TRANSPORTATION RELATED ALGORITHM DESIGN AND APPLICATION

by

ANTHONY JAMES KULICK

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

DOCTOR OF PHILOSOPHY IN APPLIED MATHEMATICAL SCIENCES

2024

Oakland University
Rochester, Michigan

Doctoral Advisory Committee:

Eddie Cheng, Ph.D., Chair
Daniel Steffy, Ph.D., Co-Chair
Serge Kruk, Ph.D.
László Lipták, Ph.D.
Meir Shillor, Ph.D.

© by Anthony James Kulick, 2024
All rights reserved

This is dedicated to my wife Jennifer and my daughter Sloane, I love you.

ACKNOWLEDGMENTS

I would like to thank my first advisor, Dan Steffy, for his flexibility in scheduling meeting around my hectic work schedule, guiding me towards finding a topic, and his endless feedback and patience while learned to write and debug my code.

I would like to thank my second advisor, Eddie Cheng, for taking over for Dan and assisting me in getting this dissertation across the finish line.

I would like to thank the other professors I've worked with during my time at Oakland university for their excellent instruction and encouragement.

Lastly, I would like to thank my wife Jennifer. Without your encouragement, support, and sacrifice this would not be possible.

Anthony James Kulick

ABSTRACT

TRANSPORTATION RELATED ALGORITHM DESIGN AND APPLICATION

by

ANTHONY JAMES KULICK

Advisers: Eddie Cheng, Ph.D., Daniel Steffy, Ph.D.

In this thesis several algorithms are proposed and developed to solve a variety of transportation related problems. First we considered, a dynamic programming approach to create an exact solver to minimize distance in a vehicle routing problem with time windows (VRPTW) variant. Several new tests are developed to reduce the size of the state space and ultimately reduce the number of state transitions.

Next, we prose a heuristic algorithm based on time slicing (TSH) and a heuristic algorithm based on clustering (CH). It was found that the TSH with 2 time partitions produced solutions at most twice that of the optimal solution. It was found that the CH with n vehicles produces solutions at most n times the optimal.

Finally, an algorithm is proposed for privacy attack on a fleet of autonomous vehicles. By formulating this attack as a shortest path reconstruction problem we show that if such a problem has a unique solution, then at least one of three some conditions hold. Using this result as foundation, an algorithm is proposed to solve a shortest path reconstruction problem. Moreover, we proposed that if a shortest path reconstruction problem has a unique solution, then the proposed algorithm will terminate and provide the correct solution.

TABLE OF CONTENTS

ACKNOWLEDGMENTS iv

ABSTRACT v

LIST OF TABLES ix

LIST OF FIGURES x

LIST OF ABBREVIATIONS xii

CHAPTER ONE
INTRODUCTION 1

1.1. VRP Variants 3

1.1.1. Capacitated VRP 3

1.1.2. Distance-Constrained VRP 3

1.1.3. CVRP with Time Windows 4

1.1.4. VRP with Backhauls 5

1.1.5. VRP with Pickup and Delivery 5

1.2. Formulations for VRPs 6

1.3. VRP Methodologies 7

1.3.1. Exact Methods 8

1.3.2. Approximate Methods 9

1.4. Outline of work 12

CHAPTER TWO
DYNAMIC PROGRAMMING SOLVER 14

2.1. A Summary of Dumas et al. 14

2.2. A Description of the VRP Variant 18

TABLE OF CONTENTS—Continued

2.3. Basic Notation 20

2.4. Description of the VRPTW Algorithm 21

2.4.1. Duplicate Label Test 24

2.4.2. Jump Ahead Test 25

2.4.3. Dominance Test 27

2.4.4. Special Sets Relevant to VRPs 28

2.4.5. Enhancement of Dumas et al. Tests 30

2.5. Implementation 33

2.6. Testing the Algorithm 36

2.6.1. Relationships Between Tests 1 & 2 39

2.7. Complexity Analysis 40

2.7.1. Dumas Tests 41

2.7.2. VRPTW Tests 42

2.8. Conclusions and Extensions 44

CHAPTER THREE
HEURISTIC ALGORITHMS FOR VRPTWS 46

3.1. Introduction 46

3.2. Time Slicing Heuristic 46

3.2.1. Description of the Algorithm 47

3.2.2. Basic Notation and Relationships 48

3.2.3. Analysis of the Algorithm 49

3.2.4. Further Analysis 57

3.3. Clustering Heuristic 61

3.4. Conclusion 63

CHAPTER FOUR
RECONSTRUCTION OF SHORTEST PATHS FROM PATH SEGMENTS 66

4.1. Introduction 66

4.2. Motivating Application 67

4.3. Problem Formulation and definitions 69

4.4. Results 80

4.5. Summary of Work 93

CHAPTER FIVE
CONCLUSIONS 94

5.1. Summary of Results 94

5.2. Further Research 95

REFERENCES 97

LIST OF TABLES

Table 2.1	Example of sub-schedules that result in duplicate labels after re-naming	25
Table 2.2	Average running time (in seconds) for algorithm completion on data sets of size 5, 10, 15, 20, 25, and 30 jobs	38
Table 2.3	Average number of state-label pairs created for algorithm completion on data sets of size 5, 10, 15, 20, 25, and 30 jobs	39

LIST OF FIGURES

Figure 3.1	triangle inequality in 2-dimensional metric space	50
Figure 3.2	Example 1: A tight example for Theorem 3.2.2 with 1 vehicle in a metric space	53
Figure 3.3	Optimal solution to Example 1.	53
Figure 3.4	A heuristic solution to Example 1.	53
Figure 3.5	Example 2: A tight example for 2 vehicles in 1D for Theorem 3.2.2	55
Figure 3.6	Optimal solution to Example 2.	56
Figure 3.7	Heuristic solution Example 2.	56
Figure 3.8	Heuristic and Optimal solutions Example 2 for $n > 2$.	56
Figure 3.9	Example 3: An example of Theorem 3.2.2 failing with more than 1 vehicle where the cost function satisfies the triangle inequality	57
Figure 3.10	Optimal solution to Example 3.	59
Figure 3.11	Heuristic solution Example 3.	59
Figure 3.12	Example 4: A tight bound example for Theorem 3.3.1	62
Figure 3.13	Optimal solution to Example 4.	63
Figure 3.14	Heuristic solution Example 4.	63
Figure 4.1	Example of concatenation of shortest paths do not results in a new shortest path.	70
Figure 4.2	Example of an unambiguous complete solution.	74
Figure 4.3	Example of an ambiguous complete solution.	75

Figure 4.4	Example of an ambiguous incomplete solution.	76
Figure 4.5	Example of a graph with a pair-unique solution, but no edge-unique solution.	78
Figure 4.6	Two edge solutions to a graph with edge unique solution and no pair unique solution.	78
Figure 4.7	Example of a graph with edge unique solution, but no pair unique solution.	79
Figure 4.8	Edge-unique/pair solution to the example for a graph with edge unique solution, but no pair unique solution.	80
Figure 4.9	A second pair solution example for a graph with edge unique solution, but no pair unique solution.	80
Figure 4.10	Example of a graph with ambiguous shortest paths	82
Figure 4.11	Example of a graph with unambiguous shortest paths	83
Figure 4.12	Subgraphs I_{u_1}, I_{u_2}, and I_{u_3}	85
Figure 4.13	Theorem 4.6.2, case 1	88

CHAPTER ONE

INTRODUCTION

Efficiency has never been more important to the functioning of a successful corporation. And as business opportunities continue to consolidate into larger and larger corporations the task of effectively managing the expanding logistic networks becomes increasingly challenging. Of particular concern is the effective management of the distribution of goods and services, Toth and Vigo state that "the transportation process involves all the stages of production and distribution systems and represents a relevant component (generally 10% to 20%) of the final cost of the goods." [1] So, with significant savings at stake the proper management of these distribution networks is of particular interest to any corporation that has to maintain such networks. With that said, the optimization of the distribution of goods between a source (or depot) and sink (customer) is a core concern. With this in mind finding optimal (or near optimal) solutions to the vehicle routing problem (VRP) are relevant to almost any corporation who plans to manufacture and/or distribute goods and/or services.

A VRP is a combinatorial optimization problem defined on a graph $G = (V, E)$ where we are given:

- a set $A \subseteq V$: this set of vertices represents the locations (customers) that must be visited,
- a set B: this set represents the fleet of vehicles where each $b \in B$ has a specified capacity, c_b,
- a weight (or distance) w_e for each $e \in E$ which represents the cost associated with each edge in G,

- a demand, d_v for each $v \in A$ which indicates the quantity that needs to be delivered or serviced required at the location (or customer).

And we are asked to minimize the total cost for the fleet of vehicles to satisfy the following constraints:

- each $v \in A$ must be visited exactly once by a vehicle in the fleet,
- each vehicle must start and end its route at a designated depot,
- the total demand assigned to each vehicle must not exceed its capacity,
- the vehicles must respect the capacity constraint for each vehicle and visit the vertices such that the overall cost is minimized.

When the number of vehicles is equal to one, the vehicle routing problem reduces to the traveling salesman problem (TSP). A TSP is also a combinatorial optimization problem defined on a graph $G = (V, E)$ where we are given:

- a set of vertices represented by a set $A \subseteq V$ which describes the locations that must be visited,
- a weight (or distance) associated with each edge in G represented by w_e for each $e \in E$.

And we are asked to find the minimum cost tour that visits each $v \in A$ exactly once and returns to the starting vertex.

Interest in solving TSPs dates back to the 1950s. The first successful large scale approach to solving a large scale TSP was by Dantzig, Fulkerson, and Johnson [2] while working for the RAND corporation. In their work they were able to successfully find the shortest path to visit a major city in each of the 48 contiguous states in the U.S. and the District of Columbia. In doing so they developed some novel techniques that laid the foundation for the formation of general integer programming and introduced the concepts of the cutting planes method.

1.1 VRP Variants

As with any application problem that seeks to model a real world system there can be many variations on that problem based on the conditions for that particular scenario. Toth and Vigo [1] identify a few of the most studied variations of the VRP. They identified the capacitated VRP, VRP with time windows, VRP with backhauls, and VRP with pickup and delivery.

1.1.1 Capacitated VRP

In the capacitated VRP (referred to as CVRP), first described by Dantzig in [3], all the customers correspond to deliveries and the demands are deterministic, known in advance, and may not be split. The vehicles are identical and based at a single central source (or depot), and only capacity restrictions for the vehicles are imposed. A solution to a CVRP is a collection of K simple circuits, where each circuit corresponds to a vehicle route, which minimizes cost such that:

- each circuit visits the depot vertex,
- each customer vertex is visited by exactly one circuit,
- the sum of the demands of the vertices visited by a circuit does not exceed the vehicle's capacity (referred to as C).

1.1.2 Distance-Constrained VRP

In the distance-constrained vehicle routing problem the distance traveled by any vehicle j may not exceed some previously specified upper bound (referred to as d_j). Commonly, this is paired with the CVRP where each vehicle is given a capacity. In this case, a solution to a distance constrained CVRP is a collection of K simple circuits, where each circuit corresponds to a vehicle route, which minimizes cost such that:

- each circuit visits the depot vertex
- each customer vertex is visited by exactly one circuit
- the sum of the demands of the vertices visited by a circuit does not exceed the vehicle's capacity (referred to as C)
- the length of the circuit assigned to vehicle j does not exceed d_j.

1.1.3 CVRP with Time Windows

The capacitated vehicle routing problem with time windows (referred to as CVRPTW) is an extension of the CVRP in which capacity constraints are imposed on each vehicle and each customer i is associated with a time interval $[a_i,\ b_i]$ (referred as a time window). Where a_i is the earliest time that customer i may be serviced and b_i is the latest time that customer i may be serviced. The time instant in which the vehicles leave the depot, the travel time for each arc $(i,\ j) \in A$ (referred to as t_{ij}), and an additional service time (referred to as s_i) are also given. A solution to a CVRPTW is a collection of K simple circuits, where each circuit corresponds to a vehicle route, which minimizes cost such that:

- each circuit visits the depot vertex
- each customer vertex is visited by exactly one circuit
- the sum of the demands of the vertices visited by a circuit does not exceed the vehicle's capacity C
- for each customer i, the service starts within the time window $[a_i,\ b_i]$, and the vehicle stops for s_i time to service customer i after which service of customer i is completed.

1.1.4 VRP with Backhauls

The capacitated vehicle routing problem with backhauls (referred to as CVRPB) is an extension of the CVRP in which the customer set (that is all vertices other than the depot, $V\backslash\{0\}$, where 0 represents the depot) is partitioned into two subsets. The first set contains n linehaul customers (referred to as set L) who each require some quantity of product to be delivered. The second subset contains m backhaul customers (referred to as set B) who each require a given quantity of product be picked up. A precedence constraint is also applied where linehaul customers must be serviced before backhaul customers. Additionally, there is a demand given for each customer i (referred to as d_i) indicating the amount of product that must be delivered to or picked up from that customer. A solution to a CVRPB is a collection of K simple circuits, where each circuit corresponds to a vehicle route, which minimizes cost such that:

- each circuit visits the depot vertex
- each customer vertex is visited by exactly one circuit
- the total demand of the linehaul and backhaul customers visited by a circuit do not exceed, separately, the vehicle capacity C.
- in each circuit all the linehaul customers precede the backhaul customers, if any.

1.1.5 VRP with Pickup and Delivery

In the most basic version of the capacitated vehicle routing problem with pickup and delivery (referred to as CVRPPD), each customer i is associated with two quantities d_i and p_i, representing the demand of homogeneous commodities to be delivered to and picked up at customer i, respectively. For each customer i the vertex that is the origin of the delivery demand is denoted O_i and the vertex that is

the destination of the pickup demand is denoted D_i. It is assumed that, at each customer location, the delivery is performed before the pickup. A solution to a CVRPPD is a collection of K simple circuits, where each circuit corresponds to a vehicle route, which minimizes cost such that:

- each circuit visits the depot vertex
- each customer vertex is visited by exactly one circuit
- the current load of a vehicle along its circuit must be nonnegative and may never exceed the vehicle capacity C.
- for each customer i, the customer O_i, when different from the depot, must be served in the same circuit and before customer i
- for each customer i, the customer D_i, when different from the depot, must be served in the same circuit and after customer i.

1.2 Formulations for VRPs

Many of the exact and approximate methods that will be reviewed in the next section will use a mathematical programming approach. The advantage of using mathematical programming is that it is often relatively simple to adapt the model of a basic CVRP to include other constraints like those mentioned in the previous section. Additionally, mathematical programming has a number of well-developed techniques that can be applied. There are three basic modeling approaches for the VRP. Those three formulations are called vehicle flow formulations, the commodity flow formulation, and the set-partitioning problem (referred to as SPP).

In the vehicle flow formulation binary decision variables are used to indicate if a vehicle traverses that arc in the graph. The three-index vehicle flow formulation was proposed by Golden et al. [4] and a two-index vehicle flow formulation was

proposed by Laporte et al. [5]. In the commodity flow formulation additional integer variables are associated with the edges and represent the flow of that commodity along the paths traveled by the vehicles. Baldacci et al. [6] presented a two-commodity network flow formulation. In the set-partitioning problem there are an exponential number of binary variables each associated with a different feasible circuit. Balinski et al. [7] proposed a set partitioning formulation for a CVRP. Additionally, in his 2002 book, Toth et al. [1] gives specific formulations for these formulations and discusses additional side constraints for some common VRP variants previously discussed.

1.3 VRP Methodologies

At first glance the VRP appears to be a straightforward scheduling problem, however it is actually quite complex. The basic VRP and its variants (by extension) are, from a computational complexity standpoint, NP-hard problems [8]. Due to the lack of polynomial time solvers for NP-hard problems the approaches to solving VRPs typically fall into two categories: exact methods and approximate methods. An exact method to solve a VRP (or any optimization problem) is designed to find an optimal solution, if one exists. Whereas, with an approximate method we settle for a solution within a percentage of the optimal. While exact methods have the advantage of producing the optimal solution, as the problem size grows, they can become quite slow. Approximate methods, by contrast cannot guarantee an optimal solution, but typically can finish the same problem much faster than an exact method.

1.3.1 Exact Methods

There are a variety of exact methods that can be used to solve a VRP. For VRPs involving a small number of vertices, exact methods can be very effective.

Some of the first forays into exact methods to solve a VRP (and its variants) involve using linear integer, and mixed integer programming. One of the earliest methods developed was by Laporte et al. [9] who proposed two algorithms, one using Gomory cutting planes, and another using branch-and-bound, to solve a distance-constrained CVRP. This remains an active area, as a common class of exact methods continue to rely upon viewing the VRP as a integer linear programming (ILP) or mixed integer linear programming (MILP) problem. Using branch-and-bound, branch-and-price, branch-cut-and-price, etc. have also been popular approaches to solving these formulations. For example Fukasawa et al. [10] successfully developed a branch-cut-and-price algorithm for solving a CVRP. Additionally, Baldacci et al. [11] presents an algorithm which finds a new optimal dual solution to the set partitioning formulation of CVRP, then a series of dual ascent heuristics which include Lagrangean relaxation, price-and-cut generation and classical pricing and cut generation. More recently there has been significant improvements made to the branch-cut-and-price algorithm for a variety of VRP variants [12], [13], [14], and [15]. Another recent development has been combining cut and column generation methods. [16], [11], [10], [17], and [18] developed cut and column generation approaches for specific VRP variants. Baldacci et al. [19] made the first attempt to create a cut and column generation algorithm that would not be specific to a single VRP variant. Currently the best generic solver using some of these branch-cut-and-price is from Pessoa et al. [20].

Another early approach for solving VRPs using exact methods was dynamic programming. In dynamic programming a complex problem is broken into simpler subproblems which should be easier to solve. Christofides et al. [21] presented a general relaxation procedure whereby the state-space associated with a given dynamic programming recursion provides a bound which could be embedded in general branch-and-bound schemes to solve the problem.

1.3.2 Approximate Methods

The approximate methods for the VRP can typically be divided into two types, classical heuristics and meta-heuristics. A classical heuristic, typically, performs a limited exploration of the search space to produce a "good" solution relatively "quickly". Additionally, a classical heuristic tends to be general enough that it is easily adapted to the different VRP variants. A meta-heuristic, typically, performs a deeper search on the most promising area (or areas) of the solution space. Generally, a meta-heuristic will return a higher quality solution than a classical heuristic, but is more time consuming.

Classical heuristics are typically categorized as constructive heuristics or two-phase heuristics. In a constructive heuristic vehicle routes are gradually built while tracking the associated cost. In two-phase heuristics the problem is divided into two parts, the clustering vertices (jobs) into feasible routes and actual route construction with possible feedback between these two phases.

Some well known constructive heuristics are variations on the Clarke and Wright Savings Algorithm [22], [23], [24] the Matching-Based Savings Algorithm [25], [26], [27] and sequential insertion heuristics [28], [29]. Some well known two-phase heuristics devoted to clustering are the sweep algorithm [30], [31], [32], the Fisher and Jaikumar Algorithms [33], the Bramel and

Simchi-Levi Algorithm [34], Truncated Branch-and-Bound [29], and Petal Algorithms [7], [35], [36], [37]. Some well known two-phase heuristics devoted to single-route improvement are λ-opt mechanism [38], [39], [40], [41] and the multi-route improvements [42], [43], [44].

Meta-heuristics can take a variety of forms and employ a wide variety of techniques. Some well-known meta-heuristics that have been developed to solve VRPs are tabu search, simulated annealing, deterministic annealing, genetic algorithms, ant algorithms, and neural network algorithms.

In a tabu search, sequences of solutions are examined and the next move is made to the best neighbor of the current solution under consideration. To avoid cycling between just a few nearby solutions, solutions that were recently considered are forbidden (or tabu), for some finite number of iterations. One of the early successful attempts made using tabu search to solve a VRP was made by Osman [45]. Gendreau et al. [46] presented the Taburoute algorithm which involved several innovative features from the previous tabu search implementations. Rochat and Taillard [47] introduced the concept of adaptive memory to tabu search. An adaptive memory is a pool of good solutions that is dynamically updated throughout the search process. Toth and Vigo [48] presented a new version of tabu search called Granular Tabu Search (referred to as GTS). The main idea behind the GTS is the observation that longer arcs will rarely appear in an optimal solution, so in GTS all arcs with length above a specific threshold are forbidden.

In Simulated Annealing (referred to as SA), an initial solution, x_1 is found. Then at each iteration t a new solution, x, is randomly drawn from the neighborhood of the current best known solution, $N(x_t)$, if x is a lower cost solution than the current best known solution, x_t, the best known solution is updated to x. Otherwise, the x_t remains or it is updated to x based on some probability p. This

probability is dynamically updated over time so the odds of accepting a worse solution decrease with time. One of the early successful SA algorithms was presented by Osman [45]. This algorithm uses a better starting solution, richer solution neighborhoods are explored, and calculation of p is more involved. Breedam [49] did some thorough testing with different versions of SA using varied neighborhood search structures and found that SA is generally outperformed by tabu search.

Deterministic annealing is similar to simulated annealing except that a deterministic rule is used to determine if a worse solution is accepted. Two different deterministic rules have been presented, they are threshold accepting [50] and record-to-record travel [51]. Golden et al. [52] noted that deterministic annealing using the record-to-record travel rule can outperform some tabu search algorithms in some cases.

