sousa, T. Runkler, and R. Palm, "Soft computing optimization methods applied to logistic processes," *International Journal of Approximate Reasoning*, vol. 40, no. 3, pp. 280–301, 2005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0888613X0500037X

[60] S. Fidanova, "Aco algorithm for mkp using various heuristic information," in *Numerical Methods and Applications*, I. Dimov, I. Lirkov, S. Margenov, and Z. Zlatev, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 438–444.

[61] S. Katoch, S. S. Chauhan, and V. Kumar, "A review on genetic algorithm: past, present, and future," *Multimed. Tools Appl.*, vol. 80, no. 5, pp. 8091–8126, 2021.

[62] M. M. Khalid Jebari, "Selection methods for genetic algorithms," *International Journal of Emerging Sciences*, vol. 3, pp. 333–344, 12 2013.

[63] K. Alabdulkareem and Z. H. Ahmed, "Comparison of Four Genetic Crossover Operators for Solving Distance-constrained Vehicle Routing Problem," *International Journal of Computer Science and Network Security*, 2020.

[64] Y. D. Der-San Chen, Robert G. Batson, *Modeling Combinatorial Optimization Problems II.* John Wiley & Sons, Ltd, 2009, ch. 6, pp. 130–151. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118166000.ch6

[65] M. M. Flood, "The traveling-salesman problem," *Operations Research*, vol. 4, no. 1, pp. 61–75, 1956. [Online]. Available: http://www.jstor.org/stable/167517

[66] G. A. Croes, "A method for solving traveling-salesman problems," *Operations Research*, vol. 6, no. 6, pp. 791–812, 1958. [Online]. Available: http://www.jstor.org/stable/167074

[67] "All-Electric 2022 Ford E-Transit Drives Ford Pro Commercial Customer Productivity with End-to-End Charging, Telematics," https://media.ford.com/content/fordmedia/fna/us/en/products/evs/e-transit/2022-ford-e-transit.html, accessed: 2024-10-08.

[68] Z. S. Shamma, B. Jones, M. Clark, C. Bailey, and M. Harper, "Electric vehicle range prediction estimator (EVPRE)," *Software Impacts*, vol. 13, p. 100369, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S266596382200077X

[69] Brooker, Aaron, Gonder, Jeffrey, Wang, Lijuan, Wood, Eric, Lopp, Sean, and Ramroth, Laurie, "FASTSim: A Model to Estimate Vehicle Efficiency, Cost and Performance," in *SAE 2015 World Congress & Exhibition*. SAE International, apr 2015. [Online]. Available: https://doi.org/10.4271/2015-01-0973

[70] R. Bellman, "On a routing problem," *Quart. Appl. Math.*, vol. 16, pp. 87–90, 1958. [Online]. Available: https://doi.org/10.1090/qam/102435

[71] "FedEx Office Pilots Ford E-Transit Vans for FedEx SameDay® City Service," https://newsroom.fedex.com/newsroom/united-states/fedex-office-pilots-ford-e-transit-vans-for-fedex-sameday-city-service, accessed: 2024-10-08.

[72] "`vehicle-data`," accessed: 2024-10-08. [Online]. Available: https://github.com/NREL/vehicle-data

[73] N. Dyer, "2022 Ford E-Transit Electric," https://www.carhp.com/ford/e-transit-electric, accessed: 2024-10-08.