Carnegie Mellon University
Tepper School of Business

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY
ALGORITHMS, COMBINATORICS, AND OPTIMIZATION

Titled

"Theoretical and Computational Methods for Network Design and Routing"

Presented by

Ziye Tang

Accepted by

Willem-Jan van Hoeve **4/23/2021**

Chair: Prof. Willem-Jan van Hoeve **Date**

Approved by the Dean

Isabelle Bajeux **5/12/2021**

Dean Isabelle Bajeux **Date**

Theoretical and Computational Methods for Network Design and Routing

Ziye Tang

April 2021

Tepper School of Business,
Carnegie Mellon University

Submitted to the Tepper School of Business in Partial Fulfillment of the Requirements for the Degree of Doctor in Algorithms, Combinatorics and Optimization

Dissertation Committee:
Willem-Jan van Hoeve (Chair)
Chrysanthos Gounaris
John Hooker
Ben Moseley
R. Ravi

Abstract

Complex decision-making is ubiquitous in contemporary supply chain management. Over the years, researchers from related fields have developed a variety of techniques to facilitate the modeling and optimization arising in supply chain areas such as network design, inventory management and vehicle routing. Traditionally, however, techniques from these fields are developed separately with their own strengths and weaknesses. For instance, approximation algorithms, developed by theoretical computer scientists, are able to produce solutions with a provable worst-case guarantee but can be too pessimistic for practical applications. On the other hand, heuristics typically focus on the improvement of feasible solutions and neglect the rigorous analysis of the gap with respect to the optimal value.

This dissertation aims to explore connections between neighboring fields such as computer science and operations research by bringing together ideas and techniques from approximation algorithms, complexity theory, constraint programming, decision diagrams and mixed-integer programming. In particular, we study the following three problems in supply chain optimization.

In Chapter 2, we study a two-level network design problem to route demand between source-sink pairs. The demand is routed in a combination of lower level routes that aggregate and disseminate demand to and from local hubs, and upper level routes that transport aggregated demands between hubs. Due to economies of scale, the unit cost of upper level arcs is significantly lower than the lower level counterpart. The objective is to cluster demand at hubs so as to optimally trade off the cost of opening and operating hubs against the discounted costs of cheaper upper level routes. We construct a mathematical model for this problem and term it the hub network design (HND) problem. We study the computational complexity and develop approximation algorithms for the HND and its variants. Inspired by our theoretical analysis, we develop matheuristics that combine evolutionary algorithms, very large neighborhood search and mixed integer programming. Using realistic data, we demonstrate that our approximation techniques provide very good starting solutions for further heuristic search. Our methods produce solutions with an optimality gap of less than 3% in a reasonable amount of time on instances of up to 100 nodes.

In Chapter 3, we consider the deterministic inventory routing problem over a discrete finite time horizon. Given clients on a metric, each with daily demands that must be delivered from a depot and holding costs

over the planning horizon, an optimal solution selects a set of daily tours through a subset of clients to deliver all demands before they are due, with the goal of minimizing total holding and tour routing costs over the horizon. For the capacitated case, a limited number of vehicles are available, where each vehicle makes at most one trip per day. Each trip from the depot is allowed to carry a limited amount of supply. Motivated by an approximation algorithm proposed in the literature that relies on a reduction to a related problem called the prize-collecting Steiner tree (PCST) problem, we develop fast heuristics for both cases by solving a family of PCST instances. Computational experiments show that proposed methods can find near optimal solutions for both cases and substantially reduce the computation time compared to a MIP-based approach.

In Chapter 4, we study a new routing problem called the traveling salesman problem with drone (TSP-D), which is a promising new model for future logistics networks that involves the collaboration between traditional trucks and modern drones. The drone can pick up packages from the truck and deliver them by air while the truck is serving other customers. The operational challenge combines the allocation of customers to either the truck or the drone, and the coordinated routing of the truck and the drone. In this work, we consider the scenario of a single truck with one drone, with the objective to minimize the completion time (or makespan). As a partial explanation for the computational challenge of the TSP-D, we show it is strongly NP-hard to solve a restricted version where drone deliveries need to be optimally integrated in a given truck route. Then we present a compact scheduling-based constraint programming (CP) formulation. Our computational experiments show that solving the CP model to optimality is significantly faster than the state-of-the-art exact algorithm at the time of publication. For larger instances up to 60 customers, our CP-based heuristic algorithm is competitive with a state-of-the-art heuristic method in terms of the solution quality.

The increasing popularity of drone-assisted routing has met with the difficulty of solving those problems to optimality, which has inspired a surge of research efforts in developing fast heuristics. To help develop scalable exact algorithm, as well as to evaluate the performance of heuristics, strong lower bounds on the optimal value are needed. In Chapter 5, we propose several iterative algorithms to compute lower bounds motivated by connections between decision diagrams (DDs) and dynamic programming (DP) models used for pricing in a branch-and-cut-and-price algorithm. Our approaches are general and can be applied to various vehicle routing problems where corresponding DP models are available. By adapting techniques from the DD literature, we are able to generate and strengthen novel route relaxations. We also propose alternative approaches to derive lower bounds from DD-based route relaxations that do not use column generation. All the techniques are then integrated into iterative frameworks to obtain improved lower bounds. When applied to the TSP-D, our algorithms are able to produce lower bounds whose values are competitive to those from the state-of-the-art approach. Applied to larger problem instances where the state-of-the-art approach does not scale, our methods are shown to outperform all other existing lower bounding techniques.