In a Genetic Algorithm (referred to as GA) a set of random initial solutions is generated (referred to as the parent chromosomes). Then, two new solutions are generated using two the of the parent chromosomes. A random mutation is (possibly) applied to each offspring. This process is applied many times and after some number of generations the algorithm outputs the best solution encountered. Historically, genetic algorithms have focused on VRP variants such as VRPTW exemplified by the work by Potvin et al. [53] and Thangiah [54].

Ant algorithms are based on the food foraging behavior of real-life ant colonies. When ants forage for food they mark their paths with a chemical called a pheromone. Better food sources (those close to the colony or with large quantities of food) are marked with larger quantities of pheromone. These pheromones attract other ants and provide information about the potential food source. Over time, the best paths are marked with more and more pheromone and an efficient food gathering process is created. Colorni et al. [55] first described ant algorithms and

applied one to solve a TSP. In their algorithm they begin by imaging artificial ants searching the solution space similar to how real ants search their environment. They used objective function values to mimic the quality of food source, and used records in an adaptive memory akin to the pheromone trails. Bullnheimer et al. [56, 57] is some of the earliest work done applying ant algorithms to VRPs. In this work the authors showed good solutions could be found in reasonable computation times on several benchmark problems.

A neural network algorithm is designed like the neurons in a human brain. One of the first attempts to apply these algorithms to CVRPs was done by Ghaziri [58] which found traditional heuristics outperformed his algorithms but also had higher computation times.

1.4 Outline of work

In Chapter 2 we will present a new exact dynamic programming algorithm to solve VRPTWs based on the work of Dumas et al. [59]. In their work, Dumas et al. present an exact dynamic programming algorithm to solve a TSP with time windows. They present several tests that are performed to help prune the partial solutions generated by their algorithm. We have adapted several of these tests for a VRPTW and also developed our own novel tests. We then implemented both our novel VRP tests and VRP versions of Dumas et al.'s tests and extended Dumas et al.'s algorithm to solve a VRPTW. We then tested the effectiveness of the VRPTW tests on the performance of our VRPTW algorithm.

In Chapter 3 we will present two meta-heuristic algorithms. The first algorithm is based on time slicing. In this approach, the jobs are "sliced" into groups based on their time window, an exact solver is used to optimize the routes in each slice, then a lowest cost matching algorithm is used to pair the subroutes from

each time slice together to create the vehicle routes. We then prove performance bounds of such an algorithm and give tight examples under varied conditions. The second algorithm is based on clustering. In this approach, the jobs are first grouped into feasible clusters and a vehicle is assigned to each cluster. Then, the route of that vehicle is optimized using an exact algorithm. We then prove performance bounds of such an algorithm and give tight examples under varied conditions.

In Chapter 4 we consider a security protocol described by Bani Hani [60] to protect the identity of autonomous vehicles. Autonomous vehicles need to be able to communicate to each other, but an outside observer should not be able to identify a specific vehicle based on its vehicle ID. Bani Hani identifies one way to stymie potential observers is for vehicles to randomly exchange IDs when they enter certain areas. He describes an attack that the observer can use to attempt to identify the identity of specific vehicles called a T-PAAD attack. We present a theorem which identifies specific conditions that guarantee a successful attack and then use that theorem as a foundation for an algorithm to ascertain the identity of the autonomous vehicles.

CHAPTER TWO

DYNAMIC PROGRAMMING SOLVER

In this section a dynamic programming algorithm is developed that can be used to obtain an exact solution to a vehicle routing problem under the time window constraint (we refer to this a VRPTW problem). In a VRPTW you have a fixed number of vehicles and a set of jobs. Each job has a location, a time window in which it must be accomplished, and a service time. An optimal solution to a VRPTW will assign each vehicle a sequence of jobs such that each vehicle arrives at each assigned jobs inside of that job's time window (and thus form a feasible route for each vehicle to visit and complete the jobs assigned to that vehicle) while minimizing the total cost for all routes.

2.1 A Summary of Dumas et al.

The general approach for our dynamic programming solver for a VRPTW comes from Dumas et al. [59]. In that paper Dumas et al. presented an exact method for solving a traveling salesman problem (TSP) with time windows (referred to as a TSPTW) using a dynamic programming algorithm. Dumas et al. are working in a network $G = (N, A)$ where $N = 1, \ldots, n$ is a set of vertices representing the cities that need to be visited and A is a set of edges which represent the travel between cities. For each vertex $i \in N$ there is a time window $[a_i, b_i]$ in which the vehicle should arrive and a service time s_i. For each edge $(i, j) \in A$ a travel time and travel cost are provided which are denoted by t_{ij} and c_{ij}, respectively. An edge $(i, j) \in A$ is considered feasible if the time needed to service vertex i and to travel along edge (i, j) allows for the possibility of arriving at vertex j by the end of the time window for vertex j, that is $a_i + s_i + t_{ij} \leq b_j$. A

path through the network G is defined as a sequence of vertices $i_1, i_2, \ldots, i_k$ such that each edge in the path, (i_j, i_{j+1}), is in A and the time arriving at node i_j, $j = 1, \ldots, k$ is inside the time window for that node. For any route represented by the feasible path $1, 2, \ldots, k$, the route departure time from node 1 is a_1. For consecutive nodes i and j along this path, the time service begins at t_j where $t_j = \max\{t_i + s_i + t_{ij}, a_j\}$ as long as $t_i + s_i + t_{ij} \leq b_j$ and $t_j = \infty$ otherwise. A optimal TSPTW solution is a least-cost feasible path starting at node 1 and ending at node n that visits every node in N exactly once.

Since this is a dynamic programming approach many partial solutions will be saved to reconstruct a least cost feasible path. $F(S, i, t)$ is defined as the least cost of a path that starts at node 1, visits the nodes in S where $S \subseteq N'$ and $N' = N - \{n\}$, ending at node $i \in S$ and is ready to service node i at time t (or later). Observe that a least cost feasible path needs to consider separately both cost and time. So, the label for each state, (S, i, t), in the dynamic program needs a two-dimensional label containing t and $F(S, i, t)$.

The dynamic programming algorithm took a "bottom-up" approach and created a label each time the viable sub-problem was solved. The "recurrence" defined by Dumas et al. and used to for their algorithm is as follows:

$$F(S, j, t) = \min_{(i,j) \in A} \left\{ F(S - \{j\}, i, t') + c_{ij} \mid t \geq t' + s_i + t_{ij},\ a_j \leq t' \leq b_j \right\} \quad (2.1)$$

for all $S \subseteq N', j \in S$, and $a_j \leq t \leq b_j$,

where the recursion formula is initialized by:

$$\begin{cases} F(\{1, j\}, j, t) = c_{1j} & \text{if } (1, j) \in A \\ F(\{1, j\}, j, t) = \infty & \text{othewise} \end{cases} \quad (2.2)$$

with $a_i \leq b_j$ and

$$\begin{cases} t & = & \max\left\{a_1 + s_1 + t_{1j}, a_j\right\} & \text{if } a_1 + s_1 + t_{1j} \leq b_j \\ t & = & \infty & \text{othewise} \end{cases} \tag{2.3}$$

And the optimal TSPTW solution is given by:

$$\min_{(i,n)\in A} \left\{ \min_{a_i \leq t \leq b_i} \left\{ F(N', i, t) + c_{in} \mid t \geq b_n - t_{in} - s_i \right\} \right\} \tag{2.4}$$

As these labels are created, a dominance check is performed to conserve only Pareto optimal labels. That is, if you have the labels (S, i, t^1) and (S, i, t^2) the label (S, i, t^2) is discarded if $t^1 \leq t^2$ and $F(S, i, t^1) \leq F(S, i, t^2)$. After this dominance test is completed the two-dimensional labels $(t, F(S, i, t))$ of (S, i) can be ordered by increasing time and decreasing cost.

To help cull additional labels maintained using their dynamic programming solver, Dumas et al. performs three checks on each state and label before they are saved. These checks are to ensure that the path represented by the state in question does not induce an infeasible solution in the future. Dumas et al. refer to these as Test 1, Test 2, and Test 3.

However, these tests rely on the computation of some special sets or values. Dumas et al. names these $LDT(i, j)$, $FIRST(S, i)$, and $BEFORE(j)$. The authors define $LDT(i, j)$ as the latest departure time from i such that the vehicle arrives at j in time to service j. $FIRST(S, i)$ is defined as the time value of the first label of two-dimensional labels created for (S, i) during the dominance check. That is, $FIRST(S, i)$ is the earliest know time that the vehicle can arrive at i while also servicing the cities in the set S. And, $BEFORE(j)$ is the set of jobs that must necessarily be visited before a specific job j.

Dumas Test 2.1.1. *Given the states (S, i, t) for all $a_i \leq t \leq b_i$ if the smallest time value to begin service at node i is greater that the latest feasible departure time toward j, for all j, i.e.*

$$FIRST(S, i) > \min_{j \notin S} LDT(i, j) \tag{2.5}$$

then the states (S, i, t) for all $a_i \leq t \leq b_i$ do not admit feasible extensions toward any node.

To calculate $FIRST(S, i)$ the authors maintain a two-dimensional label $(t, F(S, i, t))$ for each (S, i) that they sort by increasing time and decreasing cost. Thus $FIRST(S, i)$ is simply read from the appropriate label, this must be done dynamically. The calculation of $LDT(i, j)$ can be computed in advance and is done by working backward from the end of the time window for j and accounting for the service time needed at i and the travel time from i to j, thus $LDT(i, j) = b_j - t_{ij} - s_i$, these calculation can be done in advance of the start of the algorithm.

Dumas Test 2.1.2. *Given the states (S, i, t) for all $a_i \leq t \leq b_i$ and given node j, $j \notin S$, $(i, j) \in A$, if:*

$$BEFORE(j) \not\subset S, \tag{2.6}$$

then no feasible extensions exist toward j.

The $BEFORE(j)$ sets can be computed in advance and can be done by checking which jobs are impossible to reach starting from j because it takes too long to reach them, i.e. $BEFORE(j) = \{i \mid i \in S - \{j\} \text{ and } a_j + s_j + t_{ji} > b_i\}$.

Dumas Test 2.1.3. *Given the state* (S, i, t) *for a fixed* t, $a_i \leq t \leq b_i$ *and given node* j, $j \notin S$, $(i, j) \in A$, *if* (S, i, t) *can be extended to* j, *i.e.*

$$t \leq LDT(i, j), \tag{2.7}$$

but cannot be extended further to some $k \notin S$, *i.e.*

$$\exists k \notin S,\ t + t_{ij} > LDT(j, k) \tag{2.8}$$

then j *cannot succeed* i *for* (S, i, t) *and the states* (S, i, t'), $t' \geq t$ *are not extended toward* j.

2.2 A Description of the VRP Variant

Using Dumas et al.'s dynamic programming algorithm to solve a TSPTW problem as a framework, we will develop a dynamic programming algorithm to solve a VRPTW problem. The dynamic programming algorithm for a TSPTW problem is a natural starting point for the development of a dynamic programming algorithm for a VRPTW because a TSP is a special case of a VRP, that is, when the number of vehicles is equal to one. This dynamic programming algorithm for VRPs could be used in two ways. If this solver can be demonstrated to be competitive with other exact solvers it could be used as a general solver for a VRP. Even if the solver is rather slow for large VRP problems it could still be effective when implemented to solve "small" VRPs such as finding route improvements in other heuristic or meta-heuristic solvers.

The goal of a VRPTW is to determine the "cheapest" schedule of jobs for each vehicle so that each job location is visited within its service window. Our dynamic programming approach will use a "bottom up" approach and use a breadth first search to determine the feasible extensions for each sub-schedule. We say that

an extension is feasible if for a given vehicle's time and location that vehicle can make it to the new job location inside of that job's service window.

During this dynamic programming approach we will be considering (initially) a VRP where in a graph $G = (X, E)$:

- the triangle inequality holds for the travel times,
- there are k homogenous vehicles,
- the vehicles have infinite capacity,
- cost for each edge is equal to the travel time along that edge,
- there is a single depot where all vehicles start and end their routes,
- a vehicle only needs to arrive at job i before b_j, that is, the vehicle does not have to finish servicing job i before b_j. This assumption accommodates the possibility of a service time exceeding the the customer's time window and adds some flexibility into the scheduling of vehicle routes.

Before we begin, we need to develop some basic vocabulary and notation to describe the VRP. Please consider the following:

2.3 Basic Notation

x_0 = represents the depot, i.e. the start and end of each vehicles' route.

c_{ab} = cost to travel from vertex x_a to vertex x_b

$[e_a, l_a]$ = the service time window for vertex x_a where e_a is the earliest time service can begin and l_a is the latest time service can begin.

t_{ab} = the travel time to go from vertex x_a to vertex x_b

s_a = the time needed for "servicing" at vertex x_a

X = the vertex set, i.e. $\{x_0, x_1, x_2, \ldots, x_n\}$

X' = all the vertices but the depot, i.e. $X' = X - x_0$

S = a subset of X'

Our dynamic programming approach stores a set of states where each state consists of a label giving vehicle information at the current iteration which we denote (S_i, V_i, T_i), a label giving vehicle information at the previous iteration which we denote $(S_{i-1}, V_{i-1}, T_{i-1})$, and a total cost accrued for each vehicle to reach its current location which we denote $F(S_i, V_i, T_i)$. Each vehicle information label consists of the set of visited nodes, the current location (vertex) of each vehicle, and the time that each vehicle arrived at its current location (vertex). At each iteration we attempt to extend a label by adding a single vertex to one of the vehicles and checking for feasibility. So at each iteration i, we consider state i:, $((S_i, V_i, T_i), (S_{i-1}, V_{i-1}, T_{i-1}), F(S_i, V_i, T_i))$, where:

$$
\begin{aligned}
S_i &= \text{the set of vertices visited by the all vehicles at or before state } i, \\
V_i &= \text{a } k \times 1 \text{ column vector where the entry in row } j \text{ descibes the location } (x_a \in S_i) \text{ of vehicle } j \text{ duirng state } i, \\
T_i &= \text{a } k \times 1 \text{ column vector where the entry in row } j \text{ descibes the arrival time of vehicle } j \text{ at vertex } x_a \text{ duirng state } i, \\
F(S_i, V_i, T_i) &= \text{a } k \times 1 \text{ column vector where the entry in row } j \text{ describes the cost of vehicle } j \text{ during state } i.
\end{aligned}
$$

Observe that to identify the location of a specific vehicle at some point in our algorithm we need to indicate which iteration of the algorithm we are referencing and the vehicle. For example the location of vehicle j at iteration i will be denoted as v_j^i which references the j^{th} row of V_i and similarly t_j^i references the j^{th} row of T_i. Additionally, if we want to refer the current location of the vehicles at iteration i in algorithm we will denote this as $\{V_i\}$ whose elements are the entries from the vector V_i.

It is worth noting that any stored state, provides the information to reconstruct all the vehicle routes to that iteration. That is, it describes k routes (one fore each vehicle) that, in totality, service each node in S_i exactly once and vehicle j's route ends at the node v_j^i at time t_j^i. We will refer to the collection of these partial vehicle routes as the *subschedule induced by* (S_i, V_i, T_i).

2.4 Description of the VRPTW Algorithm

The algorithm initializes with all vehicles at the depot at time 0, i.e.

- $S_0 = \{x_0\}$, where x_0 is the depot,

- $V_0 = \begin{bmatrix} x_0 \\ \vdots \\ x_0 \end{bmatrix}_{k\times 1}$, where x_0 is the depot,
- $T_0 = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}_{k\times 1}$,
- $F(S_0, V_0, T_0) = \left\{ \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}_{k\times 1} \right\}$.

First, we create all the labels that represent the feasible subschedules for servicing one job. Recall that at initialization all vehicles are located at the depot so we investigate extending the route of vehicle 1 with one of the vertices in $X - \{x_0\}$, whenever the addition of any of these vertices results in a feasible subschedule a new state is created and that state, the cost of the state, and the necessary information to "trace-back" to the previous state are stored.

Next, we create only the labels that represent the feasible subschedules for servicing two jobs. To do this we use only the states that represent a feasible schedule for servicing one job. For each of these states we investigate extending the route of vehicle 1 with one of the vertices in the set of unserviced jobs. If the addition of any of these vertices represents a feasible subschedule for the service of two jobs, then we check if such a state already exists (this process will be described later) if not a new state is created and that state, the cost of the state, and the necessary information to "trace-back" to the previous state are stored. That process then is repeated for vehicles $2, \ldots, k$.

This process continues for increasing numbers of jobs, i.e., we create only the labels that represent the feasible subschedules for servicing j jobs. To do this we use

only the states that represent a feasible schedule for servicing $j-1$ jobs. For each of these states we extend the route of vehicle 1 by one vertex with each of remaining unserviced jobs. If the addition of any of these nodes represents a feasible subschedule for the service of j jobs, then we check if such a state already exists if not a new state is created and that state, the cost of the state, and the necessary information to "trace-back" to the previous state are stored. That process then is repeated for vehicles $2, \ldots, k$. We continue until all feasible states for n jobs have been explored. At which time we can obtain the minimum cost through the following recursion:

$$F(S_i, V_i, T_i) = \min_{\substack{\forall\ x_a \in \{V_i\}-\{x_0\}, \\ x_b = x_0 \text{ or } x_b \in S_i - \{V_i\}}} \{F(S_{i-1}, V_{i-1}, T_{i-1}) + t_{ba}\}, \tag{2.9}$$

$$\text{and } t_b + t_{ba} < l_a, \tag{2.10}$$

where $S_i,\ V_i,\ T_i$ are given from the state under consideration and $x_a \in N$ is the job most recently added to the subschedule induced by the label (S_i, V_i, T_i). Assume that job x_a has been added to the route for vehicle m, that is $v_i^m = x_a$ and assume $x_b \in N$ is the job that preceded job x_a on the route for vehicle m. Then we define:

- $S_{i-1} = S_i - \{x_a\}$,
- V_{i-1} is the column vector such that for $l = 1, \ldots, k$ and $l \neq m$ we have $v_{i-1}^l = v_i^l$ and for $l = m$ we have $v_{1-1}^m = x_b$,
- T_{i-1} is the column vector such that for $l = 1, \ldots, k$ and $l \neq m$ we have $t_{i-1}^l = t_i^l$ and for $l = m$ we have $t_{1-1}^m = \max\left(t_i^m - t_{ab} - s_b, e_b\right)$.

Observe here that as currently described there could be multiple labels $(S_{i-1}, V_{i-1}, T_{i-1})$ that satisfy the above recursion. We will address this later when describing the dominance test where we will not keep any labels that are Pareto dominated (in terms of entries in T_i and total cost).

Now, by using the relevant track-back information associated with each state we can construct the complete set of routes that represents the minimum cost schedule.

Right away, it becomes clear that the introduction of multiple vehicles in the VRPTW dramatically increases the number of states from that of TSPTW (a single vehicle). Thus it is an immediate concern to design ways to control the number of states that must be created.Next, we will seek to reduce the search space as much as possible by eliminating states that are either redundant or cannot possibly contribute a feasible minimum cost schedule. To eliminate redundant states we developed a *duplicate label test*, a *dominance test*, and a *a jump ahead test*.

2.4.1 Duplicate Label Test

With the duplicate label test we seek to eliminate labels that are duplicates after re-naming. For example refer to Table 2.1 where one partial solution might to vehicle 1 first job 1 and then job 2 and to vehicle 2 first job 3 and then job 4. Another partial solution could assign to vehicle 1 first job 3 and then job 4 and to vehicle 2 first job 1 and then job 2. It is obvious to see that the two labels representing these partial routes would be identical labels after vehicle renaming. To limit the creation of both of these labels we have created a duplicate label test. As our algorithm considers feasible extensions from the state currently being explored before a new label is created the solver will check to see if that identical state (up to renaming) has been explored and discard it if the solver has already encountered such a state. The table 2.1 is an example of duplicate labels, after-renaming. To better facilitate this our labels are always stored so that the vector V_i is in descending order of vertex name. So, using the example in Table 2.1 the first row

vehicle 1	vehicle 2
(1, 2)	(3, 4)
vehicle 1	vehicle 2
(3, 4)	(1, 2)

Table 2.1: Example of sub-schedules that result in duplicate labels after re-naming

represents a state we will call *state* $1 = \left(\{d, 1, 2, 3, 4\}, \begin{bmatrix} 2 \\ 4 \end{bmatrix}, \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} \right)$ and the second row represents a state we will call *state* $2 = \left(\{d, 1, 2, 3, 4\}, \begin{bmatrix} 4 \\ 2 \end{bmatrix}, \begin{bmatrix} t_2 \\ t_1 \end{bmatrix} \right)$. However, this *state 2* would need to be placed in canonical form before its creation. This means *state 2* will be rearranged by sorting the columns in the matrix V_i in descending order (the corresponding entries in all other vectors in this state are also rearranged consistent with V_i) creating the canonical version of *state* $2 = \left(\{d, 1, 2, 3, 4\}, \begin{bmatrix} 2 \\ 4 \end{bmatrix}, \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} \right)$ which is clearly identical to the *state 1.*

2.4.2 Jump Ahead Test

The next method we implemented for eliminating redundant labels is the jump ahead test. The jump ahead test is based on the following principle:

The extension of the subschedule induced from the label (S_i, V_i, T_i) to node j is not allowed if the starting time for job j is greater than the ending time for every job in

$$N - S_i.$$

To better understand the value of this test, recall that this algorithm is designed to explore all possible extensions from the current label before moving on to explore

extensions from the next saved state. With this type of state exploration protocol redundant exploration can easily occur. For example, say we have 2 vehicles at the depot and n jobs. And say that job j has a time window with start time later than all other jobs' end times. We observe that the following two scenarios of job assignments could occur:

- *scenario 1*: vehicle 1 is assigned to job j and waits there until it is time to start job j and vehicle 2 is assigned and services the other $n-1$ jobs.
- *scenario 2*: vehicle 1 is not assigned a job, vehicle 2 is assigned and services the other $n-1$ jobs, and vehicle 1 is assigned and services job j.

We observe that in both scenarios the resulting final labels created are identical (and could be handled by the duplicate label check). However, all the intermediate labels formed to create these two final labels could be different, although the only difference would be vehicle 1 is at the depot in one set of labels and vehicle 1 is at job j in the other. Thus eliminating both scenarios results in reduction of (possibly) many labels. Recall, in Dumas' work they specifically designed their algorithm so that the vehicle first traveled to the next node and if necessary waited there until the time window for that node began. Which, in the case of a TSP, will have the same result as the waiting at the previous job and arriving at the start of the next job's time window since there is only one vehicle. With the way our algorithm's exploration is designed it considers both scenarios (the vehicle proceeding to the new location and then waiting there for the start of the new time window or the vehicle waiting at its current location and then traveling to the new location to arrive at the start of the new time window). As we've argued above we do not need to consider both scenarios because they result in duplicate states, however, the question on the table is "which scenario do we disallow?"

To decide which scenario should be disallowed consider the following possibility, "what if it's not possible for one vehicle to service the jobs $N - \{j\}$?" Then we know immediately that *scenario 1* cannot be a part of feasible schedule, but in addition, *ALL* of the states that could be created from the state represented by *scenario 1* also cannot be part of a feasible schedule. However, if we consider *scenario 2* where vehicle 1 has not yet been deployed (and is still available to assist the service of the jobs in $N - \{j\}$) that state and its intermediate states *COULD* be part of a feasible schedule. And, if it were feasible for one vehicle to service all the jobs in $N - \{j\}$, then those labels are still created in *scenario 2*. Based on this argument we designed the "jump ahead test" to disallow scenario 1.

2.4.3 Dominance Test

Our last method for eliminating redundant states and/or label(s) for a given state is the dominance test. We say that a state, (S_A, V_A, T_A), with cost vector $F(S_A, V_A, T_A) = C_A$ dominates another state, (S_B, V_B, T_B), with cost vector $F(S_B, V_B, T_B) = C_B$ if the following conditions hold:

(1) the vertices visited by the states are the same, that is $S_A = S_B$.

(2) the vehicle location for each vehicle in one state are the same as the vehicle locations in another state for the corresponding vehicles, that is $V_A = V_B$.

(3) there exists at least one index i such that either $t_A^i < t_B^i$ or $c_A^i < c_B^i$ and for all other indices $j = 1, \ldots k$, $j \neq i$, $t_A^j \leq t_B^j$ or $c_A^j \leq c_B^j$.

As each new state, label pair is formed by our algorithm we check first if there is an existing state, label pair that dominates the new state, label pair. If so the new sub-schedule is discarded and no label is created. If the new label is not dominated, then we check to see if the new state, label pair dominates any existing state, label pairs. To do this we consider only the state, label pair that meets conditions (1) and

(2). From those existing state, label pair(s) (if they exist) we check if the state, label pair meets conditions (3) and (4). If so we delete the labels representing those state, label pair(s) and create a label to represent the new state, label pair. Otherwise we create the state, label pair to represent the new sub-schedule.

2.4.4 Special Sets Relevant to VRPs

Now, to eliminate the state and label pairs that cannot be extended to a feasible solution we used the work of Dumas et al. In that work the authors created a dynamic programming approach to solving a traveling salesman problem that included 3 tests to eliminate states that could not be extended into feasible solutions. We will be taking some of their ideas and modifying them to work in a VRP situation. But, before we begin to describe how we adapted the work of Dumas et al. we will summarize some of Dumas' vocabulary and extend these definitions to the VRP situation.

First we define the $BEFORE(x_j)$ in the TSP case as the set of all vertices which must necessarily be visited before the vertex x_j, $x_j \in X'$. That is,

$$BEFORE(x_i) = \{x_j \in X' \ : \ e_j + s_j + t_{ji} > l_i\}. \tag{2.11}$$

We note, by definition, that $x_a \notin BEFORE(x_a)$ for all $x_a \in X'$. To extend this idea to the VRP situation we need to know the set of all nodes that must necessarily be visited before the vehicles arrive at some set of locations, that set of locations represented by the vector V_i from some state (S_i, V_i, T_i). We will denote such a set as $BEFORE(V_i)$ and define it symbolically as:

$$BEFORE(V_i) = \{x_b \in \ X' \ : \ e_b + s_b + t_{ba} > l_a, \ \forall \ x_a \in \ V_i\}. \tag{2.12}$$

Now that we've defined the sets $BEFORE(x_i)$ and $BEFORE(V_i)$ we notice the following relationship:

Lemma 2.4.1. $BEFORE(V_i) = \bigcap_{x_a \in V_i} BEFORE(x_a)$.

Proof.

$$BEFORE(V_i) = \{x_b \in X' \ : \ e_b + s_b + t_{ba} > l_a, \ \forall \ x_a \in V_i\} \tag{2.13}$$

$$= \bigcap_{x_b \in V_i} \{x_b \in X \ : \ e_b + s_b + t_{ba} > l_a\} \tag{2.14}$$

$$= \bigcap_{x_b \in V_i} BEFORE(x_a). \tag{2.15}$$

■

Dumas et al. also defines the quantity $LDT(i, j)$ which describes the "latest departure time" allowed from node i so that node j can be serviced inside its time window. As long as the triangle inequality holds we can calculate the latest departure time by subtracting from the end of the time window for node j the service time for node j, the travel time from node i to node j, and the service time at node i; or using the notation from Dumas:

$$LDT(i, j) = b_j - s_j - t_{ij} - s_i \tag{2.16}$$

The third quantity is $FIRST(S, i)$ which describes the earliest time that vertex i can be reached having already serviced the vertices in $S - \{i\}$. Since this is highly dependent on the other vertices in S this value must be computed each time it is called. Dumas et al. defines $FIRST(S, i) = \min t \ s.t. \ \exists$ a stored state (S, i, t). Dumas et al. describe a data structure that allows them to calculate this value easily. In the VRPTW scenario we will define $FIRST(S_i, j) = \min t$ such that there exists a stored state (S_i, V_i, T_i) such that there exists $a = 1, \ldots, k$ such that $v_i^a = j$

and $t_i^a = t$. That is, it's the earliest time one of the vehicles can arrive at j while the fleet of vehicles has serviced $S - \{j\}$

2.4.5 Enhancement of Dumas et al. Tests

One of the major contributions in the work of Dumas' et al. is the development of three "tests". These tests were used to reduce the size of the search tree in the dynamic programming solver. Their algorithm is designed that each time a new state is encountered three tests are performed to determine if the state (and corresponding label) under consideration can be part of an optimal solution. If any one of these tests fail that state and label are discarded and a new state is considered.

To extend these tests to a VRPTW we will begin by recalling Dumas' Test 1 2.1.1 which was discussed previously. In other words, Dumas' test 1 checks to see if visiting the current vertex i makes the service of another unserviced vertex impossible because the vehicle cannot arrive within that vertex's service window. Dumas et al. accomplish this by examining the earliest known arrival time at vertex i (refer to the set $FIRST(S, i)$) to the minimum latest departure time from vertex i to reach any vertex j (refer to the value $LDT(i, j)$) for $j \notin S$. Furthermore, by considering the earliest time to get to vertex i while servicing the vertices in S with the minimum $LDT(i, j)$ for $j \notin S$ Test 1 can conclude that any state (S, i, t) with $a_i \leq t \leq b_i$ does not admit feasible extensions to any unserviced vertex.

To extend this idea to to a VRP situation recall in the VRP our state, (S_i, V_i, T_i), where S_i consists of a set of vertices that have been serviced, V_i is a column vector of k entries where v_i^a consists of a vertex that represents the location of vehicle a, and T_i is a column vector of k entries where entry t_i^a is the time that vehicle a arrives at location v_i^a. Consider some state (S_i, V_i, T_i) and extend that

state to some unserviced vertex x, thus creating the new state (S_j, V_j, T_j) where $S_j = S_i \cup \{x\}$, entry a in V_j is x (that is $v_j^a = x$) and all other entries are the same as the corresponding entry in V_i (that is $v_j^b = v_i^b$ for $b = 1, \ldots, k$ and $b \neq a$). In the TSP situation we only need to compare the earliest arrival time at x to the minimum latest departure time from x to any unserviced node. However, in the VRP situation this is not enough because we also have to consider what the other vehicles can do.

For example, say vehicle 1 is extended to vertex x the earliest time that vehicle 1 can reach x while servicing the nodes S_i and have the other $k-1$ vehicles at the locations v_i^a for $a = 2, \ldots, k$ is greater than the minimum latest departure time from x to all other unserviced vertices. Without loss of generality say that for node $y \notin S_i$ we have that the vehicle located at x cannot reach node y in time to feasibly service node y. This would imply that vehicle 1 can never feasibly service node y, but it is still possible that one of the other $k-1$ vehicles could feasibly service vertex y. To adapt this test to the VRP situation we need to check if by extending to vertex x if there is any unserviced vertex that is now unservicable by all the vehicles in the new state.

We begin by defining a new $FIRST$ value. Dumas et al. defined $FIRST(S, i)$ as the minimum label time for the state (S, i). We will define $FIRST(S_i, v_i^a)$ as the earliest time that the vertices in S_i have been serviced and vehicle a is at the location v_i^a. Since the latest departure time is irrespective of the number of vehicles that definition for $LDT(*, *)$ remains the same.

VRP Test 2.4.2. *Let (S_i, V_i, T_i) be a state where the time window for each node in V_i is satisfied. The state (S_i, V_i, T_i) cannot be extended to the vertex $x_b \notin S_i$ if*

$$FIRST(S_i, v_j^i) > \min_{x_b \notin S_i} LDT(v_j^i, x_b), \ \forall \ j = 1, \ldots k.$$

Note, there are some major drawbacks in this extension. First, since the state (S_i, V_i, T_i) contains a location vector and a time vector there is not an obvious way to keep an ordered list of states and labels (as Dumas et al. is able to do in the TSP situation) thus creating technical difficulties in retrieving $(FIRST(S_i, V_i^a)$s. Second, again because the state (S_i, V_i, T_i) contains a time vector (and not just a single time) we cannot say anything about the feasibility of other labels that have visited the same S_i vertices and whose vehicles are in the same locations, V_i but arrived at those locations at different times T_j unless for (S_i, V_i, T_j) all the times $t_j^a \geq t_i^a,\ a = 1, \ldots, k$.

Recall Dumas Test 2.1.2 checks that all the nodes that must be feasibly be serviced before node i have, in fact, been serviced (that is, they are a subset of S). As in Dumas Test 2.1.1 the major update we need to make is to consider multiple vehicles. So, rather than the $BEFORE$ set depending on only one vehicle's current location it needs to consider all the vehicles' current locations. We've already done this extension when we defined the set $BEFORE(V_i)$ in Lemma 2.4.1. Thus we have:

VRP Test 2.4.3. *Given the states $(S_{i-1}, V_{i-1}, T_{i-1})$, where the time window for each vertex in V_{i-1} is satisfied, a subschedule induced by the state (S_i, V_i, T_i) where $S_i = S_{i-1} \cup \{x_b\}$ for some $x_b \notin S_{i-1}$ is infeasible if $BEFORE(V_i) \not\subseteq S_i$.*

In this test we have none of the complications we experienced with VRP Test 2.4.2 because we are able to do all the $BEFORE$ calculations in the pre-processing phase as was the case with Dumas test 2.1.2 hence retaining all the computational advantages.

Recall that Dumas Test 2.4.5 checks the future extensions of a state. Specifically it checks if a new vertex j is able to be successfully added to the route

"does that make one of the remaining unserviced nodes, vertex k, unservicable inside its service window?" When we consider the VRP situation some obvious complications arise due to the availability of multiple vehicles. Consider a VRP with two vehicles. Suppose extending vehicle 1 to vertex j makes it infeasible for vehicle 1 to service vertex k in its service window. This may not imply that the extension to j creates a state that is not part of an optimal solution because vehicle 2 may be able to feasibly service vertex k. So to apply this a VRP situation we would need to check if the extension of some vehicle to vertex j makes the extension to some unserviced node, vertex k, infeasible by all the vehicles. In other words, this is now a weaker version of what is being done in VRP Test 2.4.2. Thus it is not necessary to extend Dumas Test to a VRPTW.

2.5 Implementation

Here we will describe how we went from the mathematical formulations described previously to a Python code implementation so that we could conduct performance testing on the practical running time of this algorithm. This project can be viewed at `https://github.com/anthonyjameskulick/OOP_VRP`.

The algorithm requires the following inputs from the user:

- An array of job data with $n + 1$ rows. Each row stores the data for one job (n rows) and the additional row stores the depot's data. This array has six columns. Those six columns store each job's name, x-coordinate, y-coordinate, time window start time, time window end time, and service time.
- The number of vehicles in the fleet, $k = 1, \ldots, n$.
- The tests that the algorithm is using to reduce the search space.

Once an input is received the algorithm conducts a pre-processing phase where the following calculations are performed:

- Travel distances between a vertex $x \in X$ to all other vertices $y \in X,\ y \neq x$. This data is stored in a $(n+1) \times (n+1)$ array. In our testing we assumed distance traveled was equal to travel time however as result we use this array for both time and distance calculations.
- The Dumas $BEFORE(x_a)$ sets are computed for each $x_a \in X'$ this data is stored as a list of lists where the index $a = 1, \ldots, n$ of super-list corresponds to the Dumas Before set for each job.
- Latest departure times are stored in an array where entry (a, b) represents the latest a vehicle can depart from vertex x_a and a arrive at vertex v_b and be able to service the job x_b before the end of the x_b's service window.

The data from the feasible partial solutions encountered by algorithm are managed with a dictionary referred to as the *memo*. The keys in the memo is a list with four entries. The first entry is a tuple of the vertices visited by the vehicles to this point, the algorithm initializes with the tuple zero to represent just the depot. The second entry is a tuple of length k where each entry contains one job, where entry i represents the current location of vehicle i for this state, the algorithm initializes with each vehicle at the depot so this entry is a tuple of length k containing all zeros. The third entry is a tuple of length k where each entry represents the time in which that vehicle arrived at its current location, the algorithm initializes with each vehicle at time zero to represent the start of a work day. The fourth entry is a single integer value referred to as the *key version* (we will discuss later the need for this entry), the algorithm initializes with key version equal to one. The values for each key in the memo is a tuple of length five. The first entry is a tuple of length k where each entry represents the distance traveled by each vehicle to reach the previous state, the algorithm initializes with a tuple of k zeros because each vehicle begins at a common depot having traveled zero distance to get there. The second entry is a

tuple of length k where each entry represents the location of each vehicle at the previous state, the algorithm initializes with an empty tuple of length k since there is no previous locations. The third entry is a tuple of length k where each entry represents the time each vehicle arrived at its previous state, the algorithm initializes with a tuple of k zeros since each vehicle starts the day at the depot. The fourth entry is a tuple of length k where each entry represents the *vehicle order* at the previous state, the algorithm initializes with the tuple $(1, 2, \ldots, k)$. The vehicle order represents the order in which each vehicle's data is listed in each dictionary value tuple. The fifth entry is an integer which represents the *previous key version*, the algorithm initializes with at zero.

A list is also maintained which we refer to as the *queue* which contains the memo keys that need to be checked for feasible extensions. The queue initializes with the key value representing all vehicles at the depot. After initialization, as long as the queue is nonempty, the algorithm will search the feasible ways to extend the states stored in the queue. This search begins by popping the memo key stored first in the queue and retrieves the corresponding values for that key from the memo. Then the algorithm iterates through first unserviced jobs and then vehicles adding one unserviced jobs at a time to one vehicle's route at a time. At each iteration temporary tuples are created to store the updated data from the active memo key and value so that the information represents the updated route being considered in this iteration we refer to this updated information as a temporary key and temporary value or a temporary key-value pair. This temporary key-value pair is now checked for feasibility. The feasibility check proceeds in this order: duplicate label check, time window check, strong jump ahead, VRP test 2, VRP test 1, and finally the dominance check. If at any point one of these checks fail, then all subsequent checks stop and the temporary key under consideration is recorded in a

list of keys that have already been encountered (to be used in the duplicate label check) and the that the temporary key-value pair reverts back to active key-value pair. If all the checks are passed and the temporary key-value pair is not completely dominated by existing key-value pair, but the temporary key-value pair already exists in the memo the temporary key's key version entry is updated by one and this updated temporary key-value pair is stored in the memo, the updated temporary key is added to the queue, and the updated temporary key is added the list of keys that have already been encountered. If all the checks are passed and the temporary key-value pair completely dominates any existing key-value pairs in the memo those dominated key-value pairs are deleted from the memo and the temporary-value pair is added to the memo and the queue. The algorithm then iterates to the next unserviced job, if no unserviced job is available to be checked for the current vehicle the algorithm iterates to the next vehicle and begins iterating the unserviced jobs.

Once the queue is empty the algorithm searches the memo for keys whose first entry indicates that all jobs have been serviced, stores those key-value pairs and uses the first tuple in the each key's value (recall this tuple represents the distance travel by each vehicle to that point) and calculates a total cost for each represented by key-value pair. Then the cheapest such key-value pair is selected and using the values the algorithm traces back through each previous key-value pair to reconstruct the vehicle routes.

2.6 Testing the Algorithm

This algorithm was implemented in Python. All testing represented in this section was done with randomly generated data. The node locations were coordinates randomly chosen integers on the interval $[-100, 100]$, the start times for each job were randomly chosen integers on the interval $[0, 2000]$, the end times for

each job were randomly chosen integers on the interval $[starttime + 25, endtime + 225]$, and the service times were randomly chosen integers on the interval $[0, 25]$.

To test the efficacy of the dynamic programming algorithm for the VRPTW five random data sets were created for feasible VRPTW problems of 5, 10, 15, 20, 25, and 30 jobs. We ran the algorithms using two vehicles under the 6 different configurations, as shown below:

1. All tests on.
2. Dominance test off, all other tests on.
3. Jump ahead test off, all other tests on.
4. Test 2 off, all other tests on.
5. Test 1 off, all other tests on.
6. Duplicate label test, all other tests on.

One approach to analyzing the effectiveness of these tests is to examine the time needed to solve a VRP. By empirically examining the running time we are able to examine how effective these tests are in practice. Another way would be to examine the number of the state-label pairs that needed to be created in order to the solve a VRP. An empirical examination of the state-label pairs gives allows us to see how effective each of the tests are at shrinking the search space for the algorithm. Additionally, this analysis give an indication as to the overlap between some of these tests. That is, the state-label pairs that are eliminated by more than one test.

The average running time for each of these configurations is shown table **??**: Notice that the two configurations that had the largest effect on running time was the presence of the "jump ahead test" and "dominance test" as the configurations two and three had the longest average running time regardless of the size of the data set. Additionally, the algorithm was designed to abort after 1 hour of running time

Table 2.2: Average running time (in seconds) for algorithm completion on data sets of size 5, 10, 15, 20, 25, and 30 jobs

Data Set	All Tests	No Dom	No JA	No T2	No T1	No Dup
5 job average	0.015	0.0136	0.0274	0.0146	0.0144	0.015
10 job average	0.7502	0.62	17.2228	0.7998	0.7142	0.7718
15 job average	161.6526	735.662	2394.8692	188.3862	150.1032	162.0932
20 job average	945.4582	3449.2778	3600*	1020.1434	900.8244	951.9022
25 job average	475.0278	2226.5402	3600*	542.4792	474.5848	484.8416
30 job average	763.011	2222.827	3600*	1314.8436	925.359	776.305

* indicates a timeout occurred

so the average running time for configurations two and three may actually be much longer than what is reported due to many of these runs exceeding the maximum running time. In fact no test problems of size greater than 15 jobs was able to finish inside of the one hour running time without the presence of the "jump ahead test".

It is also of interest that the configuration with the fastest average running time for data sets of 5, 10, 15, 20, and 25 jobs was configuration "No T1". This was a little surprising because that configuration does not include "test 1". However, for data sets of size 25, we found that configuration one's average running time was less than a second slower than configuration five's average running time. And at 30 jobs configuration one had the fastest average running time by a wide margin.

The average number of state-label pairs created for each of these configurations is shown in table **??**: The first, and most obvious, result of looking at the average number of state-label pairs data is that configuration one generates the least average number of state-label pairs for all sizes of data sets. This is expected as each additional test should eliminate more state-label pairs. Based on the data the least effective test is the "duplicate labels test" since in the best case scenario (data sets of 30 jobs) it only eliminates 10.6 more state-label pairs than without it.