Acknowledgments

First and foremost, I would like to thank my advisor Willem-Jan van Hoeve. Under your continuous guidance, with your unwavering encouragement and support, I improved my technical skills. More importantly, I learned how to get unstuck by approaching things from different angles and seeing opportunities through difficulties, which applies not only to research, but also to life. As I said towards the end of my defense – I will always cherish our random chats at the end of our research meetings during this eventful year.

My gratitude goes to R. Ravi, who taught me how to write the first research paper and continued to provide his mentorship throughout my PhD journey. I'd like to thank the remaining members of my dissertation committee including Chrysanthos Gounaris, John Hooker and Ben Moseley for their helpful comments. I'd like to thank Gérard Cornuéjols and Fatma Kilinç-Karzan for their advice during my time in Tepper. I'd like to thank Anupam Gupta for introducing me to beautiful geometric problems. Last but not least, I'd like to thank Liqiu Meng for offering her invaluable support and advice whenever I needed them.

The work of this dissertation as well as other research output is made possible through the collaboration with many beautiful minds, including C.J. Argue, Patrick Briest, Fei Fang, Takuro Fukunaga, Oleksandr Rudenko, Paul Shaw and Ryan Zheyuan Shi. My life outside research is made colorful by the many people I had fun with, including Alex, Amin, Anthony, Arash, Ashley, Christian, Chenmei, Dabeen, Gerdus, Goran, Mehmet, Melda, Musa, Nam, Neda, Ozgun, Qiao, Rudy, Sagnik, Thiago, Thomas, Violet, Yuchen, Yuyan and Zhiqi. I also would like to thank Sadless for making my PhD journey joyful and I wish you all the best in your future.

My warmest gratitude goes to my family, especially my parents, my grandparents, my great-grandparents and my aunt's family. Thank you for showing me how to be kind and strong. Thank you to my mom for her microscopic care. Thank you to my dad for leading as an example of a hard-working and responsible person. I dedicate this dissertation to all of you.

Table of Contents

1 Introduction **1**

2 Hub Network Design **5**

2.1 Introduction . . . 5

2.1.1 Our Contributions . . . 9

2.1.2 Related Work . . . 10

2.2 Hub Steiner Tree . . . 12

2.2.1 Hardness of Hub Spanning Tree Problem . . . 13

2.2.2 Non-metric Hub Steiner Tree . . . 15

2.3 Metric Hub Steiner Tree . . . 17

2.4 Approximation Algorithm for HND . . . 20

2.5 Heuristic Algorithms for HND . . . 23

2.5.1 MIP Model for HND . . . 23

2.5.2 Heuristic Algorithms for Initial Population . . . 24

2.5.3 Evolutionary Framework . . . 26

2.6 Numerical Experiments . . . 27

2.6.1 Data and Parameter Description . . . 27

2.6.2 Comparison of Heuristic Search . . . 28

2.6.3 Improvement from Evolutionary Framework . . . 30

2.7 Conclusion . . . 31

3 Combinatorial Heuristics for Inventory Routing Problems **33**

3.1 Introduction . . . 33

3.2 Related Work . . . 37

3.3 Uncapacitated Local Search Heuristics . . . 38

3.3.1 DELETE . . . 39

3.3.2 ADD . . . 40

3.3.3 Prioritized Local Search . . . 42

3.4 Uncapacitated IRP Results . . . 42
3.4.1 Experimental Setup . . . 43
3.4.2 Comparison of Different Heuristics . . . 43
3.4.3 Comparison between the Prioritized Local Search and the Baseline MIP . . . 44
3.4.4 True Optimality Gap when MIP is solved exactly . . . 47
3.5 Capacitated Local Search . . . 48
3.5.1 Capacitated ADD . . . 50
3.5.2 Capacitated DELETE . . . 51
3.5.3 Prioritized Local Search . . . 52
3.5.4 From PCST to Feasible Subtours . . . 52
3.6 Capacitated IRP Results . . . 52
3.6.1 Experimental Setup . . . 52
3.6.2 Parameter Settings . . . 53
3.6.3 Performance Evaluation . . . 53
3.6.4 True Optimality Gap when MIP is solved exactly . . . 55
3.7 Conclusion . . . 56

4 A Study on the Traveling Salesman Problem with Drone 57
4.1 Introduction . . . 57
4.2 Related Work . . . 58
4.3 Problem Definition . . . 59
4.4 Theoretical Analysis . . . 60
4.4.1 Computational Complexity . . . 60
4.4.2 Approximation Algorithm for a Special Case . . . 62
4.5 Constraint Programming Formulation . . . 64
4.6 Logic-based Benders Decomposition . . . 67
4.6.1 The Partition Master Problem . . . 68
4.6.2 The Truck-Drone Scheduling Subproblem . . . 70
4.6.3 Benders Cuts . . . 70
4.6.4 Preliminary Results and Limitations . . . 70
4.7 Conclusion . . . 71

5 Truck-Drone Routing with Decision Diagrams 73
5.1 Introduction . . . 73
5.2 Related Work . . . 74
5.3 Problem Definition . . . 76
5.4 Preliminaries . . . 79
5.4.1 Dynamic Programming Model . . . 79