Table 2.3: Average number of state-label pairs created for algorithm completion on data sets of size 5, 10, 15, 20, 25, and 30 jobs

Data Set	All Tests	No Dom	No JA	No T2	Not T1	No Dup
5 job average	30.2	30.8	68.8	30.4	30.2	30.2
10 job average	627.6	1195.6	3822.6	650	629.4	627.6
15 job average	6320.8	27188.2	42614	7015.4	6330.4	6321
20 job average	18921.4	85300	55863.2*	19572.2	18937.4	18922
25 job average	11110.8	71232.4	53411.8*	12018.4	11291.2	11122.4
30 job average	18417.2	101706.2	56465.4*	24794.4	20453	18427.8

* indicates a timeout occurred

This is supported by the average running time as configurations one and six are always quite similar in average running time, differing by at most 13.2 seconds (data sets of 30 jobs). The two most effective tests at eliminating state-label pairs is the "dominance test" and "jump ahead test" which supports what we see when we look at the most effective configurations for running time. An unusual results is that the absence of the "dominance test" has a bigger effect on the average number of state-label pairs than the absence of the "jump ahead test" but the absence of the "jump ahead test" has a bigger effect on the average running time than the absence of the "dominance test". One would assume that the number of state-label pairs would directly effect the running time, but that is not what we observe here. The likely cause of this is the "dominance test" involves a lengthy search through all state-label pairs that the "jump ahead test" does not.

2.6.1 Relationships Between Tests 1 & 2

A natural question is: *Do "test 1" and "test 2" actually eliminate different state-label pairs?* An examination of the data in **??** gives us empirical evidence that there are some state-label pairs eliminated by "test 1" but not "test 2" and vice

versa. Take, as an example, the average number of state-label pairs created for 30 job data sets. In configuration one (where all tests used) 18,417.2 state-label pairs are created on average, in configuration four (where all tests except "test 2" are used) 24,794.4 state-label pairs are created on average, and in configuration five (where all tests except "test 1" are used) 20,453 state-label pairs are created on average. This implies that "test 2" eliminated $24,794.4 - 18,417.2 = 6,377.2$ state-label pairs that "test 1" was not able to. Similarly, "test 1" eliminated $20,453 - 18,417.2 = 2,035.8$ state-label pairs that "test 2" was not able to. So clearly, there are state-label pairs that can only be eliminated by "test 1" and other state-label pairs that can only be eliminated by "test 2".

For a more theoretical analysis recall that in "test 1" we need to check for $j = 1, \ldots, k$ that $t_j^{i-1} > LDT(x_a, x_b)$ for all $x_b \notin S_{i-1}$ and where $v_j^{i-1} = x_a$. Recall that the definition for $LDT(x_a, x_b) = l_b - s_b - t_{ab} - s_a$. Therefore, our check becomes for $j = 1, \ldots, k$ that $t_j^i > l_b - s_b - t_{ab} - s_a$ for all $x_b \notin S_{i-1}$ and where $v_j^i = x_a$. This means the decision if the addition of x_b creates a valid label is dependent $l_b,\ s_b,\ t_{ab}$. However, in "test 2" we need to check if $BEFORE(V_i) \subseteq S_i$. Recall that to determine if a vertex, x_b is in the set $BEFORE(V_i)$ we need to check if $e_b > l_a - s_b - t_{ba}$ for all $v_j^i = x_a \in V_i$. This means the decision if x_b is a member of $BEFORE(V_i)$ is dependent on $e_b,\ s_b,\ t_{ba}$. It is worth noting that for a fixed $v_j^i = x_a$ these two tests depend on different values of from x_b and are truly different tests. We also observe, that in the case where x_b is mutually conflicting with each $v_j^i = x_a$, then these two tests will reject the same label.

2.7 Complexity Analysis

Since the VRP is a well-known NP-hard problem we do not expect to find a polynomial time algorithm. However, examining the computational cost of the tests

introduced in the previous sections is of interest as it can help inform situations where these tests could be best utilized. We begin with an analysis of the tests described by Dumas et al.

2.7.1 Dumas Tests

- Dumas Test 1

  Recall, that in Dumas et al.'s Test 1 some state (S, i, t) which satisfies the time window requirements for node i we check $FIRST(S, i) > \min_{j \notin S} LDT(i, j)$. As part of Dumas et al.'s book-keeping for their algorithm two separate lists of (S, i) are kept, one organized by increasing time and the other by decreasing cost. Thus the $FIRST(S, i)$ computation is a constant time operation. We also note that the $LDT(i, j)$ computations are completed in the preprocessing phase and stored in an array thus resulting in a constant time look up for each node j, $j \notin S$, that is $|N - S|$ array lookups and comparisons to the $FIRST(S, i)$ value. Thus, this test has complexity $\mathcal{O}(N)$ where $|N|$ is the number of nodes to be visited.

- Dumas Test 2

  Recall, that in Dumas et al.'s Test 2 for the state (S, i, t), we need to check the subset inclusion of $BEFORE(j)$ and S for some $j \notin S$, that is $BEFORE(j) \not\subset S$. Note here that $BEFORE(j)$ is computed in the preprocessing phase this requires a single lookup. Thus, this test has complexity $\mathcal{O}(1)$ or constant time.

- Dumas Test 3

  Recall, that in Dumas et al.'s Test 3 for the state (S, i, t) for any feasible extension of (S, i, t) to some node $j \notin S$ we need to check if the further extension to any $k \notin S \cup \{j\}$ is infeasible. To determine if the extension to

node j is feasible we need to check if $t \leq LDT(i,j)$ which can be done in constant time. To check, for each node $k \notin S \cup \{j\}$ if there is an infeasible extension to some other node k we need to check if $t + t_{ij} \leq LDT(j,k)$. Thus, this test has complexity $\mathcal{O}(N)$ where $|N|$ is the number of nodes.

Note, that no VRP equivalent for Test 3 was created.

- Dominance Test

  Recall, that in Dumas et al.'s algorithm they are only storing the states that are Pareto optimal elements. That is given states (S, i, t^1) and (S, i, t^2), the state (S, i, t^2) is eliminated if $t^1 \leq t^2$ and $F(S, i, t^1) \leq F(S, i, t^2)$. So, for the state (S, i, t), this involves searching the stored states for all states with the label $(S, i, _)$ and comparing the times to t and the costs $F(S, i, t)$. Due to the book-keeping employed by Dumas et al. as described previously this is a constant time lookup for each state with label $(S, i, _)$. Thus, this test has complexity $\mathcal{O}(L)$ where L is the length of the list of stored states for the form $(S, i, _)$.

Now that we have a thorough understanding of the complexity of the tests employed by Dumas et al. we can conduct an analysis of the tests developed for the VRPTW.

2.7.2 VRPTW Tests

- VRPTW Test 2.4.2

  Recall, that this requires that for a state (S_i, V_i, T_i), that we check $FIRST(S_i, v_j^i) > \min_{x_b \notin S_i} LDT(v_j^i, x_b),\ \forall\ j = 1, \ldots, k$ where k is the number of vehicles in the fleet. Much like Dumas et al. we will employ a specific list to aid in the $FIRST(S_i, v_j^i)$ calculation. Each time a label $(S_i, V_i, _)$ is stored the minimum time the nodes in V_i are encountered are stored in an list. Thus computation for $FIRST(S_i, v_j^i)$ is a constant time lookup for each of the k

vehicles. As, in the Dumas et al. version of Test 1 the $LDT()$ calculations are done in the preprocessing phase and stored in an array and therefore can be obtained in constant time. So for we do one lookup for the $FIRST()$ value for each vehicle and for each of these $FIRST()$ values we do $|N - S_i|$ comparisons to the $LDT()$ values of the remaining unvisited nodes. Thus, this test has complexity $\mathcal{O}(Nk)$ where $|N|$ is the number of nodes to be visited and k is the number of vehicles in the fleet.

- VRPTW Test 2.4.3

  Recall, that this test for some state (S_i, V_i, T_i) is used to check if the extension to the most recently added node $j \in S_i$ is possible we need to determine if the $BEFORE(V_i)$ is a subset of S_i. Note that as mentioned previously that $BEFORE(V_i)$ is the intersection of the before set for each of the k vehicles and just like in Dumas et al. the $BEFORE(j)$ computation is done in the preprocessing phase so the look up for these k sets can be done in constant time. Thus, this test has complexity $\mathcal{O}(k)$ where k is the number of vehicles in the fleet.

- VRPTW Dominance Test

  Recall, as in Dumas et al.'s algorithm we only store the states that are Pareto optimal elements. That is given states (S_i, V_i, T_i) and (S_i, V_i, T_j) we eliminate state (S_i, V_i, T_j) if $t_a^i \leq t_a^j$ and $F(S_i, v_a^i, T_i) \leq F(S_i, v_a^j, T_j),\ \forall\ a = 1, \ldots, k$. So for the state (S_i, V_i, T_i), this involves searching the stored states for the all states with the label $(S_i, V_i, _)$ and for each state comparing the time and cost for each vehicle in these stored states to the time and cost for each vehicle in (S_i, V_i, T_i). Once again if we employ a list that stores the time and cost vectors that correspond to all stored states with the label $(S_i, V_i, _)$ the lookup for these labels can be done is constant time. Thus we for each stored label we

need to do a comparison of time and cost for each vehicle. Thus, this test has complexity $\mathcal{O}(kL)$ where L is the length of the list of stored states for the form $(S_i, V_i, _)$ and k is the number of vehicles in the fleet.

- Jump Ahead Test

  Recall, that Jump Ahead Test checks for a nodes whose start time is later than the end time of remaining nodes. So, a state (S_i, V_i, T_i) will not be extended to a vertex $j \notin S_i$ if $a_j \geq b_d, \ \forall \ d \in N - S_i - j$. Note, that a_j is the job start time for node j and is a constant time lookup and that b_d is the job end time for node d and is also a constant time lookup. So we have $|N - S_i| - 1$ constant time comparisons. Thus, this test has complexity $\mathcal{O}(N)$ where $|N|$ is the number of nodes to be visited.

- Duplicate Label Test

  Recall, that the Duplicate Label Test checks to see if we've encountered this exact state before for the exact same cost. Once again by keeping a list for each encountered label (S_i, V_i, T_i) with its cost vector we can accomplish the lookup in constant time. All that would remain is checking the cost vector of each of these previously encountered states with the current state. Thus, this test has complexity $\mathcal{O}(k\mathcal{L})$ where $\mathcal{L}$ is the length of the list of previously encountered states for the form (S_i, V_i, T_i) and k is the number of vehicles in the fleet.

2.8 Conclusions and Extensions

We have constructed a dynamic programming algorithm that can find exact solutions to VRPs by extending the work of Dumas et al. [59]. Additionally, we created two novel tests, the "jump ahead test" and the "duplicate label test" and successfully adapted Dumas' test 1 and test 2 for VRPs creating "New Test 1" and

"New Test 2". And, while effective in finding correct solutions to these problems our implementation performed poorly in comparison to other modern solvers. For example, Pessoa et al. [20] developed an exact solver that was able to solve most Solomon instances (VRPTW each of 100 customers) in less than 5 minutes with the most difficult instance taking 37 minutes. Even in relatively small instances (forty customers) and the simplest situation (2 vehicles), the running time for this algorithm, by comparison, was approaching 1 hour. Despite this we did find some encouraging results when we examine the effectiveness of our tests. In particular, the "jump ahead test" was extremely effective in reducing solving time. We noticed that the longest running times corresponded to the solver configuration without the "jump ahead test", often exceeding the 1 hour time limit we included on the solver.

While conducting performance testing on our algorithm we observed that most of our time was spend searching dictionary keys. Perhaps, improvements to the implementation could provide significant time savings. Or perhaps there is a data structure solution that would allow for better organization and storage in order to make these searches less time consuming.

CHAPTER THREE

HEURISTIC ALGORITHMS FOR VRPTWS

3.1 Introduction

The vehicle routing problem is a generalization of the well-known NP-hard traveling salesman problem. In the case of an NP-hard problem using any exact algorithm can present challenges often in the form of lengthy running time. This running time issue is nicely illustrated by our results from Chapter 2. The obvious approach in these situations is to use a heuristic algorithm where the optimality of the solution is sacrificed for improved running time. Now, if we think about possible real-world applications for a VRP we can imagine a scheduling deliveries for a courier service or scheduling service technicians making house calls or scheduling jobs for a taxi or ride sharing service as obvious applications. In a practical sense most of these types of applications involving getting new information or updated information and require updating their schedule in real time. In these sorts of situations it could be useful to plan one time interval at time in order to leave the planning flexibility for new or updated job information.

In this chapter we explore some heuristic approaches and investigate analyze the worst case behavior and provide tight examples of these simple heuristic methods. The first such approach is, what we will refer to as, *time slicing heuristics*.

3.2 Time Slicing Heuristic

There are many versions of the vehicle routing problem in this approach we will focus of the VRP with the following constraints:

- All vehicles must start and end the business day at the depot.

- All vehicles have infinite capacity, by this we mean that each vehicle can service an unlimited amount of customers.
- Each customer supplies a time window for service.

3.2.1 Description of the Algorithm

The design of a time slicing heuristic is to divide the job requests into groups (or slices) by placing jobs with similar time windows into the same group. In doing so we divide the main problem into subproblems, each with a smaller number of jobs. Since time is a continuous quantity there is no limit to the number of slices that could be made. We begin by considering the situation where there are two slices. An obvious, real world, way to create two slices could be to divide the problem into two groups (A.M. and P.M.). An exact solver could then used to find an optimal solution for the A.M. schedule and then to find an optimal solution to the P.M. schedule. The A.M. schedule for each vehicle begins at the depot and then lists the order in which the A.M. customers are serviced where the last A.M. customer serviced is the end of that vehicle's A.M. route. The P.M. schedule for each vehicle begins with the first P.M. customer to be serviced by that vehicle, and then lists the order in which the P.M. customers are serviced, and then ends at the depot. Each route in the A.M. schedule is then paired with a route in the P.M. schedule, so that the last customer in an A.M. schedule is matched to the first customer in a P.M. schedule, to create a complete route for each vehicle. This matching of A.M. and P.M. schedules will be done in the optimal way by computing a minimum cost bipartite matching, which is a problem that can be solved in polynomial time.

3.2.2 Basic Notation and Relationships

To develop an understanding of the worst case behavior of this approach we will define some notation and relationships that are a direct consequence of the procedure described above.

- Let TSH be the cost of the time slicing heuristic solution.
- Let A_{TSH} be the cost of the morning schedule from the heuristic solution.
- Let P_{TSH} be the cost of the afternoon schedule from the heuristic solution.
- Let C_{TSH} be the cost of the connections between the morning and afternoon schedules in the heuristic solution.
- Let OPT be the cost of the optimal solution.
- Let A_{OPT} be the cost of the morning schedule from the optimal solution.
- Let P_{OPT} be the cost of the afternoon schedule from the optimal solution.
- Let C_{OPT} be the cost of the connections between the morning and afternoon schedules.

Now that we've defined the different part of our proposed heuristic solution and the optimal solution we can identify several relationships that are a direct result of the above definitions:

- Since OPT is the cost of an optimal schedule, we have:

$$OPT \leq TSH. \tag{3.1}$$

- Since A_{TSH} is the cost of a optimal schedule for the A.M. jobs, we have:

$$A_{TSH} \leq A_{OPT}. \tag{3.2}$$

- Since P_{TSH} is the cost of a optimal schedule for the P.M. jobs, we have:

$$P_{TSH} \leq P_{OPT}. \tag{3.3}$$

- From the definitions of OPT and TSH and equation 3.1, we have:

$$A_{OPT} + C_{OPT} + P_{OPT} \leq A_{TSH} + C_{TSH} + P_{TSH}. \tag{3.4}$$

- From equation 3.4, 3.2, and 3.3, we have:

$$C_{OPT} \leq C_{TSH}. \tag{3.5}$$

- From equations 3.5, 3.2, and 3.3, we have:

$$(A_{TSH} - A_{OPT}) + (P_{TSH} - P_{OPT}) \leq C_{TSH}. \tag{3.6}$$

3.2.3 Analysis of the Algorithm

Now that we've defined the parts of our heuristic solution and developed some basic relationships between our heuristic solution and the optimal solution we can begin exploring the worst case performance of a time slicing approach. That is, in the worst case, "how much better is the optimal solution than our heuristic solution?" Using the equations above the only part of the heuristic solution that can be outperformed by the optimal solution is the connections costs between the time slices. As a result, we begin by investigating the maximum cost connecting two jobs.

Lemma 3.2.1. *For vehicle routing problems using n vehicles where the cost function satisfies the triangle inequality, the maximum distance between any two jobs is at most $OPT/2$.*

Proof. Consider a vehicle routing problem with n vehicles with a cost function that satisfies the triangle inequality. Let D be the cost of the longest arc between any two jobs (j_h, j_k). And let $SCHEDULE$ be the cost of the schedule that includes the arcs $(depot, j_h),\ (j_h, j_k),\ (j_k, depot)$. Since any feasible schedule must visit both j_h and j_k we know that the cost of any feasible schedule must be at least $SCHEDULE$. Specifically, $OPT \geq SCHEDULE$. The cost for $SCHEDULE$ is:

$$\begin{aligned} SCHEDULE &= \text{cost}(depot, j_h) + \text{cost}(j_h, j_k) + \text{cost}(j_k, depot) \\ &= \text{cost}(depot, j_h) + D + \text{cost}(j_k, depot). \end{aligned}$$

And, by the triangle inequality, we have that

$\text{cost}(depot, j_h) + \text{cost}(j_k, depot) \geq \text{cost}(j_h, j_k) = D$. Which is demonstrated in Figure 3.1.

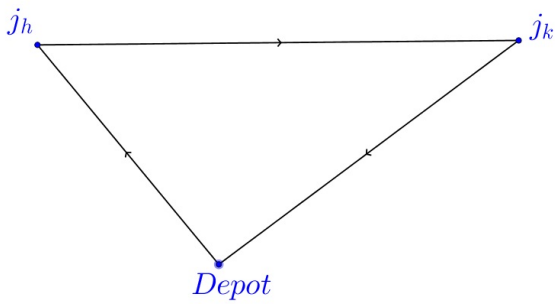

Figure 3.1: triangle inequality in 2-dimensional metric space

$$\begin{aligned} SCHEDULE &= \text{cost}(depot, j_h) + \text{cost}(j_h, j_k) + \text{cost}(j_k, depot) \\ &= \text{cost}(depot, j_h) + D + \text{cost}(j_k, depot) \\ &\geq D + D \\ &= 2D. \end{aligned}$$

Therefore we have that:

$$\begin{aligned} 2D &\leq SCHEDULE \leq OPT \\ D &\leq OPT/2. \end{aligned}$$

■

Now that we have a bound for the maximum connection cost for a route we can begin to develop a performance bound for our heuristic solution. We begin by

considering a simplified version of the problem were either we have 1 vehicle in metric space or we have n vehicles in a 1-dimensional space.