5.4.2 Basic Definitions of Decision Diagrams . . . 81
5.4.3 DD Compilation based on DP . . . 82
5.4.4 Lower Bound from Set Partitioning . . . 83
5.5 Route Relaxation . . . 84
5.5.1 ng-Route Relaxation . . . 84
5.5.2 DD-based Route Relaxation . . . 85
5.5.3 Conflict Refinement . . . 87
5.6 Lower Bound Computation . . . 90
5.7 Iterative Framework . . . 93
5.8 Implementation Details . . . 95
5.8.1 Conflict Refinement . . . 96
5.8.2 Path Decomposition . . . 97
5.8.3 Step Size for the Subgradient Method . . . 97
5.8.4 Improvement Criterion for Function lagAdapt . . . 98
5.8.5 Sensitivity of Bucket and Buffer Size . . . 98
5.9 Computational Experiments . . . 98
5.9.1 Size and Lower Bound from Initial Route Relaxation . . . 99
5.9.2 Lower Bound Improvement . . . 100
5.9.3 Scalability of Iterative Refinement Algorithms . . . 102
5.9.4 Effect of Drone-Truck Speed Ratio . . . 104
5.10 Conclusion . . . 106

6 Conclusions **109**

Appendices **123**

A Appendix for Chapter 3 **125**
A.1 Greedy Heuristic . . . 126
A.1.1 Greedy Framework . . . 126
A.1.2 Approximate Minimum Density Set . . . 127
A.1.3 Implementation Detail . . . 129
A.2 Primal Dual Heuristic . . . 130
A.2.1 LP Formulation . . . 131
A.2.2 A Primal-dual Approach . . . 132
A.2.3 Defining a Feasible Dual . . . 133
A.2.4 Implementation . . . 135
A.3 Uncapacitated IRP Results . . . 137
A.3.1 MIP Formulation for Uncapacitated IRP . . . 137
A.3.2 Additional Experimental Results on Uncapacitated IRP . . . 139

A.4 MIP Formulation for Capacitated IRP 141

B Appendix for Chapter 5 **147**

Chapter 1

Introduction

The supply chains of large corporations consist of various components such as retailers, distributors, plants and suppliers (Mentzer et al., 2001; Min et al., 2019). The goals of corporate supply chains is to provide customers with the products they want in a timely way and as efficiently and profitably as possible. Fueled in part by the rise of e-commerce, the development of models of supply chains and their optimization has emerged as an important way of coping with this complexity (Min and Zhou, 2002; Mula et al., 2010).

Over the years, researchers from related fields such as computer science and operation research have made contributions to a wide range of problems arising in supply chain management (SCM), including network design (Costa, 2005; Gupta and Könemann, 2011), inventory management (Silver, 1981), vehicle routing (Toth and Vigo, 2014) and their integrations (Andersson et al., 2010; Melo et al., 2009). Although their works share a common theme to improve upon the current state of supply chain management, each field has its own focus and many connections between neighboring fields remain unexplored. As a first example, the field of approximation algorithms (Vazirani, 2013; Williamson and Shmoys, 2011) aims to design efficient algorithms that output solutions whose quality has a provable guarantee in the worst case. Therefore the focus is on the theoretical worst-case scenario, rather than the practical performance. On the other hand, its neighboring community of heuristics (Rothlauf, 2011; Talbi, 2009) relies on designing problem-specific solution representation, neighborhood structure, search strategy as well as population-based methods, which typically does not require involved theoretical analysis and proof techniques. As a second example, decision diagrams are originally designed for switching circuits or more generally, Boolean functions (Akers, 1978; Lee, 1959; Wegener, 2000). Only in recent years were they applied to the area of combinatorial optimization (Bergman et al., 2016a).

This dissertation aims to further explore connections among related fields. In particular, we leverage theoretical and computational techniques, including complexity theory, approximation algorithms, mixed-integer programming, constraint programming and decision diagrams, developed from computer science and oper-

ations research, to tackle the following three supply chain optimization problems:

- We begin by studying a hierarchical network design problem that trades off the cost of hub installation and management, against the reduction on the inter-hub routing cost due to the economies of scale. We prove complexity results and develop tight approximation algorithms. Inspired by the theoretical analysis, we design fast matheuristics to compute near-optimal solutions.

- We then zoom in on the inventory management and routing component. In particular, we study the deterministic inventory routing problem. We design an efficient very large neighborhood search heuristic based on an approximation algorithm proposed in the literature and Lagrangian relaxation.

- We further zoom in on the routing component. We first study the traveling salesman problem with drone (TSP-D), which is a truck-drone coordinated routing problem. We prove the computational complexity of a restricted version of the TSP-D and propose a compact constraint programming model. We further explore connections between decision diagrams and dynamic programming models used for pricing in a branch-and-cut-and-price algorithm. This motivates us to develop iterative frameworks that generate lower bounds for general vehicle routing problems. The proposed algorithms are applied and tested on TSP-D instances.

Below we give a more detailed description.

In Chapter 2, we study a two-level network design problem to route demand between source-sink pairs. The demand is routed in a combination of lower level routes that aggregate and disseminate demand to and from local hubs, and upper level routes that transport aggregated demands between hubs. Due to the economies of scale, the unit cost of upper level arcs is significantly lower than the lower level counterpart. The objective is to cluster demand at hubs so as to optimally trade off the cost of opening and operating hubs against the discounted costs of cheaper upper level routes. We formulate a mathematical model for this problem and term it the hub network design (HND) problem. In addition, we propose a simplified variant called the hub Steiner tree (HStT) problem which focuses on the connectivity of a network by abstracting away the routing component. We start with this simplified variant. We prove that it is NP-hard by reducing it to the set cover problem. We then develop tight (up to constants) approximation algorithms for the HStT by reducing the non-metric (metric, resp.) HStT to the node-weighted (edge-weighted, resp.) Steiner tree. Next we consider solving the HND in practice. In particular, we first develop an approximation algorithm for the HND by problem reduction and graph spanners. The HND is reduced to the uncapacitated facility location (UFL) problem, which highlights the importance of hub locations. This inspires us to develop an evolutionary framework where the initial population consists of approximate solutions to a family of UFL instances or a mixed-integer program. Using realistic data, we demonstrate that our approximation techniques can provide very good starting solutions for further heuristic search. Our framework can produce solutions with an optimality gap of less than 3% in a reasonable amount of time.

In Chapter 3, we consider the deterministic inventory routing problem (IRP) over a discrete finite time

horizon. Given clients on a metric, each with daily demands that must be delivered from a depot and holding costs over the planning horizon, an optimal solution selects a set of daily tours through a subset of clients to deliver all demands before they are due and minimizes the total holding and tour routing costs over the horizon. In the capacitated case, a limited number of vehicles are available, where each vehicle makes at most one trip per day. Each trip from the depot is allowed to carry a limited amount of supply to deliver. We exploit a conceptual reduction proposed in the literature to a related problem called prize-collecting Steiner tree (PCST) problem, for which fast exact algorithms exist in the literature. A unifying theme of our heuristics is to reduce the search space by creating and solving PCST instances as intermediate steps to determine which clients to visit each day. For the uncapacitated case, we define a very large neighborhood search step and reduce it to a PCST problem. For the capaciated case, we show a similar approach is possible with the help of Lagrangian relaxation. Computational experiments show our heuristics can find near optimal solutions for both cases and substantially reduce the computation time compared to a MIP-based approach. This work will appear in INFORMS Journal on Computing (Tang et al., to appear).

In Chapter 4, we study a new routing problem called the traveling salesman problem with drone (TSP-D). TSP-D is a promising new model for future logistics networks that involves the collaboration between traditional trucks and modern drones. The drone can pick up packages from the truck and deliver them by air while the truck is serving other customers. The operational challenge combines the allocation of customers to either the truck or the drone, and the coordinated routing of the truck and the drone. In this work, we consider the scenario of a single truck and one drone, with the objective to minimize the completion time (or makespan). Since TSP-D generalizes the well-known traveling salesman problem (TSP), it is theoretically hard to solve to optimality. However this theoretical result is not sufficient to explain the computational findings in the literature that TSP-D is significantly harder to solve than TSP. In this chapter, we first shed light on this question: we prove that this problem is strongly NP-hard, even in the restricted case when drone deliveries need to be optimally integrated in a given truck route. We then present a constraint programming formulation that compactly represents the operational constraints between the truck and the drone. Our computational experiments show that solving the CP model to optimality is significantly faster than the state-of-the-art exact algorithm at the time of publication. For larger instances up to 60 customers, our CP-based heuristic algorithm is competitive with a state-of-the-art heuristic method in terms of the solution quality. This work is published in the conference proceedings of CPAIOR 2019 (Tang et al., 2019).

The increasing popularity of drone-assisted routing has met with the difficulty of solving those problems to optimality, which has inspired a surge of research efforts in developing fast heuristics. To help develop scalable exact algorithm, as well as to evaluate the performance of heuristics, strong lower bounds on the optimal value are needed. In Chapter 5, we propose several iterative algorithms to compute lower bounds motivated by connections between decision diagrams (DDs) and dynamic programming (DP) models used for pricing in a branch-and-cut-and-price algorithm. Our approaches are general and can be applied to various vehicle routing problems where corresponding DP models are available. By adapting merging and refinement techniques from the DD literature, we are able to generate and strengthen novel route relaxations. We also

propose two alternative approaches to derive lower bounds from DD-based route relaxations which use a flow model with side constraints and Lagrangian relaxation, respectively, in place of column generation. All the techniques are then integrated into iterative frameworks to obtain improved lower bounds. When applied to the TSP-D, our algorithms are able to produce lower bounds whose values are competitive to those from the state-of-the-art approach. Applied to larger problem instances where the state-of-the-art approach does not scale, our methods are shown to outperform all other existing lower bounding techniques.

Chapter 2

Hub Network Design

2.1 Introduction

Designing and operating logistics networks has become increasingly important with the evolution of online shopping and fulfillment and a variety of delivery services. Logistics service providers typically operate at minimal margins, so that a high degree of consolidation and the resulting economies of scale are mandatory to run a profitable business. This leads to a typical layered or hierarchical design of logistics networks, in which the upper layer handles high volume traffic and achieves lower unit cost by consolidating multiple flows on longer distance connections, while the lower layer solves the last mile issues using local aggregation and distribution services.

In this chapter, we study this abstract problem that arises in the design of logistics networks that handle periodic demands between origin-destination (O-D) pairs (or simply client pairs). The intention is to exploit the cost differential between full-truck routing (FTL) and less-than-full truck routing (LTL). In FTL, the fleet is owned by the logistics firm which strives to maximize its amortized utilization. In LTL, typically smaller volumes than the capacity of the truck are subcontracted to other providers who operate the truck, but since only a fraction of the capacity is used (both in time and space), the rates per unit distance per unit demand (weight or volume) are also much higher than FTL. Furthermore, FTL operations require opening *hubs* which are locations where the fleet is parked and maintained and serve as potential transshipment points for material flow. This necessitates that FTL routes are operated only between pairs of hubs. Non-hub demand locations are served by LTL routes (also called ”milkruns”) that originate and end at a hub. We think of the short-haul LTL milkruns as pickup and delivery runs, while the FTL runs between facilities form the mainhaul routes. We refer to the network of LTL milkruns as the *lower level network* and the network of FTL runs as the *upper level network*.