Theorem 3.2.2. *For Vehicle Routing Problems using 1 vehicle where the cost function satisfies the triangle inequality or for n vehicles in 1-dimensional space; the cost of the heuristic is at most 1.5 times the cost of the optimal schedule.*

Proof. Let D be the greatest cost to travel between two jobs. Observe that the connection costs will be at most D. With a single vehicle where the cost function satisfies the triangle inequality the connection cost involve "connecting" one early slice with one later slice and since D is the greatest cost to travel between two jobs the maximum cost of that connection is D. For n vehicles in a 1-dimensional space assume, for the sake of contradiction, that the connection costs are more than D. If $n = 1$ this is clearly a contradiction as D is the greatest cost to travel between two jobs. Now, consider $n > 2$. Without loss of generality, let $n = 2$. This means we have two arcs that make up the connection costs. Let C_1 be the connection between route A_1 and route P_1 and let C_2 be the connection between route A_2 and route P_2. Since this $cost(C_1) + cost(C_2) > D$, the greatest distance between two jobs is D, and the jobs making up routes A_1, A_2, P_1, P_2 are in a 1-dimensional space we know that the arcs C_1 and C_2 must overlap. By connecting A_1 to P_2 and A_2 to P_1 we can create connections C_3 and C_4, respectively, that do not overlap. Therefore $cost(C_3) + cost(C_4) < cost(C_1) + cost(C_2)$ this contradicts that $cost(C_1) + cost(C_2)$

is the optimal connection cost.

$$
\begin{aligned}
TSH &= A_{TSH} + P_{TSH} + C_{TSH} && \text{this is the definition of } TSH \\
&\leq A_{OPT} + P_{OPT} + C_{TSH} && \text{since } A_{TSH} \text{ and } P_{TSH} \text{ are optimal solutions} \\
&\leq A_{OPT} + P_{OPT} + D && \text{by the observation above} \\
&\leq A_{OPT} + P_{OPT} + OPT/2 && \text{by lemma 3.2.1} \\
&\leq OPT + OPT/2 && \text{by the definition of } OPT \\
&\leq 3/2OPT
\end{aligned}
$$

■

We will now give an example to demonstrate that the bound stated in Theorem 3.2.2 is tight. Since Theorem 3.2.2 covers two cases (1 vehicle where the cost function satisfies the triangle inequality and n vehicles in 1-dimensional space). First, we will consider the case of 1 vehicle where the cost function satisfies the triangle inequality. In this case, let there be three jobs J_1, J_2, and J_3 and a depot where J_1 and J_2 are in the A.M. schedule and J_3 is in the P.M. schedule. Those three jobs and a depot that are located at:

$$
\begin{aligned}
depot &= 0 \\
J_1,\ J_3 &= -1 \\
J_2 &= 1 + \epsilon
\end{aligned}
$$

We will refer to this situation as Example 1, which can be seen in Figure 3.2. The optimal solution for this problem uses a single vehicle (all other remain at the depot). That vehicle leaves the depot, services job 2, then job 1, then job 3, and then returns to the depot. This schedule can be seen in Figure 3.3. The cost for this schedule, $OPT = 4 + 2\epsilon$, is calculated below:

Figure 3.2: Example 1: A tight example for Theorem 3.2.2 with 1 vehicle in a metric space

$$\begin{aligned} A_{OPT} &= \text{cost}(depot,\ J_2) + \text{cost}(J_2,\ J_1) &= 1 + 2 + 2\epsilon \\ C_{OPT} &= \text{cost}(J_1,\ J_3) &= 0 \\ P_{OPT} &= \text{cost}(J_3,\ depot) &= 1 \end{aligned}$$

Now, consider a heuristic solutions for this problem, once again using one vehicle. One solution would be identical to the optimal solution described previous, however another feasible output from the TSH has the vehicle leave the depot, service job 1, then job 2, then job 3, and then return to the depot. This schedule can be seen in Figure 3.4. Obviously, this schedule is more costly, specifically the cost for that schedule, $TSH = 6 + 2\epsilon$, is calculated below:

$$\begin{aligned} A_{TSH} &= \text{cost}(depot,\ J_1) = \text{cost}(J_1,\ J_2) &= 1 + 2 + 2\epsilon \\ C_{TSH} &= \text{cost}(J_2,\ J_3) &= 2 + \epsilon \\ P_{TSH} &= \text{cost}(J_3,\ depot) &= 1 \end{aligned}$$

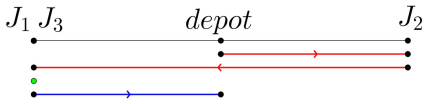

Figure 3.3: Optimal solution to Example 1.

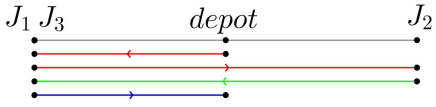

Figure 3.4: A heuristic solution to Example 1.

If we consider the most expensive feasible output of the TSH and let $\epsilon \to 0$, we have that the $TSH = \frac{3}{2}OPT$, demonstrating that the bound stated in the Theorem 3.2.2 is tight.

Now we will consider the case were we have n vehicles in 1-dimensional space. In attempting to find a tight example for Theorem 3.2.2 in a 1-dimensional space with n vehicles we were successful when $n = 2$, but we needed to add a precedence constraint for the A.M. jobs (which was not necessary with 1 vehicle). It is worth noting that without this precedence constraint that the time slicing heuristic will always return the optimal solution for $n \geq 2$ in one dimension. Suppose we have 5 jobs $J_1,\ J_2,\ J_3,\ J_4$, and J_5 and a depot where $J_1,\ J_2, J_3$ are on the A.M. schedule and J_4 and J_5 are on the P.M. schedule. And, as we mentioned moments ago we also need to include a precedence constraint which states that J_1 and J_2 must be completed before J_3. Those five jobs and a depot that are located at:

$$\begin{aligned} \textit{depot},\ J_3,\ J_5 &= 0 \\ J_1,\ J_4 &= -1 \\ J_2 &= 1 \end{aligned}$$

Since precedence is important we have drawn this 1-dimensional example in a 2-dimensional space where the vertical axis represents time. Since the jobs J_1 and J_2 must be serviced before J_3 we drew J_1 and J_2 lower than J_3. We will refer to this as Example 2, which can be seen in Figure 3.5: For 2 vehicles in 1-dimension, the optimal solution will call for one vehicle servicing jobs J_1 and J_4 and the other vehicle servicing jobs $J_2,\ J_3$, and J_5, as seen in Figure 3.6. The cost of this schedule, $OPT = 4$, is calculated below:

$$\begin{aligned} A_{OPT} &= \text{cost}(\textit{depot},\ J_1) + \text{cost}(\textit{depot},\ J_2,\ J_3) &= 1 + 2 \\ C_{OPT} &= \text{cost}(J_1,\ J_4) + \text{cost}(J_3,\ J_5) &= 0 + 0 \\ P_{OPT} &= \text{cost}(J_4,\ \textit{depot}) + \text{cost}(J_5,\ \textit{depot}) &= 1 + 0 \end{aligned}$$

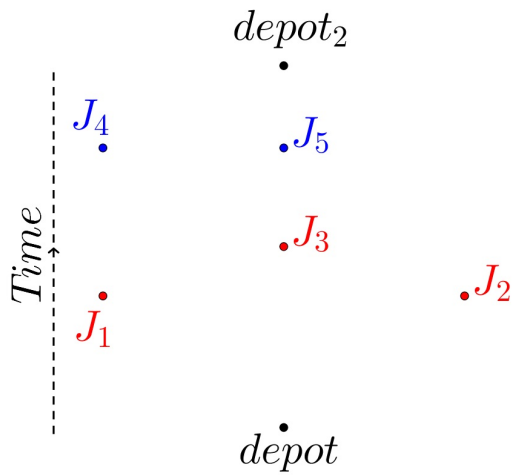

Figure 3.5: Example 2: A tight example for 2 vehicles in 1D for Theorem 3.2.2

For 2 vehicles in 1-dimension, consider a possible heuristic solution can have one vehicle servicing jobs J_1, J_3, and J_4 and the other vehicle servicing jobs J_2 and J_5, as seen in Figure 3.7. The cost of this schedule, $TSH = 6$, is calculated below:

$$\begin{aligned} A_{TSH} &= \text{cost}(depot,\ J_1,\ J_3) + \text{cost}(depot,\ J_2) &&= 2+1 \\ C_{TSH} &= \text{cost}(J_3,\ J_4) + \text{cost}(J_2,\ J_5) &&= 1+1 \\ P_{TSH} &= \text{cost}(J_4,\ depot) + \text{cost}(J_5,\ depot) &&= 1+0 \end{aligned}$$

Thus, we have that the $TSH = \frac{3}{2}OPT$ for 2 vehicles in 1-dimension.

However, if we try to extend this example to $n > 2$ vehicles we get the same schedule for both the optimal and heuristic solutions where $TSH = OPT = 4$. In each case one vehicle services jobs J_1 and J_4, another vehicle services jobs J_3 and J_5, and the third vehicle services job J_2, as seen in Figure 3.8. The cost of such a schedule is calculated below:

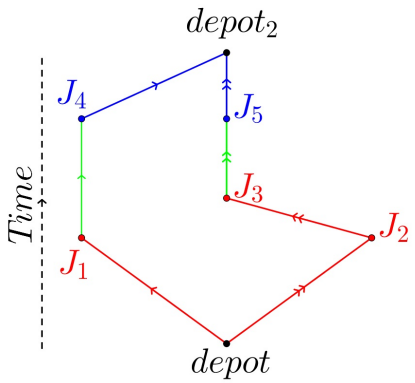

Figure 3.6: Optimal solution to Example 2.

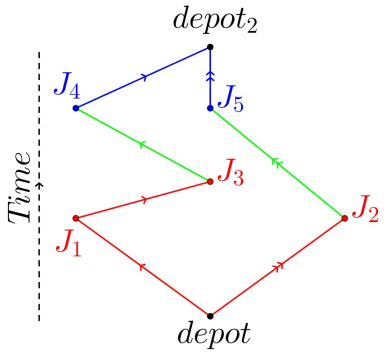

Figure 3.7: Heuristic solution Example 2.

$$
\begin{aligned}
A_{TSH} &= \text{cost}(depot,\ J_1) + \text{cost}(depot,\ J_3) + \text{cost}(depot,\ J_2) &&= 1+0+1 \\
C_{TSH} &= \text{cost}(J_1,\ J_4) + \text{cost}(J_3,\ J_5) + \text{cost}(J_2,\ J_2) &&= 0+0+0 \\
P_{TSH} &= \text{cost}(J_4,\ depot) + \text{cost}(J_5,\ depot) + \text{cost}(J_2,\ depot) &&= 1+0+1
\end{aligned}
$$

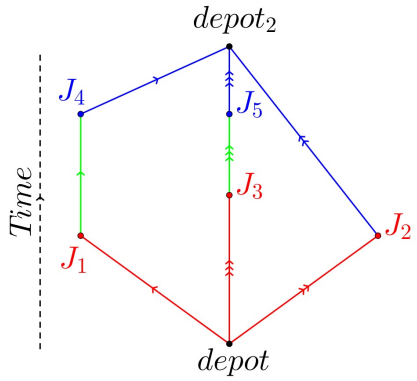

Figure 3.8: Heuristic and Optimal solutions Example 2 for $n > 2$.

3.2.4 Further Analysis

The obvious next question is "does this bound extend to more than 1 vehicle where the cost function satisfies the triangle inequality?" It does not, to demonstrate this consider the family of tight examples with the following assignment of jobs:

- depot at $(0, 0)$
- A.M. jobs $(\frac{i}{m}, \pm\frac{1}{m}),\ \forall\ i = 1, \ldots, m$
- P.M. jobs $(-\frac{i}{m}, \pm\frac{1}{m}),\ \forall\ i = 1, \ldots, m$

For visualization purposes, consider this specific example, which we will refer to as Example 3, can be seen in Figure 3.9.

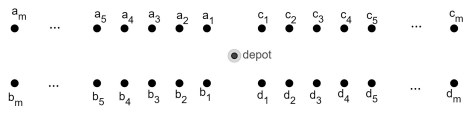

Figure 3.9: Example 3: An example of Theorem 3.2.2 failing with more than 1 vehicle where the cost function satisfies the triangle inequality

Observe that the jobs $c_1, c_2, \ldots, c_m$ and $d_1, d_2, \ldots, d_m$ are all A.M. jobs and the jobs $a_1, a_2, \ldots, a_m$ and $b_1, b_2, \ldots, b_m$ are all P.M. jobs. Let the distance from the depot to jobs c_m and d_m be greater than the sum of the length of the arcs: $(c_1, d_1),\ (c_2, d_2),\ (c_3, d_3),\ \ldots,\ (c_m, d_m)$. Let the distance from the depot to jobs $a_m,\ b_m,\ c_m,\ d_m$ be 1 and the distance between any two jobs be negligible, which we designate ϵ.

The optimal solution for this deployment of jobs will use only 1 vehicle (the other $n-1$ vehicles will remain at the depot) and perform one "loop" as seen in Figure 3.10. The calculation of the cost of the optimal schedule is shown below:

$$\begin{aligned} A_{OPT} &= \text{cost}(\textit{depot}, c_1, c_2, \ldots, c_m, d_m, d_{m-1}, \ldots, d_1) \\ P_{OPT} &= \text{cost}(b_1, b_2, \ldots, b_m, a_m, a_{m-1}, \ldots, a_1, \textit{depot}) \\ C_{OPT} &= \text{cost}(d_1, b_1) \end{aligned}$$

This is schedule is represented Figure 3.10; where the red path is the schedule for the A.M. jobs (A_{OPT}), the blue path is the schedule for the P.M. jobs (P_{OPT}), and the green path is the connection between the A.M. and P.M. schedules (C_{OPT}).

Now, consider how the time slicing heuristic algorithm would service these jobs. Since the time slicing heuristic has n vehicles to use, two vehicles can be deployed (the other $n-2$ vehicles will remain at the depot) to service the A.M. jobs where job d_m is the last job services by vehicle 1 and job c_m is the last job serviced by the vehicle 2 in the algorithm's A.M. schedule. Similarly, the algorithm will use these same two vehicles to service the P.M. jobs where job b_m is the last job services by vehicle 1 and job a_m is the last job serviced by the vehicle 2 in the algorithm's P.M. schedule, as seen in Figure 3.11. Unfortunately, this deployment of vehicles in this way results in large connection costs. The calculation of the cost of the heuristic schedule is shown below:

$$
\begin{aligned}
A_{TSH} &= A_{TSH,1} + A_{TSH,2} &&= \text{cost}(depot, d_1, d_2, \ldots, d_m) + \text{cost}(depot, c_1, c_2, \ldots, c_m) \\
P_{TSH} &= P_{TSH,1} + P_{TSH,2} &&= \text{cost}(b_1, b_2, \ldots, d_m, depot) + \text{cost}(a_1, a_2, \ldots, a_m, depot) \\
C_{TSH} &= C_{TSH,1} + C_{TSH,2} &&= \text{cost}(d_m, b_1) + \text{cost}(c_m, a_1)
\end{aligned}
$$

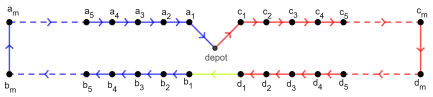

Figure 3.10: Optimal solution to Example 3.

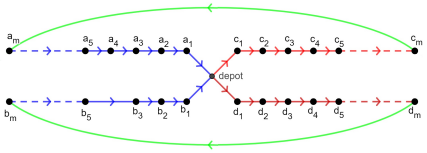

Figure 3.11: Heuristic solution Example 3.

If we compare the cost of the A.M. and P.M. schedules for time slicing heuristic to that of the optimal schedule they are the same. Observe this below:

$$
\begin{aligned}
A_{OPT} &= A_{TSH,1} + A_{TSH,2} &&= 2 \\
P_{OPT} &= P_{TSH,1} + P_{TSH,2} &&= 2
\end{aligned}
$$

However, the connection costs differ greatly. Observe this below:

$$
\begin{aligned}
C_{OPT} &= 2\epsilon \\
C_{TSH} &= 4
\end{aligned}
$$

Thus, when we compute the costs we get:

$$
\begin{aligned}
OPT &= A_{OPT} + P_{OPT} + C_{OPT} &&= 2 + 2 + 2\epsilon &&= 4 + 2\epsilon \\
TSH &= A_{TSH} + P_{TSH} + C_{TSH} &&= 2 + 2 + 4 &&= 8
\end{aligned}
$$

Now, considering the family of examples described earlier, as $n \to \infty$ implies $\epsilon \to 0$. Thus $TSH \to 2OPT$.

Theorem 3.2.3. *For vehicle routing problems using n vehicles in a metric space* $TSH \leq 2OPT$.

Proof. Consider a VRP that is solved using the time slicing heuristic. The result of this heuristic will be a set of schedules for each vehicle. The schedule for vehicle i will consist of an A.M. schedule (denoted $A_{TSH,i}$) and a P.M. schedule (denoted $P_{TSH,i}$) and a connection between the last job in the A.M. schedule and the first job in the P.M. schedule (denoted $C_{TSH,i}$). Recall, for any vehicle $i = 1, \ldots, n$, the A.M. schedule begins at the depot and then contains jobs listed in order that they will be serviced. Let us designate the last job in such a schedule as $j_{AM,last}$. And, for any vehicle $i = 1, \ldots, n$ the P.M. schedule begins with the first job serviced in the P.M. schedule, let us designate this job as $j_{PM,first}$ and then is followed by a list of jobs listed in order that they will be serviced and then ends with the depot. By the triangle inequality we know that:

$$\begin{aligned} d(depot, j_{AM,last}) &\leq A_{TSH,i} \\ d(depot, j_{PM,last}) &\leq P_{TSH,i} \\ d(j_{AM,last},\ j_{PM,first}) &\leq d(depot, j_{AM,last}) + d(j_{PM,first}, depot) \end{aligned}$$

. Thus, it is clear that for each vehicle $i = 1, ..., n$ we have:

$$\begin{aligned} C_{TSH,i} &= d(j_{AM,last}, j_{PM,first}) \\ &\leq d(depot, j_{AM,last}) + d(j_{PM,first}, depot) \\ &\leq A_{TSH,i} + P_{TSH,i} \end{aligned}$$

And, since the total travel cost of the schedule is equal to the sum of the total travel cost for each vehicle we have:

$$
\begin{aligned}
TSH &= \sum_{i=1}^{n} \left(A_{TSH,i} + P_{TSH,i} + C_{TSH,i}\right) \\
&\leq \sum_{i=1}^{n} \left(A_{TSH,i} + P_{TSH,i} + A_{TSH,i} + P_{TSH,i}\right) \\
&= \sum_{i=1}^{n} \left(2A_{TSH,i} + 2P_{TSH,i}\right) \\
&= 2\left(A_{TSH} + P_{TSH}\right) \\
&\leq 2OPT.
\end{aligned}
$$

■

3.3 Clustering Heuristic

Now, consider a heuristic algorithm that groups jobs into spatial clusters (rather than temporal clusters) and assigns one vehicle to service the jobs in a cluster, and then finds for the optimal route for each cluster. We will refer to the algorithm as the clustering heuristic (CH) algorithm.

Theorem 3.3.1. *For vehicle routing problems using n vehicles in a metric space total cost for the clustering heuristic (CH) is at most n times the optimal solution (OPT), i.e. $CH \leq n \cdot OPT$.*

Proof. Consider a vehicle routing problem with n vehicles. The algorithm begins by assigning the jobs into clusters. The maximum number of clusters has to be equal to the number of vehicles, so, without loss of generality assume there are n clusters.

It is clear that the cost for each of the vehicle routes, which we will denote CRC_i, for the clusters is at most the cost of the optimal solution for the problem because the optimal solution also services the jobs in each cluster. Thus we have:

$$CRC_i \leq OPT.$$

And, clearly the total cost for the clustering heuristic is:

$$CH = \sum_{i=1}^{n} CRC_i \leq n \cdot OPT.$$

■

To demonstrate that this bound is tight consider the situation where there are k jobs and the distance between any two jobs is ϵ and the distance between the any job and the depot is 1. We will call this situation Example 4, which can be seen Figure 3.12 The optimal solution will use one vehicle (the other $n-1$ vehicles will

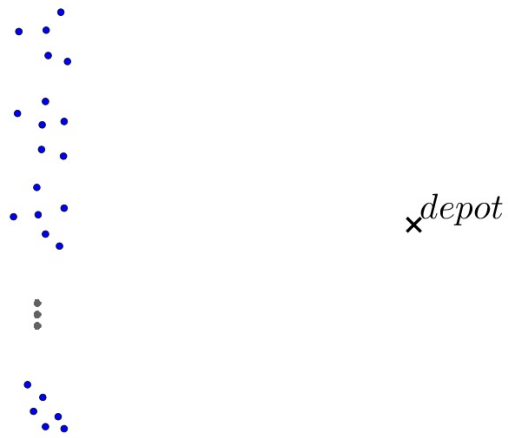

Figure 3.12: Example 4: A tight bound example for Theorem 3.3.1

remain at the depot) that one vehicle will leave the depot and service all the jobs in each cluster before returning to the depot. This schedule can be seen in Figure 3.13. The cost of this schedule is clearly $2 + (k-1)\epsilon$.

The clustering heuristic will create one cluster for each available vehicle and assign one vehicle to each cluster. Thus each of the n vehicles is assigned a cluster of jobs to service, resulting in n clusters. Each vehicle will leave the depot and service all the jobs in its assigned cluster and then return to the depot. This

schedule can be seen in Figure 3.14. The cost of this schedule is clearly $2n + (k - n)\epsilon$. Recall that the cost of the optimal solution is $OPT = 2 + (k - 1)\epsilon$

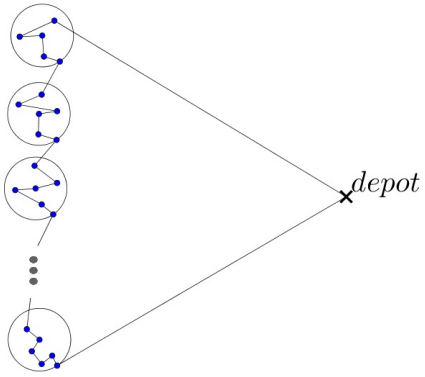

Figure 3.13: Optimal solution to Example 4.

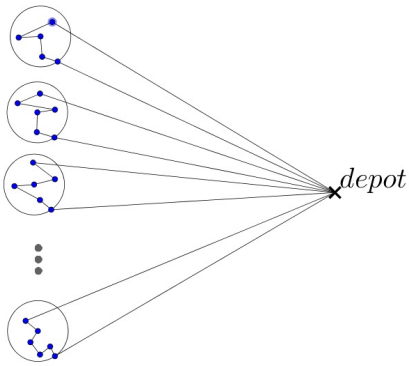

Figure 3.14: Heuristic solution Example 4.

and the cost of the clustering heuristic is $CH = 2n + (k - n)\epsilon$. If we allow $\epsilon \to 0$ we have that $OPT = 2$ and $CH = 2n$, thus we have that $CH = n \cdot OPT$ or that the cost of the clustering heuristic solution is n the cost of the optimal solution.