The network is typically designed for periodic multi-commodity demand. Every period, each client pair may

be served by a route consisting of the following three segments: a pickup milkrun that collects the demand from the origin and aggregates it at its hub; a series of mainhaul FTL routes that transport the demand from the pickup hub to a delivery hub; a final delivery milkrun that deposits the demand from the final hub to the destination. For operational simplicity, we assume that each client is served by a *unique* hub which may be located in the same place. We also assume a direct connection between each client and its assigned hub, where the cost is linear in the distance and the quantity demanded. Thus, we have a two-level network design problem: in the lower level we have a partition clustering of the demand locations into hubs that serve them using LTL pickup-delivery routes – a set of stars under our assumption; In the upper level, we have trucks that carry all implied demand using inter-hub routes; however we allow aggregated flows between hubs to be split during the routing process. We call this two-level problem the *hub network design* (HND) problem, and describe it more formally next.

Problem Definition: Hub Network Design (HND)

We are given an undirected graph $G = (V, E)$ with non-negative edge cost c_e for $e \in E$ per unit flow, non-negative hub opening cost f_v for $v \in V$, and an upper level truck capacity M and an economies-of-scale parameter $\lambda \in [0, 1]$. Each client pair (i, j) has a nonnegative demand requirement W_{ij} for flow between them. The HND problem involves three sets of decisions.

1. The choice of a subset of vertices $F \subseteq V$ to install hubs.
2. The assignment of each client to a single hub via an allocation function $\pi : V \to F$, which we call the *parent hub* of the client, and
3. A routing of all origin-destination demands (i, j) in three segments: $(i, \pi(i))$ from the origin to its parent hub in a direct link, a set of paths in the upper level P^u_{ij} from the origin's parent hub $\pi(i)$ to the destination's parent hub $\pi(j)$ via other hubs if necessary, and $(\pi(j), j)$ from the destination's parent hub to the destination. See Figure 2.1 for an illustration of a single path routing such a demand. We emphasize that the aggregated flow between the hubs can be routed in a splittable fashion using more than one path from any hub to another hub.

Recall that we refer to the star network which sends flows from clients to hubs as the lower level network and the network which sends flows between hubs as the upper level network. Each of the decisions above gives rise to a term in the final cost. We pay the hub opening costs $f : V \to \mathbb{R}^+$ for vertices in the hubs F. The demands to and from hubs is paid at the rate of the edge costs c per unit demand in the lower level. The upper level flows between hubs are supported by copies of bidirected arcs of capacity M each representing roundtrips of trucks, but the cost of such an arc in an edge e is $\lambda M c_e$ reflecting the economies of scale. We detail the assumptions and cost calculations below.

We make the following assumptions in defining HND and its cost function. (a) Each client sends its total

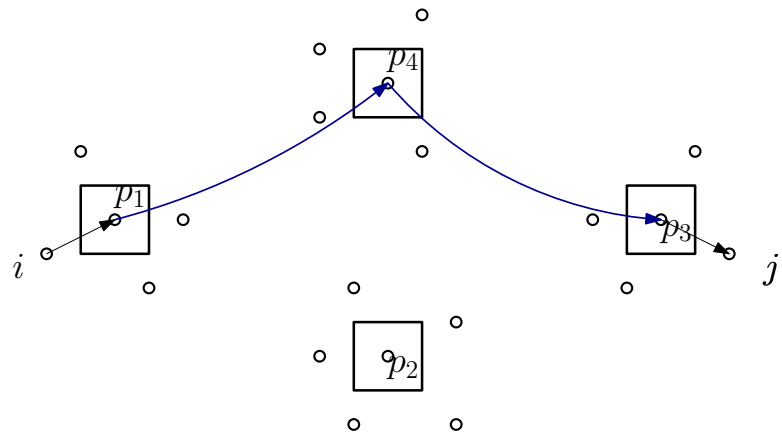


Figure 2.1: An example of flow routing from origin i to destination j: each point inside a square denotes an open hub. The flow travels from i to $p_1 = \pi(i)$ using a direct LTL route, from $p_1 = \pi(i)$ to $p_3 = \pi(j)$ using fractional FTL routes such as the one shown via p_4 and finally from $p_3 = \pi(j)$ to j again using a direct LTL route. Note that the costs of the LTL segments are paid fractionally while the FTL edges in the network between hubs are installed and paid for integrally.

demand *directly* to its parent hub, i.e. the lower level network is a collection of star networks (Section 2.6.1 explains how we move from the milkrun problem to this approximation to the star). Note that this implies a simplification of the costs on these star networks: If i is assigned to parent hub $\pi(i)$, and the total flow out of i is $O_i = \sum_j W_{ij}$ and the total flow into i is $D_i = \sum_j W_{ji}$, the flow cost of the edge from i to its parent hub is $c(i, \pi(i)) \cdot (O_i + D_i)$. (b) The upper level network consists of 2-cycles (anti-parallel arcs) representing the movement of the same truck between two hubs (The upper level FTLs are implemented in practice as round trips between a pair of hub locations). (c) Each commodity (represented by the flow) is assumed to be divisible, except for the part to and from the hubs. (d) We assume all upper level trucks have the same capacity M which cannot be exceeded when transporting goods. However, if the total aggregated flow on an upper level edge exceeds M, we are allowed to buy multiple copies of anti-parallel arcs to support it.

We emphasize that the distinctive feature between the lower and upper level network is that trucks can be leased fractionally in the lower level whereas only integer copies of trucks can be deployed on the upper level network. Thus the flow in the upper level network implements the paths between hubs fractionally but the number of integral FTL links purchased between a pair of hubs g, h is determined by the maximum of the flow in either direction between the pair (due to the 2-cycle requirement). Let $y_{ij}(g, h)$ be the amount of flow that demand pair (i, j) sends via arc (g, h). Then the number of FTL trucks purchased between g and h is

$$t_{gh} = t_{hg} = \left\lceil \max \left(\sum_{(i,j):P^u_{ij} \ni (g,h)} \frac{y_{ij}(g,h)}{M}, \sum_{(i,j):P^u_{ij} \ni (h,g)} \frac{y_{ij}(h,g)}{M}. \right) \right\rceil,$$

where P^u_{ij} is the set of upper level arcs in the chosen path sending demand from i to j.