3.4 Conclusion

Clearly from the bounds determined in Theorem 3.2.2, Theorem 3.2.3, and Theorem 3.3.1 the time slicing approach and clustering approach show some potential as effective heuristic algorithms. However, we have not implemented an actual algorithm here, but rather considered two possible approaches for a heuristic algorithm that incorporates the use of an exact solver to find the optimal solution to sub-problems and 'stitch' those optimal sub-problem solutions together. A first

extension to this work would be implement such an algorithm and conduct empirical testing on the algorithms to determine the practical performance.

Based on the approximation bounds developed in Theorem 3.2.2, Theorem 3.2.3, and Theorem 3.3.1 the TSH shows more promise as it, in the worst case scenario, provides a tighter bound than the clustering approach. Additionally, the clustering approach leaves several nontrivial questions unaddressed. For example we have not touched on "how the jobs assigned to a cluster?", we mention doing this based on location of the jobs, but proximity alone does not guarantee that a single vehicle may feasibly service the jobs in a given cluster inside their time windows. Nor do we address "how many clusters would be optimal?", we just mention using n clusters (one per available vehicle), but one can imagine a situation where route costs could improve dramatically by reducing the number of clusters formed. Also, the motivating situation for this exploration was a situation where jobs might be canceled or added in real time. The clustering approach seemingly would add a new job to a cluster based on geography, however that could easily create an infeasible route for the vehicle already assigned to that cluster. Contrast this with the time slicing approach, when the time slice is created and an exact solver is employed the solver will make many of these decisions. The solver would optimally assign jobs to vehicles for that time slice so there is no need to pre-group jobs based on location. The solver will determine the optimal number of vehicles to use to service the jobs in the time slice, and since jobs are not pre-assigned to a vehicle (as in the clustering algorithm) an exact solver can be re-run if there are jobs that are canceled or added to a given time slice. The first major question with the time slicing approach is "how many time slices should be used?" Based on Theorems 3.2.2 and 3.2.3 we would want to minimize the number of slices since we know that the connection costs between the time slices account for the difference in the

heuristic and optimal solution. However, exact solvers can be slower than heuristics so reducing the problem size given to the exact solver would improve running time, but possible sacrificing the quality of the solution. A second major question would be "after the time slicing heuristic has determined the vehicle routes is there a technique to use that would result in a net cost savings by trading an increase in the exact solution for the time slice with the reduction of connection costs between the time slices?" the answer to either of these questions are nontrivial and would be an obvious extension of this work.

CHAPTER FOUR

RECONSTRUCTION OF SHORTEST PATHS FROM PATH SEGMENTS

4.1 Introduction

In 2020 Bani Hani [60] proposed a security protocol for autonomous vehicle communications. Autonomous vehicles communication could take place between neighboring autonomous vehicles to share information to aid in collision prevention such as speed, location, route, etc. Additionally, autonomous vehicle communication could also take place between third parties in order to plan routes, obtain specific GPS locations, monitor real time traffic conditions for route adjustments, or other location based services. Any special identifications associated to a vehicle, such MAC addresses, IP addresses, or certificates could allow an outside observer to associate it with a specific vehicle. To protect the privacy of a vehicle's identity, the protocol presented calls for the creation of a pseudo-ID for each vehicle and *mixed zones* at specific geographic locations. At these mixed zones vehicles silence their communications and whenever two or more vehicles occupy a mixed zone concurrently their pseudo-IDs are changed. A vehicle's path from the start of a route to a mixed zone, mixed zone to mixed zone, and from mixed zone to end of route is called a *trajectory*. To conduct a privacy attack on an autonomous vehicle an outside observer would like to take the trajectory information and identify the original pseudo-ID of each vehicle. Since one of the most appealing factors of autonomous vehicles is their efficiency, that is they can be programmed to take the cheapest route (distance, time, etc), we will assume that these vehicles move by traveling on a shortest path. So, this privacy attack seeks to identify the original pseudo-ID for each vehicle along its entire trajectory. This can be modeled as a

graph theory problem of reconstructing a set of shortest paths (one for each vehicle) from a collection of segments.

In this chapter we will describe in greater detail the autonomous vehicle security protocol for the communication mentioned above, the formulation of this to a graph theory problem, and propose a polynomial time algorithm to correctly reconstruct a collection of shortest paths from their segments, under mild assumptions.

4.2 Motivating Application

Under many proposed frameworks, for autonomous vehicles to function optimally the vehicle information (such as location, speed, direction traveling, etc.) is shared between vehicles, making it easier to safely navigate the roadways. However, we'd like to protect the privacy of these vehicles and keep them safe from malicious behavior by making it difficult for an outside observer to determine which signal is associated with a particular vehicle. One security measure that can be employed is the use of *pseudo-IDs*. Using this security measure the vehicle's permanent ID is hidden and replaced with a temporary ID called a pseudo-ID. Furthermore, these pseudo-IDs can be replaced (or even exchanged between vehicles) multiple times along any trip to further obfuscate a vehicle's permanent ID. One proposed method of replacing these pseudo-IDs would be to designate mixed zones, a specific geographical area where vehicles enter and silence their wireless communications for a period of time in order to change their pseudo-IDs, so whenever two or more vehicles are present in the same mixed zone at the same time they randomly alter their pseduo-IDs. In [60] Bani Hani defines the legs of a vehicle's journey from source to mixeds zones, mixed zone to mixed zone, and mixed zone to final destination as trajectories.

It's worth reiterating that a significant advantage of autonomous vehicles is the efficiency gained in travel using the vehicle's ability to travel the cheapest route to its destination. For the sake of this discussion we will assume the objective to be minimized by the autonomous vehicle is distance traveled. That is, we will assume that the autonomous vehicle's software will choose the shortest possible route from source to destination. That is, we will be assuming that the vehicles are designed to travel the shortest possible route through the road network as part of the path planning phase of the automated vehicle system model.

We can assume the perpetrator of a vehicle identity attack has detailed information about the road network the vehicles are traversing such as location of mixed zones and the edge weight (length) of the shortest paths between any two locations in this network. Additionally, we can assume we have the trajectory information about the vehicles traversing the network such as the number of vehicles in operation, the starting locations of these vehicles or *sources*, the ending location of these vehicles or *sinks*, and the time at which any vehicle enters or departs a mixed zone, starting location, or ending location. Though the perpetrator would not be given which starting location and ending location pair representing an actual vehicle route.

The perpetrator's goal would be to re-assemble the actual vehicle trajectories in order to ascertain the vehicle's pseudo-ID at any point along its route. If we were able to accomplish such a reconstruction we could (in the best case scenario) track the vehicle pseudo-ID as it changes at each mixed zone and thereby ascertain the true identity of the vehicle at any stage of its journey.

4.3 Problem Formulation and definitions

To model this problem as a graph theory problem, the road network can be represented by a graph $G = (V, E)$ where V represents the set of vertices corresponding to the intersections in the road network map and the starting and ending locations of the vehicles, E represents the set of undirected edges corresponding to the road(s) connecting two adjacent intersections and/or vehicle locations, $d_G(e)$ represents the weight (length) of edge $e \in E$ in graph G, and $d_G(P_{a,b})$ represents the weight (length) of the shortest path from $a \in V$ to $b \in V$ in G. The trajectories of the vehicles of each of the n vehicles are represented by a starting location, an ending location, and the roads that are traversed by the vehicles. The starting location of each of the n vehicles can be represented by a set of vertices $S = \{s_1, s_2, \ldots, s_n\} \subset V(G)$. The ending location of each of the n vehicles can be represented by a set of vertices $T = \{t_1, t_2, \ldots, t_n\} \subset V(G)$. The vehicle trajectories observed from the n vehicles can be represented by a multiset of edges $\mathcal{E} = \{e_1, e_2, \ldots, e_m\}$, where each $e_i \in \mathcal{E}$ is a edge in the graph G. It is worth noting here that it possible for two different vehicle trajectories to use the same edge in G, thus $\mathcal{E}$ can contain multiple copies of an edge $e_i \in E(G)$. Our goal as the perpetrator, then would be to construct a feasible set of n shortest paths in G, $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ by concatenating a subset of "compatible edges" where $S_i \subset \mathcal{E}$. We should note here that this set $\mathcal{S}$ need not be unique, however we will be particularly interested in when $\mathcal{S}$ is unique.

We say that $\mathcal{S}$ is *feasible* if:

- Every start vertex $s_j \in S$ is the start of some path $S_i \in \mathcal{S}$.
- Every end vertex $t_k \in T$ is the end of some path $S_i \in \mathcal{S}$.
- Every edge $e_l \in \mathcal{E}$ is an edge in some path $S_i \in \mathcal{S}$.
- Every path $S_i \in \mathcal{S}$, is a shortest (s_j, t_k)-path in G.

Let $e_i, e_j \in \mathcal{E}$ where $e_i = (a, c)$ and let $e_j = (d, b)$. We say that two edges, e_i and e_j are *compatible edges* if:

- The destination of the previous edge, e_i, is the start of the next edge, e_j. That is vertex c and vertex d are the same vertex.
- The (a, b)-path resulting from the concatenation of the edge e_i with the edge e_j results in a shortest (a, b)-path in G. That is $d_G(e_i) + d_G(e_j) = d_G(P_{a,b})$ where $d_G(P_{a,b})$ is the length (or distance) of a shortest path from a to b in G.

Consider a shortest (a, d)-path, P_i, formed by the concatenation of compatible edges $e_i = (a, c)$ and $e_j = (c, d)$ and a shortest $(d - b)$-path, P_j, formed by the concatenation of compatible edges $e_k = (d, f)$ and $e_l = (f, b)$ and assume edges e_j and e_k are compatible edges. Despite e_i being compatible with e_j, e_j being compatible with e_k and e_k being compatible with e_l the concatenation of paths P_i and P_j need not result in a shortest (a, b)-path in G. Consider the graph in Figure 4.1 where edges $e_i,\ e_j,\ e_k$ and e_l all have length 2 and edge e_m has length 7. Thus motivating the need for a new definition. Let P_i be a path with initial vertex a and

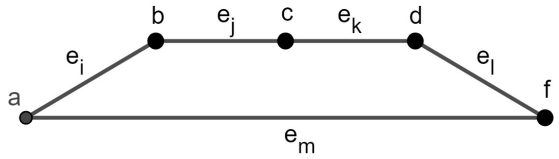

Figure 4.1: Example of concatenation of shortest paths do not results in a new shortest path.

terminal edge e_i and let P_j be a path with initial edge e_j and terminal vertex b. We say that P_i and P_j are *compatible segments* if:

- e_i and e_j are compatible edges.
- The (a, b)-path resulting from the concatenation of P_i and P_j results in a shortest path from (a, b)-path in G.

In order to accomplish this reconstruction of shortest paths we will create a multigraph which we will denote $H = (V, \mathcal{E})$ which contains only the trajectory information for the vehicles, i.e. $V(H) = S \cup T \cup M$ where $M = \{a, b \in V(G) \backslash (S \cup T) : e_i \text{ is incident to } a \text{ and } b, \ \forall \ e_i \in \mathcal{E}\}$ and $E(H) = \mathcal{E}$. And since H contains only the information for the trajectories traveled by the vehicles in the network we will refer to H as the *trajectory network*. Recall that each of the shortest paths we seek to reconstruct must originate at a vertex $s_i \in S$ (i.e. a source) and since $S \subset V(H)$ we will refer to S as the set of *source vertices*. Likewise, each of the shortest paths we seek to reconstruct must terminate at a vertex $t_i \in T$ (i.e. a sink) and since $T \subset V(H)$ we will refer to T as the set of *sink vertices*. Additionally, we noted above that the mixed zones are geographic locations where vehicle communication is silenced in order to change pseudo-IDs, the natural choice for such mixed zones are intersections of roadways. And since by definition M contains only vertices in G that represent intersections of roads we will refer to M as the set of *mixed zones vertices*. Note that this construction of H allows us to check the feasibility of $\mathcal{S}$ by simply checking that the graph formed by the shortest paths in $\mathcal{S}$ covers H. However, we must have G to check compatibility since a shortest path in the trajectory network H need not be a shortest path in the road network G.

Here we will include a discussion of the specific parameters we will consider for the remainder of this analysis. First, we will be discussing how much information we are given about the sources and sinks for the shortest path reconstruction problem. Consider the situation where you are given only a list of trajectories and given no indication of which vertices are the start of a shortest path

(source) and which vertices are the end of some shortest path (sink). For any shortest path we know that the start vertex of that path has in-degree equal to zero and out-degree equal to one, the internal vertices of that path have in-degree equal to one and out-degree equal to one, and the last vertex on the path has in-degree equal to one and out-degree equal to zero. If we assume that no vertex can be both a source and a sink we can identify sources by looking at the difference between the in-degree and out-degree. Let $k = \text{out-degree} - \text{in-degree}$. If $k > 0$, then that vertex is a source and is used in k shortest paths. If $k < 0$, then that vertex is a sink and used in $|k|$ shortest paths. And if $k = 0$, then that vertex is neither a source nor a sink. However, if vertices are allowed to be both sources and sinks it is no longer possible to determine the sources and sinks given only trajectories. Consider a vertex that is used exactly once a source and exactly once as a sink. If we calculate k for this vertex would get $k = 0$ indicating that this vertex is neither a source nor a sink (which contradicts the assumption). Thus we will be working with the assumption either no vertex can be both a source and a sink or that the sources and sinks for the shortest paths are known.

It is worth mentioning here that the above graph description does not take into account time and we will not be considering time as a factor in this analysis. If we were to include time, one obvious place where the inclusion of time would impact this formulation is that the compatibility of any path through this graph would also need to be checked against the actual time constraints on the given trajectories. For example, edges (v_i, v_j) and (v_j, v_k) may appear compatible, however if the time stamp associated with the trajectory corresponding to the edge (v_i, v_j) is later than the time stamp associated with the trajectory corresponding to the edge (v_j, v_k) clearly a vehicle could not traverse from v_i to v_j to v_k. Thus we would need to include a third criterion to our definition of *compatible* stated previously.

Specifically, "The time stamp for the vehicle trajectory which corresponds to the previous edge P_i, overlaps with the time stamp for the vehicle trajectory corresponding to the next edge P_j at the mixed zone corresponding to the end of edge P_i."

One could consider a successful privacy attack for autonomous vehicles to have a variety of meanings. A fully successful attack would reveal the true vehicle identity for all vehicles at every segment of its trip through the observable road network. We will refer to such a solution as an *unambiguous complete solution*. One type of partially successful solution would reveal the true identity of every vehicle at its sink, but not along every segment of its shortest path through the observable road network. We will refer to such a solution as an *ambiguous complete solution*. Another type of partially successful solution would reveal the true identity of some (but not all) of the vehicles along some of the segments of its shortest path through the observable road network. We will refer to such a solution as an *ambiguous incomplete solution*.

Figure 4.2 illustrates an example where an unambiguous complete solution, can be found. In Figure 4.2 each edge has weight equal to one, edges from $E(G)\backslash E(H)$ are dashed, and edges from H are solid. Suppose a vehicle A travels from location u_1 to a mixed zone m and another vehicle B travels from location u_2 to the same mixed zone m, arriving at the same time as the first vehicle. Then from the mixed zone m one vehicle proceeds to location v_1 and the other vehicle proceeds to location v_2, as pictured below. However, the distance from location u_1 to location v_2 in G is less than the sum of the distances from location u_1 to the mixed zone m and the distance from the mixed zone m to location v_2 so we can conclude that the vehicle A could not have traveled to v_2 because the route it took would violate its shortest path assumption. Therefore we can conclude that vehicle A must

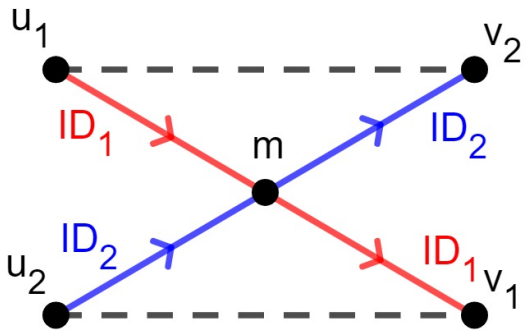

Figure 4.2: Example of an unambiguous complete solution.

have traveled to v_1 since the path of vehicle A must be a shortest path and (u_1, m, v_2) is not a shortest path from u_1 to v_2. So we can conclude that vehicle A travels the path (u_1, m, v_1) and vehicle B travels the path (u_2, m, v_2) (represented by the red and blue paths respectively in Figure 4.2), giving us an unambiguous complete solution.

Figure 4.3 illustrates an example where an unambiguous complete solution does not exist but where an ambiguous complete solution can be found. In Figure 4.3 each edge has weight equal to one, edges from $E(G)\backslash E(H)$ are dashed, and edges from H are solid. Suppose that vehicle A travels from location u_1 to location v_1 and passes through mixed zone m_1, and mixed zone m_2. Thus the edges traveled by vehicle A are $(u_1, m_1), (m_1, m_2)$ and (m_2, v_1). Now suppose that vehicle B travels from location u_2 to location v_2 and passes through mixed zone m_1 and the mixed zone m_2. Thus the edges traveled by vehicle B are $(u_2, m_1), (m_1, m_2)$, and (m_2, v_2). Observe that the shortest path from location u_1 to location v_2 does not contains the edge (m_1, m_2) and that the shortest path from location u_2 to location v_1 also does not contains the edge (m_1, m_2). So we know that vehicle A travels the

edges (u_1, m_1) and (m_2, v_1) and vehicle B travels the edges (u_2, m_1) and (m_2, v_2) (represented by the red and blue edges respectively). However, since both vehicles are traveling on edge (m_1, m_2) at the same time we cannot differentiate vehicle A from vehicle B while traveling the edge (m_1, m_2) (represented by the black undirected edges). Since we know which pseudo-ID belongs to vehicle A and vehicle B at the end of their paths this is a complete solution, but because we do not know the identity of the vehicles as they traverse the edge (m_1, m_2) this solution is ambiguous. Figure 4.4 illustrates an example of an ambiguous incomplete solution.

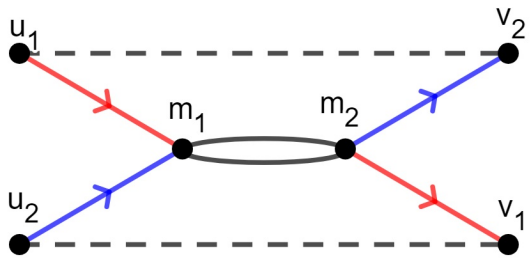

Figure 4.3: Example of an ambiguous complete solution.

In Figure 4.4 each edge has weight equal to one, edges from $E(G)\backslash E(H)$ are dashed, and edges from H are solid. Suppose vehicle A travels from location u_1 to mixed zone m, vehicle B travels from location u_2 to mixed zone m, and vehicle C travels from location u_3 to mixed zone m at the same time. From the mixed zone m the vehicles then proceed to location v_1, v_2, or v_3. We observe that the shortest path from u_1 to v_3 does not include (u_1, m) so we know that vehicle A cannot travel on the edge (m, v_3), but it could still use use edge (m, v_2) or (m, v_1). Similarly, we know that vehicle C cannot travel on the edge (m, v_1), but it could still use edge (m, v_2)

or (m, v_1). And since the shortest path from u_2 to location v_1, v_2, and v_3 includes edge (u_2, m) vehicle B could use edges $(m, v_1), (m, v_2)$ or (m, v_3). In this case we can identify the vehicle identity on some of the edges it's impossible to determine the IDs for any vehicle at their final location. Thus, this is an ambiguous incomplete solution. Clearly unambiguous complete and ambiguous complete solutions seem to

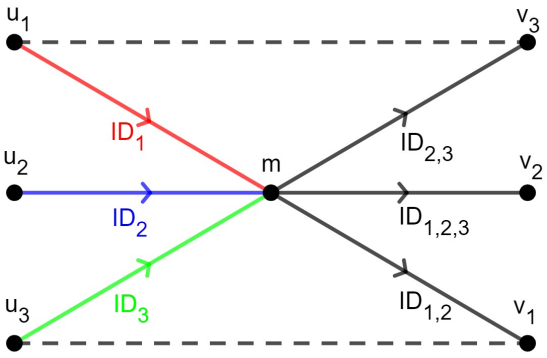

Figure 4.4: Example of an ambiguous incomplete solution.

be the most desirable outcomes as every vehicle's identity is determined by the end of its route through the road network. This result translates into the graph theory problem of correctly pairing the beginning (a source node) and end (a sink node) of each shortest path. The unambiguous complete solution has the additional feature of obtaining the identity for each vehicle as it exits every mixed zone, this translates to the graph theory problem of uniquely assigning every edge trajectory to some shortest path, thus reconstructing all paths taken by the all vehicles.

As observed in Figure 4.4, there can be cases when there is more than one way to feasibly pair the sources and sinks. We call the set of sinks that can feasibly

form shortest paths with a source u using only the edges in H the *set of feasible pairings for* u which we denote $feas(u)$. For example, in Figure 4.4 the set of feasible pairings for vertex u_1 is $\{v_1, v_2\}$ or $feas(u_1) = \{v_1, v_2\}$. If there is only one sink that can be feasibly paired with a source u we say that u has a *unique feasible pairing*. For example, in Figure 4.3 vertex u_1 has a unique feasible pairing because $feas(u_1) = \{v_1\}$. We will refer to any feasible solution to the source-sink pairing as a *pair solution*. For example in Figure 4.4 both (u_1, v_2) and (u_1, v_1) are both pair solutions. And if there is only one such pairing possible we call this a *pair-unique solution*. For example, in Figure 4.3 (u_1, v_1) is a pair-unique solution because $feas(u_1) = \{v_1\}$.

Additionally, there can be cases when there is more than one way to feasibly assign all edge trajectories to reconstruct shortest paths. For example in Figure 4.4 $\mathcal{S} = \{(u_1, m, v_1), (u_2, m, v_2), (u_3, m, v_3)\}$, $\{(u_1, m, v_2), (u_2, m, v_1), (u_3, m, v_3)\}$, $\{(u_1, m, v_1), (u_2, m, v_3), (u_3, m, v_2)\}$ are all feasible solutions for $\mathcal{S}$. We will refer to any such feasible partition of edges to reconstruct the shortest paths an *edge solution*. For example in Figure 4.4 $\{(u_1, m, v_1), (u_2, m, v_2), (u_3, m, v_3)\}$, $\{(u_1, m, v_2), (u_2, m, v_1), (u_3, m, v_3)\}$, $\{(u_1, m, v_1), (u_2, m, v_3), (u_3, m, v_2)\}$ are each edge solutions. And if there is only one such edge assignment possible we call this an *edge-unique solution*. For example in Figure 4.2 $\{(u_1, m, v_1), (u_2, m, v_2)\}$ is the only edge solution to Figure 4.2 so it an edge-unique solution.

Observe that it is possible to have a pair-unique solution that is not edge-unique. Consider the example from Figure 4.5 where each edge has weight equal to 1, edges from $E(G) \backslash E(H)$ are dashed, and edges from H are solid. The only feasible source-sink pairing is u_1 with v_1 and u_2 to v_2 since there is a shorter path in G from u_1 to v_2 and u_2 to v_2 that is not possible to take using the edges in H. However, we can see that there there are two possible ways to assign the edge

trajectories that both result in feasible shortest paths from u_1 to v_1 and u_2 to v_2 as demonstrated in Figure 4.6. Observe that it possible to have an edge-unique

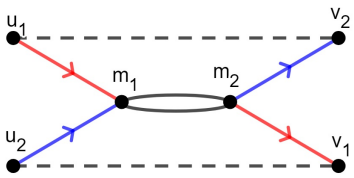

Figure 4.5: Example of a graph with a pair-unique solution, but no edge-unique solution.

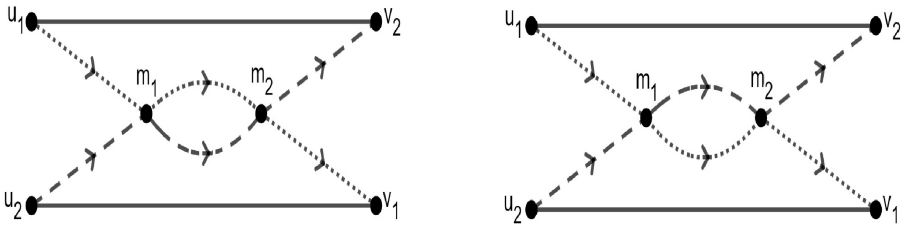

Figure 4.6: Two edge solutions to a graph with edge unique solution and no pair unique solution.

solution that is not pair-unique. Consider the graph in Figure 4.7 in which the edge trajectories from E_H are represented with directed edges and other (relevant) shortest paths from the graph G are represented with undirected edges. Here we can see that u_1 cannot be paired with v_3 (because the shortest path from u_1 to v_3

in G are not edges in H), u_2 cannot be paired with v_1 (because the shortest path from u_2 to v_1 in G are not edges in H), and u_3 cannot be paired with v_2 (because the shortest path from u_3 to v_2 in G are not edges in H). Thus, it is possible to pair u_1 with v_1 or v_2, u_2 with v_2 or v_3, and u_3 with v_1 or v_3, which implies that this graph is not pair-unique. When we select the pairing of u_1 to v_1, u_2 to v_2, and u_3 to v_3 we get a feasible solution pictured in Figure 4.8. However if we use the pairing u_1 with v_2, u_2 with v_3, and u_3 with v_1 we get the infeasible solution shown in figure 4.9. We note this solution is infeasible because not every trajectory edge is assigned to some shortest path, specifically the trajectory edges
$(m_2, m_1),\ (m_3, m_2),$ and (m_1, m_3). Leaving the single feasible edge solution shown in Figure 4.8. Note that the edges $(m_2, m_1),\ (m_3, m_2),$ and (m_1, m_3) are special.

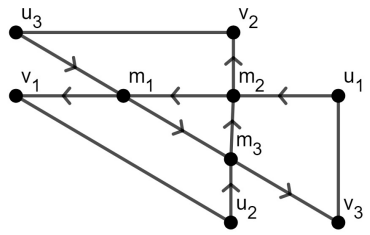

Figure 4.7: Example of a graph with edge unique solution, but no pair unique solution.

We observe the source-sink pairs for the above graph are $feas(u_1) = \{v_1,\ v_2\}$; $feas(u_2) = \{v_2,\ v_3\}$; and $feas(u_3) = \{v_1,\ v_3\}$. Note that the only edges that can precede (m_2, m_1) on a path are (u_1, m_2) or (m_3, m_2), the only edges that can precede (m_3, m_2) on a path are (u_2, m_3) and (m_1, m_3), and the only edges that can precede (m_1, m_3) on a path are (u_3, m_1) and (m_2, m_1). We will refer to edges, such

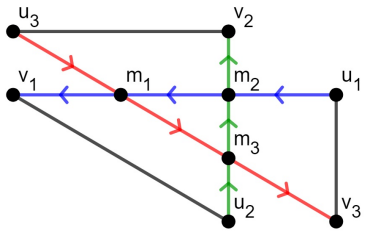

Figure 4.8: Edge-unique/pair solution to the example for a graph with edge unique solution, but no pair unique solution.

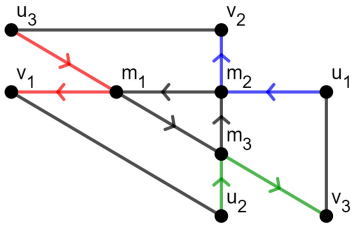

Figure 4.9: A second pair solution example for a graph with edge unique solution, but no pair unique solution.

as (m_2, m_1), (m_3, m_2), and (m_1, m_3), that are traversable from only one source as *source unique edges*.

4.4 Results

We will focus our results on the situation where an unambiguous complete solution is possible and we will use these results to develop an algorithm that can be used to reconstruct the short paths in unambiguous complete solution. We begin by looking at the degree of the source and sink vertices in H.

Proposition 4.4.1. *A shortest path reconstruction problem with an unambiguous complete solution induces a trajectory graph H, $\delta^-(s) = 0$, $\delta^+(s) = 1$, $\delta^-(t) = 1$, $\delta^+(t) = 0$ for every $s \in S$ and $t \in T$.*

Proof. Let G be the graph representing a road network for a shortest path reconstruction problem. Let H be the graph induced by the trajectories for a shortest path reconstruction problem with a unambiguous complete solution. As we have defined the shortest path reconstruction problem we do not allow sources and sinks to also act as mixed zones so we know immediately no edges can enter any source in H and no edges can leave any sink in H giving us $\delta^-(s) = 0$ for every $s \in S$ and $\delta^+(t) = 0$ for every $t \in T$.

As we have defined the shortest path reconstruction problem we do not allow more than one shortest path to use the same source or sink. Assume, for the sake of contradiction, there are more than one edge in H leaving some $s^* \in S$, then there there must be two shortest paths, P_1 and P_2 that begin at s^* and terminate at vertices t_1 and t_2 respectively. In that case we can get a solution were vehicle 1 travels P_1 and vehicle 2 travels P_2 or vice versa. This contradicts the assumption of an unambiguous complete solution. We can make a similar argument if more than one edge in H enters some $t^* \in T$. ■

Now that we know that concatenating compatible path segments can produce a shortest path we'd like to investigate the instances where we will be guaranteed a complete unambiguous solution. We will begin the obvious situation where every mixed zone node in $V(H)$ has an in degree of one and an out degree of one. Or, in other words, none of the vehicle paths overlap. Consider the following:

Lemma 4.4.2. *If every shortest path represented by vehicle routes have no vertices in common, then the edges making up those shortest paths can be reassembled to yield a unambiguous complete solution.*

Proof. Each path can be assembled naively by beginning with a source and concatenating compatible edges until a sink is reached. Since the paths have no

vertices in common each edge will have only one possible edge compatible with it until an edge with an endpoint in T (that is an endpoint that is a sink). So each path can be reconstructed by starting with a source and concatenating edges until a sink is reached and then moving to the next path until all edges are used. ■

To show that if we remove the requirement that the shortest paths have no vertices in common it will be easy to see that we lose the guarantee of a unambiguous complete solution. Consider a specific example where these shortest paths have one vertex in common. Take, for example, the graph $G = (\{A, B, C, D, E\}, \{(A, C), (B, C), ((C, D), (C, E)\})$ as represented in Figure 4.10 where all the edge weights are equal and the paths $(A, C, D),\ (A, C, E),\ (B, C, E),$ and (B, C, D) are all shortest paths in G. Clearly

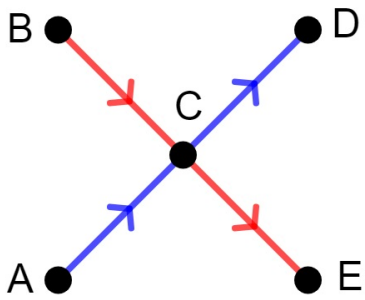

Figure 4.10: Example of a graph with ambiguous shortest paths

there is no way, when assembling these edges from the shortest paths to know if the edge (A, C) should be paired with edge (C, D) or with edge (C, E). Since both pairings result in a shortest path and the edge (B, C) could also be paired with

either edge to also form a shortest path. So, in this case there is no unique solution to the shortest path reassembly problem, i.e. this situation is ambiguous incomplete.

Note that the presence of shared vertices in the shortest paths does not preclude the possibility of an unambiguous complete solution. For example consider Figure 4.11, suppose there is an edge in the road network that permits travel from vertex B to vertex D directly and that distance is less than the distance of the path from vertex B to vertex D that also pass through vertex C. In this graph we could eliminate the possibility of the path from vertex B to vertex D being a shortest path because there is another path in the road network that is shorter. And, in this case there is a unique solution to the shortest path reassembly problem, making this an unambiguous complete solution. As we have stated, the best outcome is the

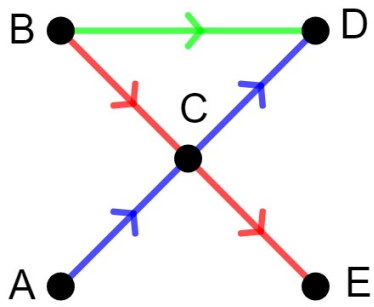

Figure 4.11: Example of a graph with unambiguous shortest paths

unambiguous complete solution. So the rest of our analysis is focused on situations where this outcome is possible. We begin with the simplest situation:

Lemma 4.4.3. *If there is an unambiguous complete solution to the shortest path reconstruction problem for two paths, then the two shortest paths have no more than one vertex in common.*

Proof. Let path A be a shortest path from node u_1 to node v_1 and let path B be a shortest path from node u_2 to node v_2. Suppose, for the sake of contradiction, that the nodes m and n both appear on path A and on path B. Let $P_{1,A}$ represent the segment of path A from u_1 to m, $P_{2,A}$ represent the segment of path A from m to n, $P_{3,A}$ represent the segment of path A from n to v_1, $P_{1,B}$ represent the segment of path B from u_2 to m, $P_{2,B}$ represent the segment of path B from m to n, and $P_{3,B}$ represent the segment of path B from n to v_2. Clearly, if we assemble segments $P_{1,A}$, $P_{2,A}$ and $P_{3,A}$ we return the shortest path A (from u_1 to v_1). However, the assembly of $P_{1,A}$, $P_{2,B}$, and $P_{3,A}$ also results in a shortest path from u_1 to v_1 since the length of the segments $P_{2,A}$ and $P_{2,B}$ must be identical. If the segment $P_{2,A}$ was shorter then path B could not be a shortest path because replacing $P_{2,B}$ with the shorter segment $P_{2,A}$ would result in a shorter path and a contradiction. Thus there is not a unique assembly of the edges of path A and path B. ■

Next, we consider a situation which an unambiguous complete solution *cannot* be found.

Lemma 4.4.4. *A shortest path reconstruction problem with a unambiguous complete solution can have no parallel edges in H.*

Proof. Let G be a graph representing a road network for a shortest path reconstruction problem. Let H be the graph induced by the trajectories for a shortest path reconstruction problem with a unambiguous complete solution. Suppose, for the sake of contradiction, that there exists parallel edges e_1 and e_2 in H. Let $\mathcal{S} = (S_1, S_2, \ldots, S_n)$ be the unambiguous complete solution. Let S_i be the

shortest path from the correct solution that includes e_1 and S_j be the correct shortest path that includes e_2. Since e_1 and e_2 are parallel we can construct S_i^* and S_j^* by exchanging e_1 and e_2 in the paths S_i and S_j. Furthermore, since the edges e_1 and e_2 are parallel the length of S_i is the same as the length of S_i^* and the length of S_j is the same as the length of S_j^*. Thus the solution containing $\mathcal{S} - \{S_i^*, S_j^*\}$ is also a feasible solution which contradicts the uniqueness of the solution. ■

Suppose we return to the idea illustrated by Figure 4.7, the graph whose feasible pairing was not unique even though there was an edge-unique solution. Suppose we consider the subgraph I_{u_1}, shown in 4.12, which is constructed using the shortest paths from the source, u_1, to its feasible sinks, v_1 and v_2. Similarly, construct I_{u_2}, and I_{u_3} also shown in 4.12. Observe that in each of these subgraphs the edges (m_2, m_1) occurs only in I_{u_1}, (m_3, m_2) occurs only in I_{u_2}, and (m_1, m_3) occurs only in I_{u_3}. Since each of these edges are trajectory edges they must be used in some shortest path. Therefore we can conclude that u_1 must be paired with v_1 because there is no other shortest path from u_1 that uses the edge (m_2, m_1) and reaches v_2. We can make a similar argument we can conclude u_2 must pair with v_2 and u_3 must pair with v_3. Giving a unique feasible pairing for H. This observation

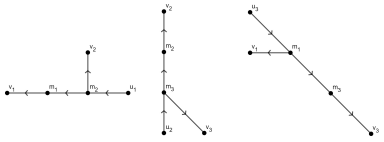

Figure 4.12: Subgraphs I_{u_1}, I_{u_2}, and I_{u_3}

necessitates the following definitions and motivates the next theorem:

- Let M_{s_i,t_j} represent the mixed zone vertices that appear on any shortest path in H from source s_i to sink t_j.
- Let E_{s_i,t_j} represent the set of edges $e \in E_H$ that appear on a shortest path from the source s_i to the sink t_j in H.
- We say an edge is a *source-unique edge* if that edge connects two mixed zone vertices, that is if $e = (m_i, m_j),\ m_i,\ m_j \in M$ and $\exists$ a unique s_i and t_j with $e \in E_{s_i,t_j}$.
- Let an *edge solution* be a set of k paths where each path starts at a unique source, ends at a unique sink, and are constructed by partitioning the trajectory edges to some path. That is, $\epsilon = \{R_1, \ldots, R_k\}$ such that each $R_i \in \epsilon$ starts at a unique $s \in S$, ends at a unique $t \in T$, and forms a partition of E_H.
- Let a *feasible pairing in* H, $T_H(s_i)$, be the set of $t_j \in T$ such that $d_G(s_i, t_j) = d_H(s_i, t_j)$ for a specific $s_i \in S$.
- Let a *source-sink subgraph*, which we denote as A_{s_i}, be a graph which describes only the shortest paths from the source s_i to the sinks in its feasible pairing. That is, $A_{s_i} = (\overline{V}, \overline{E})$ where $\overline{V} = \{s_i\} \cup T_{s_i} \cup \left(\bigcup_{t_j \in T_{s_i}} M_{s_i,t_j} \right)$ and $\overline{E} = \bigcup_{t_j \in T_{s_i}} E_{s_i,t_j}$.

Theorem 4.4.5. *If a shortest path reconstruction problem has an unambiguous complete solution, then at least one of the following occurs:*

- *There exists at least one source whose set of feasible pairings contains exactly one sink.*
- *There exists at least one sink that is a member of exactly one source's set of feasible pairings.*
- *There exists at least one source-unique edge.*

Proof. Let G be a graph representing a road network for a shortest path reconstruction problem with k vehicle routes. Let H be the graph induced by the trajectories for a shortest path reconstruction problem with a unambiguous complete solution.

Suppose, for the sake of contradiction, that each source has a set of feasible pairings that contains at least two sinks, each sink is a member of at least two source's set of feasible pairings, and no edges in H are source unique.

Let s_1 be a vertex in H. From Proposition 4.4.1 we know that each source vertex can only have one edge leaving it, so there is the edge $e_1 = (s_1, m_1)$. We know m_1 is a mixed zone vertex, otherwise it would have to be a sink vertex which would make s_1 the only member of m_1's feasible pairings which is a contradiction. Since m_1 is a mixed zone vertex we know $\delta^-(m_1) = \delta^+(m_1)$ so there must be at least one edge leaving m_1. If there is exactly one edge leaving m_1 that edge is only reachable by s_1 making that edge a source unique edge, which is a contradiction. Thus $\delta^+(m_1) \geq 2$. Since all the edges in H must be edges in some shortest (s, t) any mixed zone vertex v must satisfy $\delta^-(m_1) = \delta^+(m_1)$. Thus $\delta^-(m_1) \geq 2$. Let s_2 be a source vertex that can reach m_1 in H. If the edge (s_2, m_1) exists we have found two vertices in H that can reach the same set of sinks. Otherwise, there must be some vertex m_2 that precedes vertex m_1 on the path from s_2 to m_1. Without loss of generality suppose that the edge $e_2 = (m_2, m_1)$ is on the path from s_2 to m_1. And since there are finitely many sources, using the previous argument we can find two sources, s^* and s^{**} and vertex m^* such that $e^* = (s^*, m^*)$ and $e^{**} = (s^{**}, m^*)$ are edges in H.

Since there is an ambiguous complete solution assume we can find the $k - 2$ correct source-sink pairings for the sources (other than those for s^* and s^{**}), form those correct shortest paths, and delete the edges that are assigned to these $k - 2$

shortest paths from H. We call the resulting graph H'. Note that H' contains only the two sources s^* and s^{**} and two sinks, call them t^* and t^{**}. And by construction we know H' is connected because it must contain the edges $e^* = (s^*, m^*)$ and $e^{**} = (s^{**}, m^*)$. Since every sink vertex in H that was reachable from s^* is also reachable from s^{**} (and vice versa) we can say the same is true for H' because any such paths must have m^* as the first non-source vertex. So in H' there are the edges e^*, e^{**}, and the path, $P_{H'}(m^*, t^*)$, from m^* to t^* and the path, $P_{H'}(m^*, t^{**})$, from v to t^{**}. However, we cannot say immediately if the concatenation of any of these paths are actually shortest paths, thus we have 3 cases:

case 1: In H' the set of feasible pairings for s^* and s^{**} have two sinks in common. Without loss of generality assume $T_{H'}(s^*) = T_{H'}(s^{**}) = \{t^*, t^{**}\}$.

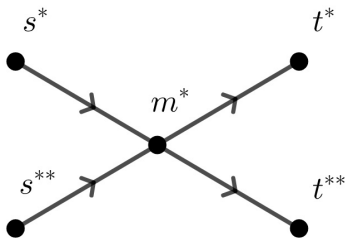

Figure 4.13: Theorem 4.6.2, case 1

From H' we can construct these two sets of source-sink paths:

solution 1: $P_{H'}(s^*, m^*) + P_{H'}(m^*, t^*)$ and $P_{H'}(s^{**}, m^*) + P(m^*, t^{**})$

solution 2: $P_{H'}(s^*, m^*) + P_{H'}(m^*, t^{**})$ and $P_{H'}(s^{**}, m^*) + P(m^*, t^*)$

And since t^* and t^{**} are both in the set of feasible pairings for s^* and s^{**} we know that above paths are all shortest paths. This implies that there are two

solutions which contradicts the assumption that the shortest path reconstruction problem has a unique solution.

case 2: In H' the set of feasible pairings for s^* and s^{**} have one sink in common. Without loss of generality assume $T_{H'}(s^*) = \{t^*, t^{**}\}$, $T_{H'}(s^{**}) = \{t^{**}\}$. In H' we know that the source s^{**} is connected to sink t^* so if $t^* \notin T_{H'}(s^{**})$ there must be a shorter path in H that could connect s^{**} to t^*. Assume, for the sake of contradiction, such a path, $P_H(s^{**}, t^*)$ exists. If $P_H(s^{**}, t^*)$ does not contain the edge e^{**}, then there must be a second edge leaving s^{**} which contradicts Proposition 4.4.1. So the path $P_H(s^{**}, t^*)$ must contain the edge e^{**}. Thus $P_H(s^{**}, t^*)$ can be formed from the concatenation of edge e^{**} and path $P_{H'}(m^*, m^{**})$ and the path $P_H(m^{**}, t^*)$ which is in H but not in H'. However, $t^* \in T_{H'}(s^*)$ so we know the path through H', $P_{H'}(s^*, t^*)$, from s^* to t^* is a shortest path in H. And since that shortest path $P_H(s^*, t^*)$ can be made from the concatenation of edge e^*, path $P_{H'}(m^*, m^{**})$, path $P_{H'}(m^{**}, t^*)$. This is a contradiction since the path $P_H(s^*, t^*)$ can also be formed by the concatenation of the edge e^*, path $P_{H'}(m^*, m^{**})$ and path $P_H(m^{**}, t^*)$ which must be shorter than $P_{H'}(s^*, t^*)$ since $d(P_H(m^{**}, t^*)) < d(P_{H'}(m^{**}, t^*))$. So case 2 cannot happen.

case 3: In H' the set of feasible pairings for s^* and s^{**} have no sinks in common. Without loss of generality assume $T_{H'}(s^*) = \{t^*\}$ and $T_{H'}(s^{**}) = \{t^{**}\}$. Since t^* is not in the set of feasible pairings for s^{**} the path $P_{H'}(m^*, t^*)$ contains a source-unique edge, which is a contradiction, or there exists a mixed zone edge on the path $P_{H'}(m^*, t^*)$ with in-degree at least two that is reachable via shortest path from another source. Without loss of generality choose such a mixed zone edge closest to t^*, we will denote this edge

$e^{***} = (m^{**}, m^{***})$ and refer the source as s^{***}. Note that this implies that t^* must be in the set of feasible pairings for s^{***}.