However the upper level truck is λ times cheaper per unit distance than the lower level truck for a given fixed economies-of-scale factor $\lambda \in [0, 1]$. Recall $c(\cdot)$ is the lower level cost per unit demand. Thus, the total cost of these FTL routes is $\sum_{g,h \in F} \lambda c(g, h) M \cdot t_{gh}$, where F is the set of hubs opened to operate the upper level edges. To summarize, the total cost to be minimized has three corresponding components.

1. The cost of opening hubs $\sum_{v \in F} f(v)$,
2. The cost of lower level star networks $\sum_{i,j} (c(i, \pi(i)) + c(\pi(j), j) \cdot W_{ij} = \sum_{i \in V} c(i, \pi(i)) \cdot (O_i + D_i)$ and
3. The cost of upper level truck routes $\sum_{g,h \in F} \lambda c(g, h) M \cdot t_{gh}$.

The complexity of the HND problem is driven by the combination of lower and upper level network design. In order to approach the problem from a theoretical angle, we next define two variations with different degrees of simplifications on the network architecture – (i) by relaxing the integral requirement in the upper level and (ii) further relaxing a flow routing network altogether and just building a Steiner tree with two levels representing economies of scale. Investigations of the approximability of these two variations yields helpful insights into combinatorial aspects of the problem. The first, the *clique hub location problem* (CHLP) defined below provides the basis for our approximation algorithm for the HND. The second, *hub Steiner tree* (HStT), provides a simplified model that highlights the difficulty of the non-metric version of the problem where the cost function c does not obey the triangle inequality. For this problem, we show polynomial-time reductions to well-known graph problems in both non-metric and metric cases, which yield approximation algorithms with matching approximation factors (up to constants).

Problem Definition: Clique Hub Location Problem (CHLP)

Definition 2.1.1. The clique hub location problem (CHLP) is a relaxed version of the HND where we allow trucks to be fractionally leased on the upper level network, and the flows in both directions in the upper level network need not be equal.

Since we no longer have integrality constraints on the number of trucks deployed on the upper level, each flow on the upper level will travel directly from an origin hub to a destination hub in an unsplittable fashion along the shortest path. Moreover, when the cost function obeys the triangle inequality, this path will be the direct edge from the origin to destination. We use CHLP as an intermediate step in designing an approximation algorithm for the general HND.

Problem Definition: Hub Steiner Tree (HStT)

Definition 2.1.2. Given an undirected graph $G = (V, E)$ with a terminal set $R \subseteq V$, non-negative edge cost c_e for $e \in E$, non-negative hub opening cost f_v for $v \in V$ and a constant $\lambda \in [0, 1]$ reflecting the cost differential between two levels, a *hub Steiner tree (HStT)* is a tree T spanning the terminal set R along with a set of hubs $H \subset V$. Let T_H denote the set of edges in T induced by H (i.e., with both ends in H). We call T_H the set of *upper-level* edges and $T \backslash T_H$ the set of *lower-level* edges. We use the shorthand notation to let $c(S) = \sum_{e \in S} c_e$ for $S \subseteq E$ and $f(U) = \sum_{v \in U} f_v$ for $U \subseteq V$. The cost of the hub spanning tree (T, H) is $\lambda c(T_H) + c(T \backslash T_H) + f(H)$. Note that the cost of upper-level edges is λ times cheaper than the cost of lower-level edges. The goal of the HStT problem is to find an HStT of minimum cost.

Remark 2.1.1. When G is a complete graph and the edge costs c satisfy the triangle inequality (i.e., $c_{xy} + c_{yz} \geq c_{xy}$ for all $x, y, z \in V$), then the instance belongs to the *metric case*, and the problem consisting of such instances is called the *metric* HStT problem.

Remark 2.1.2. It is immediate to extend the HStT to more general network design problems such as the Steiner forest and the generalized one-connected network design problems using the formalism and techniques in Klein and Ravi (1995a), but we do not elaborate on these extensions here.

2.1.1 Our Contributions

1. One of our key contributions is the theoretical modeling of the two-level logistics network problem as a hub network design problem, and introducing its simplified variants, the CHLP and the HStT.

2. We consider the HStT in Section 2.2.

   (a) In Section 2.2.1, we show NP-hardness and logarithmic approximation hardness for the HStT from the set cover problem.

   (b) For the non-metric HStT in Section 2.2.2, we show a polynomial-time reduction to the node-weighted Steiner tree problem. This implies an approximation algorithm with matching logarithmic ratio that extends to more general connectivity requirements modeled in Klein and Ravi (1995a).

   (c) For the metric HStT in Section 2.3, we show a polynomial-time reduction to its original version (with no hub installation or two-level edge cost). This implies a constant-factor approximation algorithm that also extends to the more general cases modeled in Klein and Ravi (1995a).

3. Next we consider the HND with metric costs in Section 2.4.

   (a) We design the first constant approximation algorithm for the CHLP with performance ratio $1 + 2\gamma_{UFL}$ where γ_{UFL} is the best known approximation ratio for the uncapacitated facility location

problem with metric costs.

(b) To design an approximation algorithm for the metric HND problem, we reduce it to the CHLP. By constructing a light graph spanner for the upper level and sending consolidated flows along the spanner, we obtain an $O(\log n)$ approximation algorithm for the HND on an n-node graph.

4. In Section 2.5 we study the HND from a computational lens. We propose an evolutionary framework motivated by the aforementioned theoretical analysis.

   (a) In Section 2.5.1, we propose a mixed-integer programming (MIP) model for the HND, which will be used for our evolutionary framework.

   (b) The initial set of solutions for our framework comes from two heuristic algorithms proposed in Section 2.5.2 – one based on the MIP model and the other based on solving a family of uncapacitated facility location instances, both of which are motivated by the design of our approximation algorithm for the HND.

   (c) Section 2.5.3 describes our evolutionary framework in more detail. In particular, we implement a MIP-based very large neighborhood search. This search mechanism is used in both the exploration and exploitation step in our framework.

5. Finally in Section 2.6, we present numerical experiments on both randomly sampled and real-world datasets. It is shown that our approximation techniques can provide very good starting solutions, and that our framework can find near-optimal solutions for all test instances. When tested on realistic data, our methods can provide solutions within 3% of optimal solution in less than an hour.

2.1.2 Related Work

Models. First we review network design models closely related to the HND. A two-level network design that appears very similar to HND and has been well-studied in telecommunication network design goes by the name of *access network design* (Balakrishnan et al., 1994). The key difference is that in access networks, the inner or upper networks need more resilience and hence are of higher cost per unit volume per unit distance, while in HND, the costs of the upper level are lower than that of the lower level. Andrews and Zhang (1998) showed an $O(K^2)$ approximation algorithm for a special case of the problem where K is the number of edge types. We refer the reader to Carpenter and Luss (2006) for a comprehensive review.

Our work is more directly related to the so-called *buy-at-bulk network design* problem (Salman et al., 2001). In buy-at-bulk network design, one needs to minimize the cost of sending flow from sources to sinks along paths where the cost of each edge displays economies of scale, i.e. the more capacity is installed on an edge, the cheaper it is per unit capacity (per unit distance). For the uniform cost model where each edge has the same cost function, Awerbuch and Azar (1997) showed an $O(\log n)$ approximation algorithm for

multicommodity buy-at-bulk problem while $O(1)$ approximations are known for the single-sink case (Guha et al., 2009; Gupta et al., 2003; Kawahara et al., 2009). For the non-uniform case, Chekuri et al. (2010) improved upon Charikar and Karagiozova (2005) and showed a poly-logarithmic approximation. A super-constant hardness of approximation for the multi-commodity case was shown by Andrews (2004). Our problem differs from buy-at-bulk network design in two ways. First our problem involves more levels of decisions (hub location, client allocation and flow routing). Second the economies of scale in our problem relies on the fact that FTL routes have lower cost per unit volume per unit distance than LTL routes but needs hubs to operate between, while in buy-at-bulk network design these economies are modeled by considering several types of cables, where cables with larger capacity has lower cost per capacity than ones with smaller capacity with no hubs to open.

The most closely related problem to HND that has been studied in the literature is the *hub location problem* (HLP). HLP involves hub opening and client-hub allocation decisions. The key difference from HND is that in HLP, the upper level cost is also linear in distance times the quantity demanded, which means the mainhaul upper level trucks can also be bought fractionally. Further structural assumptions are often made on both lower and upper level network, the most common ones being that the upper level is fully connected, forms a star, or a cycle (see e.g. Contreras and Fernández (2012)). The clique hub location problem (CHLP) we defined is a variant of this general hub location problem. Also, our mixed integer programming (MIP) model in Section 2.5.1 is a modification of a user-user demand model from Contreras and Fernández (2012) with additional integrality constraints on the upper level network. For a comprehensive review of general HLP and its variants, we refer the reader to Alumur and Kara (2008); Farahani et al. (2013) and the references therein.

Another related problem is *location routing problem* (LRP). In the classical setting of this problem, one needs to determine 'depots' to open and vehicle routes traveling from each depot to servicing its clients. The cost consists of three parts: cost of opening depots, fixed costs of vehicle used and the cost of routes. Our problem (HND) differs from LRP in two aspects. First our problem considers sending flow from client to client instead of distributing flow from each depot to clients assigned to it. Second we make a further simplification on the vehicle routes as star networks in the lower level. For a review of LRP, we refer the reader to two surveys by Nagy and Salhi (2007); Prodhon and Prins (2014).

Table 2.1 presents a comparison of the key features of various models above with HND. We consider the multi-commodity versions of all these problems in the table

Techniques. We next review previous work containing algorithmic results that we draw upon in our work.

Our approximation for the non-metric HStT uses a reduction to the node-weighted Steiner tree problem, for which Klein and Ravi (1995b) showed a greedy algorithm which achieves a logarithmic approximation factor. For the metric HStT, we show a reduction to the (edge-weighted) Steiner tree problem, for which the

	Hub costs	Two levels of edge costs	Upper level integral	Lower Level	Upper versus Lower costs
HND	Yes	Yes	Yes	Star	Cheaper
Buy-at-Bulk	No	Yes	General	General	Cheaper
Access ND	Yes	Yes	Yes	Star	Costlier
HLP	Yes	Yes	No	Star	Cheaper
LRP	Yes	No	N/A	Tours	N/A

Table 2.1: Comparison of HND with related problems

best approximation ratio $\gamma_{ST} = \ln 4 + \epsilon$ is obtained by Byrka et al. (2013); Goemans et al. (2012).