Now, restore H. Since this problem has an unambiguous complete solution, assume you have the correct $k-2$ source-sink pairings for all sources other than s^* and s^{***}, form the correct shortest paths, and delete the edges assigned to the $k-2$ correct shortest paths from the graph of H, call this graph H''. In H'' we have the two sources s^* and s^{***} and the two sinks t^* and t^{***}. Since we know that t^* is contained in the set of feasible pairings for both s^* and s^{***} we know we are either in case 1 or case 2 and therefore a contradiction.

Since every case results in a contradiction, the statement holds. ■

Theorem 4.4.6. *If the shortest path reconstruction problem has an unambiguous complete solution and no source has a unique feasible pairing, then there exists source-unique edges that can induce a unique feasible pairing for each source.*

Proof. Let G be the graph representing a shortest path reconstruction problem with a unique solution. Let the H be the graph induced by the trajectory edges for the shortest path reconstruction problem where the feasible pairings for H are not unique. By Theorem 4.6.1 we know that there must be a source-unique edge in H. Suppose we find this source-unique edge, $e_1 = (x_1, x_2)$, that appears only on the unique shortest path from s_1 to t_1. Suppose the (s_1, t_1) shortest path containing e_1 is not unique. Without loss of generality, suppose there are two such (s_1, t_1) shortest paths, R_1 and R_2. Since the lengths of R_1 and R_2 are equal and both paths have the same source and sink and contain the edge e_1 we know that if the subpath of R_1 from s_1 to x_1 is different from the subpath of R_2 from s_1 to x_1 then those two subpaths have the same length (we can say the same about the subpaths of R_1 and

R_2 from x_2 to t_1). If that is the case we have contradicted that there is an unambiguous complete solution. Suppose, for the sake of contradiction, the subpaths are not the same length. Without loss of generality, suppose the subpath of R_1 from s_1 to x_1 is shorter than the subpath of R_2 from s_1 to x_1. This would imply that the subpath of R_1 from x_2 to t_1 must be longer than the subpath of R_2 from x_2 to t_1. Which contradicts that R_1 and R_2 are shortest paths. Suppose we form the unique shortest path, P_1, and create the graph H_1 by deleting the edges used in path P_1 along with s_1 and t_1. Since e_1 can only be used on the unique shortest path between s_1 and t_1, we can remove the s_1, t_1 and edges used in the path P_1 from the shortest path reconstruction problem represented by H, making a new shortest path reconstruction problem (represented by H_1) also with an unambiguous complete solution. We apply Theorem 4.6.1 again and search for a source-unique edge, e_2, WLOG assume this edge is on the shortest path from s_2 to t_2. We form the unique shortest path P_2 from s_2 to t_2 and delete that information from H_1, forming a new graph H_2 which also represents a shortest path reconstruction problem with an unambiguous complete solution. We can continue this process as described until all shortest paths have been formed. And, since there are finitely many sources in a shortest path reconstruction problem this process is guaranteed to finish. ■

The previous theorem gives the outline for an algorithm to solve the shortest path reconstruction problem. We begin by discussing some pre-processing work before discussing the main body of the algorithm. First, we perform the all pairs shortest path algorithm on G and H returning distance matrices D_G and D_H respectively. Create a set of feasible pairings for each source by comparing the distance for each source to each sink in D_G to the distance in D_H and if $d_G(s,t) = d_H(s,t)$ we add the sink t to the set of feasible pairings for the source s. Then, construct the

source-sink subgraph which consists of the shortest paths from a specific source s through H to the sinks in the set of feasible pairings for that source s. Lastly, create a list for each source to hold the edges in the correct shortest path that begins at that source. See the complete algorithm outlined in Algorithm 4.1.

Algorithm 4.1: An algorithm to solve the shortest path reconstruction problem, given that a unambiguous complete solution exists

```
1.  while there exists any source specific sinks or any source specific edges
      or source unique sinks do
2.    if there exists a source specific sink t for the source s then
3.      Use the source-sink subgraph for s to construct the shortest
          path from s to t and add those edges to SP_s
4.      Delete t from all other sets of feasible pairings
5.      Delete the edges added to SP_s from E_H and all other source
          sink subgraphs
6.      Delete any edges in any source sink subgraph that have a node
          with in degree 0 (other than at a source) or out degree of 0
          (other than at a sink)
7.    end
8.    if there exists a source specific edge e for the source s then
9.      Add e to SP_s
10.     Delete e from E_H and all other source sink subgraphs
11.     Delete any edges in any source sink subgraph that have a node
          with in degree 0 (other than at a source) or out degree of 0
          (other than at a sink)
12.   end
13. end
14. if |V_H| = 0 then
15.   Return the shortest path sets
16. else
17.   Return "no unique solution"
18. end
19.
```

4.5 Summary of Work

In conclusion, we have used the autonomous vehicles privacy attack as presented by Bani Hani [60] to formulate the graph theory problem of reconstruction a collection of shortest paths from their edges. In doing so we defined the graph H created by the edges of these shortest paths. On H we defined three special features: sources whose set of feasible pairings contains exactly one sink, sinks that is a member of exactly one source's set of feasible pairings, and source-unique edge. We were able to prove Theorem 4.4.5 which guarantees that if a shortest path reconstruction problem has a unique solution, than H has at least one of the special properties mentioned prior. Using this result we proved Theorem 4.4.6 which states "if the shortest path reconstruction problem has a unique solution and no source has a unique feasible pairing, then there exists source-unique edges that can induce a unique feasible pairing for each source". This provides the framework for an algorithm which will find a unique solution if one exists (or identify that no unique solution exists) in polynomial time.

CHAPTER FIVE

CONCLUSIONS

5.1 Summary of Results

In Chapter 2 an exact solver for the VRPTW, based on the work of [59], was proposed. Some of the processing time saving methods proposed by [59] were successfully adapted and implemented, with the exclusion of test 3. Additionally, original tests were developed for the VRPTW that were found to also improve the algorithm's performance, namely the jump ahead test and the duplicate label test. However, the algorithm was not competitive in comparison to state of the art exact solvers.

In Chapter 3 two heuristic algorithms were proposed. The first was based on time slicing. This algorithm was not implemented, but it was shown that its worst case performance is at most $1.5OPT$ for n vehicles in a 1-dimensional space and at most $2OPT$ for n vehicles in a metric space. The second was based on clustering. This algorithm was not implemented, but it was shown that its performance is at most $nOPT$ for n vehicles in a metric space.

In Chapter 4 the T-PAAD attack described by [60] is formulated as a shortest path reconstruction problem. The concept of the unambiguous complete solution to the shortest path reconstruction problem is defined and we prove that a shortest path reconstruction problem must have at least one of three properties. An algorithm was presented to solve the shortest path reconstruction problem. A proof is provided guaranteeing this algorithm will terminate at the correct solution provided there is an unambiguous complete solution.

5.2 Further Research

- In Chapter 2 we found that the dynamic programming algorithm performed rather better when the number of jobs was small and briefly discussed that this algorithm could be used for route improvement as part of a heuristic algorithm. For example, a heuristic algorithm could produce non-optimal schedule for a VRPTW. To find a "better" solution one could select the route assignments for two (or more) vehicles and use the dynamic programming algorithm to determine if there is a cheaper way to assign these jobs to the two vehicles and if so find a better schedule.
- In Chapter 3 we proposed a time slicing heuristic algorithm and a clustering heuristic. We were able to determine some theoretical bounds for the worst case performance of these heuristic. It is possible that in practice these algorithms obtain better solutions, but an implementation and empirical testing would be needed.
- As mentioned in the Chapter 2 research ideas, after implementing the time slicing and clustering algorithms we could develop a route improvement step using the dynamic programming algorithm from Chapter 2. Empirical testing of the route improvement step could then be done.
- In Chapter 4 we proposed an algorithm to defeat a security protocol for a fleet of autonomous vehicles. We proved that if a unique solution exists, then this algorithm will identify the identities of each vehicle using only route information. However, the performance of this algorithm, in practice, needs to be tested. One could implement the algorithm and perform empirical testing.
- We were able to prove that the Chapter 4 algorithm will identify the identities of the each vehicle, provided an unambiguous complete solution exists, we have not thoroughly explored its performance when no unique solution exists.

One could explore methods to identify if no unique solution exists, the number of possible solutions when the solution is not unique, and use these observations to build an improved security protocol.

REFERENCES

[1] P. Toth and D. Vigo, *The Vehicle Routing Problem*. Philadelphia, PA: SIAM, 2002.

[2] G. Dantzig, D. Fulkerson, and S. Johnson, "Solution of a large-scale traveling-salesman problem," *Operations Research*, vol. 2, pp. 393–410, 1954.

[3] G. Dantzig and J. Ramser, "The truck dispatching problem," *Management Science*, vol. 6, no. 1, pp. 80–91, 1959.

[4] B. L. Golden, T. L. Magnanti, and H. Q. Nguyen, "Implementing vehicle routing algorithms," *Networks*, vol. 7, no. 2, pp. 113–148, 1977.

[5] G. Laporte, Y. Nobert, and M. Desrochers, "Optimal routing under capacity and distance restrictions," *Operations Research*, vol. 33, no. 5, pp. 1050–1073, 1985.

[6] R. Baldacci, E. Hadjiconstantinou, and A. Mingozzi, "An exact algorithm for the capacitated vehicle routing problem based on a two-commodity network flow formulation," *Operations Research*, vol. 52, no. 5, pp. 723–738, 2004.

[7] M. L. Balinski and R. E. Quandt, "On an integer program for a delivery problem," *Operations research*, vol. 12, no. 2, pp. 300–304, 1964.

[8] J. Lenstra and A. Rinnooy-Kan, "Complexity of vehicle routing and scheduling problems," *Networks*, vol. 11, pp. 221–227, 1981.

[9] G. Laporte, M. Desrochers, and Y. Nobert, "Two exact algorithms for the distance-constrained vehicle routing problem," *Networks*, vol. 14, pp. 161–172, 1984.

[10] R. Fukasawa, H. Longo, J. Lysgaard, M. P. de Aragao, M. Reis, E. Uchoa, and R. Werneck, "Robust branch-and-cut-and-price for the capacitated vehicle routing problem," *Mathematical Programming Ser. A*, vol. 106, pp. 491–511, 2006.

[11] R. Baldacci, N. Christofides, and A. Mingozzi, "An exact algorithm for the vehicle routing problem based on the set partioning formulation with additional cuts," *Mathematical Programming Ser. A*, vol. 115, pp. 351–385, 2008.

[12] D. Pecin, A. Pessoa, M. Poggi, and E. Uchoa, "Improved branch-cut-and-price for capacitated vehicle routing," *Mathematical Programming Computation*, vol. 9, pp. 61–100, 2017.

[13] D. Pecin, C. Contardo, G. Desaulniers, and E. Uchoa, "New enhancements for the exact solution of the vehicle routing problem with time windows," *Journal on Computing*, vol. 29, no. 3, pp. 489–502, 2017.

[14] A. Pessoa, R. Sadykov, and E. Uchoa, "Enhanced branch-cut-and-price algorithm for heterogenous fleet vehicle routing problems," *European Journal of Operational Research*, vol. 270, no. 2, pp. 530–543, 2018.

[15] D. Pecin and E. Uchoa, "Comparative analysis of capacitated arc routing formulations for designing a new branch-cut-and-price algorithm," *Transportation Science*, vol. 53, no. 6, pp. 1673–1694, 2019.

[16] C. Bode and S. Irnich, "Cut-first branch-and-price-second for the capacitated arc-routing problem," *Operations Research*, vol. 60, no. 5, pp. 1167–1182, 2012.

[17] M. Jepsen, B. Petersen, S. Spoorendonk, and D. Pisinger, "Subset-row inequalities applied to the vehicle-routing problem with time windows," *Operations Research*, vol. 56, no. 2, pp. 497–511, 2008.

[18] S. Ropke and J.-F. Cordeau, "Branch and cut and price for the pickup and delivery problem with time windows," *Transportation Science*, vol. 43, no. 3, pp. 267–286, 2009.

[19] R. Baldacci and A. Mingozzi, "A unified exact method for solving different classes of vehicle routing problems," *Mathematical Programming*, vol. 120, no. 2, pp. 347–380, 2009.

[20] A. Pessoa, R. Sadykov, E. Uchoa, and F. Vanderbeck, "A generic exact solver for vehicle routing and related problems," *Mathematical Programming*, vol. 183, no. 1, pp. 483–523, 2020.

[21] N. Christofides, A. Mingozzi, and P. Toth, "State-space relaxation procedures for the computation of bounds to routing problems," *Networks*, vol. 11, pp. 145–164, 1981.

[22] G. Clarke and J. W. Wright, "Scheduling of vehicles from a central depot to a number of delivery points," *Operations research*, vol. 12, no. 4, pp. 568–581, 1964.

[23] T. Gaskell, "Bases for vehicle fleet scheduling," *Journal of the Operational Research Society*, vol. 18, no. 3, pp. 281–295, 1967.

[24] P. Yellow, "A computational modification to the savings method of vehicle scheduling," *Journal of the Operational Research Society*, vol. 21, no. 2, pp. 281–283, 1970.

[25] M. Desrochers and T. Verhoog, "A matching based savings algorithm for the vehicle routing problem," *Cahiers du GERAD*, 1989.

[26] K. Altinkemer and B. Gavish, "Parallel savings based heuristics for the delivery problem," *Operations research*, vol. 39, no. 3, pp. 456–469, 1991.

[27] P. Wark and J. Holt, "A repeated matching heuristic for the vehicle routeing problem," *Journal of the Operational Research Society*, vol. 45, no. 10, pp. 1156–1167, 1994.

[28] R. Mole and S. Jameson, "A sequential route-building algorithm employing a generalised savings criterion," *Journal of the Operational Research Society*, vol. 27, no. 2, pp. 503–511, 1976.

[29] N. Christofides, A. Mingozzi, and P. Toth, "The vehicle routing problem," *Combinatorial optimization*, pp. 315–335, 1979.

[30] B. E. Gillett and L. R. Miller, "A heuristic algorithm for the vehicle-dispatch problem," *Operations research*, vol. 22, no. 2, pp. 340–349, 1974.

[31] A. Wren and J. Carr, "Computers in transport planning and operation," 1971.

[32] A. Wren and A. Holliday, "Computer scheduling of vehicles from one or more depots to a number of delivery points," *Journal of the Operational Research Society*, vol. 23, no. 3, pp. 333–344, 1972.

[33] M. L. Fisher and R. Jaikumar, "A generalized assignment heuristic for vehicle routing," *Networks*, vol. 11, no. 2, pp. 109–124, 1981.

[34] J. Bramel and D. Simchi-Levi, "A location based heuristic for general routing problems," *Operations research*, vol. 43, no. 4, pp. 649–660, 1995.

[35] B. A. Foster and D. M. Ryan, "An integer programming approach to the vehicle scheduling problem," *Journal of the Operational Research Society*, vol. 27, no. 2, pp. 367–384, 1976.

[36] D. M. Ryan, C. Hjorring, and F. Glover, "Extensions of the petal method for vehicle routeing," *Journal of the Operational Research Society*, vol. 44, no. 3, pp. 289–296, 1993.

[37] J. Renaud, F. F. Boctor, and G. Laporte, "A fast composite heuristic for the symmetric traveling salesman problem," *INFORMS Journal on computing*, vol. 8, no. 2, pp. 134–143, 1996.

[38] S. Lin, "Computer solutions of the traveling salesman problem," *Bell System Technical Journal*, vol. 44, no. 10, pp. 2245–2269, 1965.

[39] S. Lin and B. W. Kernighan, "An effective heuristic algorithm for the traveling-salesman problem," *Operations research*, vol. 21, no. 2, pp. 498–516, 1973.

[40] I. Or, *TRAVELING SALESMAN TYPE COMBINATORIAL PROBLEMS AND THEIR RELATION TO THE LOGISTICS OF REGIONAL BLOOD BANKING.* Northwestern University, 1976.

[41] J. Renaud, F. F. Boctor, and G. Laporte, "An improved petal heuristic for the vehicle routeing problem," *Journal of the operational Research Society*, vol. 47, no. 2, pp. 329–336, 1996.

[42] P. M. Thompson and H. N. Psaraftis, "Cyclic transfer algorithm for multivehicle routing and scheduling problems," *Operations research*, vol. 41, no. 5, pp. 935–946, 1993.

[43] A. Van Breedam, *An Analysis of the Behavior of Heuristics for the Vehicle Routing Problem for a Selectrion of Problems with Vehicle-related, Customer-related, and Time-related Constraints.* RUCA, 1994.

[44] G. A. Kindervater and M. W. Savelsbergh, "10. vehicle routing: handling edge exchanges," in *Local search in combinatorial optimization*, pp. 337–360, Princeton University Press, 2018.

[45] I. H. Osman, "Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem," *Annals of Operations Research*, vol. 41, pp. 421–451, 1993.

[46] M. Gendreau, A. Hertz, and G. Laporte, "A tabu search heuristic for the vehicle routing problem," *Management Science*, vol. 40, pp. 1276–1290, 1994.

[47] Y. Rochat and E. D. Taillard, "Probabilistic diversification and intensification in local search for vehicle routing," *Journal of Heuristics*, vol. 1, no. 1, pp. 147–167, 1995.

[48] P. Toth and D. Vigo, "The granular tabu search and its application to the vehicle routing problem," *Jounral on Computing*, vol. 15, no. 4, pp. 333–346, 2003.

[49] A. V. Breedam, "Improvement heuristics for the vehicle routing problem based on simulated annealing," *European Journal of Operational Research*, vol. 86, pp. 480–490, 1995.

[50] G. Dueck and T. Scheuer, "Threshold accepting: A general purpose optimization algorithm appearing superior to simulated annealing," *Journal of Computational Physics*, vol. 90, no. 1, pp. 161–175, 1990.

[51] G. Dueck, "New optimization heuristics: The great deluge algorithm and the record-to-record travel," *Journal of Computational Physics*, vol. 104, no. 1, pp. 86–92, 1993.

[52] B. Golden, E. Wasil, J. Kelly, and I. Chao, "Metaheuristics in vehicle routing," in *Fleet Management and Logistics* (T. Crainic and G. Laporte, eds.), pp. 33–56, Boston, MA: Kluwer, 1998.

[53] J.-Y. Potvin and S. Benigo, "The vehicle routing problem with time windows part ii: genetic search," *Journal on Computing*, vol. 8, no. 2, pp. 165–172, 1996.

[54] L. Chambers, ed., *Vehicle routing with time windows using genetic algorithms*. Boca Raton, FL: CRC Press, 1995.

[55] A. Colorni, M. Dorigo, and V. Maniezzo, "Distributed optimization by ant colonies," *Proceedings of the first European conference on artificial life*, vol. 142, pp. 134–142, 1991.

[56] B. Bullnheimer, R. F. Hartl, and C. Strauss, "Applying the ant system to the vehicle routing problem," in *Meta-heuristics*, pp. 285–296, Springer, 1999.

[57] B. Bullnheimer, R. F. Hartl, and C. Strauss, "An improved ant system for the vehicle routing problem," *Annals of Operationas Research*, vol. 89, pp. 319–328, 1999.

[58] H. Ghaziri, "Supervision in the self-organizing feature map: Application to the vehicle routing problem," in *Meta-heuristics*, pp. 651–660, Springer, 1996.

[59] Y. Dumas, J. Desrosiers, E. Gelinas, and M. M. Solomon, "An optimal algorithm for the traveling salesman problem with time windows," *Operations research*, vol. 43, no. 2, pp. 367–371, 1995.

[60] A. G. M. Bani Hani, *Trajectory Privacy in Autonomous Vehicles Communications and Applications*. PhD thesis, 2020.