Our approximation for the HND uses a reduction to the CHLP. Sohn and Park (1997) first considered this problem and proved polynomial-time solvability when the number of hubs is at most two. Subsequently, they showed the NP-hardness result when the number of hubs is more than two (Sohn and Park, 2000). When all the hub locations are given, Iwasa et al. (2009) showed the first constant approximation for the CHLP. When all the nodes are in the Euclidean plane, Ando and Matsui (2011) showed a $1 + \frac{2}{\pi}$-approximation based on a dependent rounding procedure applied to an LP relaxation.

We also make use of a light graph spanner (subgraphs that preserve all-pair distances approximately while being not much larger than a minimum spanning tree) to give a new approximation algorithm for the HND. Mansour and Peleg (1994) used a light spanner to give an approximation algorithm for the buy-at-bulk network design problem with a single cable type. In particular, they showed how to construct such a spanner for a general weighted n-node graph that is $O(\log n)$ on both the stretch of all-pair distances and the weight of the spanner w.r.t. the minimum spanning tree. Recent developments for improving the lightness of the spanner can be found in Chechik and Wulff-Nilsen (2016); Elkin et al. (2015).

We use the UFL as a way to obtain approximation algorithms for the CHLP. We note that $\gamma_{UFL} = 1.488$ is the best known approximation ratio for the UFL problem due to Li (2011).

For heuristic algorithms presented in Section 2.5, our MIP-based evolutionary framework is based on the very large neighborhood search idea from Rothberg (2007). The integration of evolutionary frameworks and MIP models falls into the category of matheuristics. We refer the reader to Raidl and Puchinger (2008) and references therein for this type of hybridization.

2.2 Hub Steiner Tree

In this section, we first study the computational complexity of the HStT by proving hardness results for a special case of the HStT, which we call the *hub spanning tree* (HST) problem. Next, we develop tight

approximation algorithms (up to constants) for the non-metric and the metric HStT. The special case is defined as follows. Recall R is the set of terminals for the HStT.

Definition 2.2.1 (Hub spanning tree (HST))**.** The hub spanning tree (HST) problem is a special case of the HStT where the terminal set $R = V$.

Since the HST is a special case of the HStT, hardness results for the former immediately implies those for the latter.

2.2.1 Hardness of Hub Spanning Tree Problem

In this section, we prove hardness results for the HST problem by reducing from the set cover problem. Similar reductions have been used in several related network design problems (e.g., Klein and Ravi (1995a); Bentert et al. (2017); Erzin et al. (2013)).

Definition 2.2.2 (Set Cover)**.** Let $S_1, \ldots, S_n$ be arbitrary subsets on a ground set of elements $x_1, \ldots, x_t$. The set cover problem is to find a minimum cardinality set of subsets whose union is the set of all elements.

The following hardness of approximation result for the set cover problem is due to Dinur and Steurer Dinur and Steurer (2014).

Theorem 2.2.1 (Dinur and Steurer (2014))**.** *For every $\delta > 0$, it is NP-hard to approximate the set cover problem to within $(1-\delta)\ln n$, where n is the size of the instance.*

Our hardness results presented in this section repeatedly use the following reduction from the set cover problem.

Reduction 1. We construct an undirected weighted graph $G = (V, E)$ as follows: create a node v_{S_i} for each set S_i, a vertex v_j for each element x_j, and a new vertex v_r as the root. Let $A := \{v_{S_i} : i = 1, \ldots, n\}$ and $B := \{v_j : j = 1, \ldots, t\}$. For each $v_S \in A$ we create an edge (v_r, v_S) with cost 0. For each $v_S \in A$ and $v_j \in B$ such that $j \in S$, we create an edge (v_j, v_S) with cost β whose value will be set later to achieve desired hardness results. The hub opening cost is one for all vertices in A and zero for all others. See Figure 2.2 for an illustration.

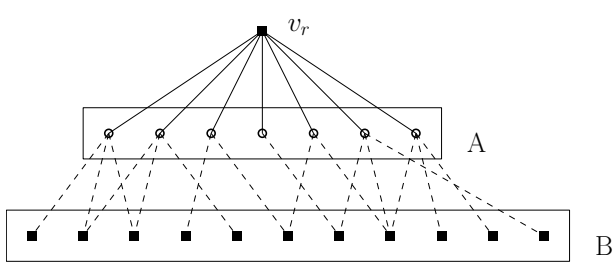


Figure 2.2: An illustration of the reduction. The cost of a solid (dashed, resp.) edge is zero (β, resp.).

Non-metric HST Based on the above construction, we have the following theorem:

Theorem 2.2.2. *For any $\lambda \in [0, 1)$, the non-metric HST is NP-hard.*

Proof. Set $\beta > \frac{1}{1-\lambda}$ in Reduction 1. We show that the minimum set cover has cardinality k if and only if the optimal HST cost in Reduction 1 is $k + \lambda\beta t$, where t is the number of elements. Notice this proves the theorem.

Without loss of generality, assume an optimal set cover is $\{S_j\}_{j=1}^k$. Our HST is constructed by opening hubs in $\{v_r\} \cup \{v_{S_j}\}_{j=1}^k \cup B$ and selecting the following set of edges: $\{(v_r, v_{S_j}), (v_{S_j}, v_l) : \forall j = 1, \ldots, k, l \in S_j\}$. In the resulting HST, a vertex in set B is connected to only one vertex in set A (breaking ties arbitrarily between a pair of A nodes that have edges to it). The resulting HST has cost $k + \lambda\beta t$. This shows the minimum cost of the HST problem is less than or equal to $k + \lambda\beta t$. Conversely, we claim all edges in the optimal HST between A and B have both endpoints opened as hubs: suppose not, let (v_S, v_l) be an edge violating this property. We can alternatively open hubs on both endpoints which incurs a unit hub opening cost and reduces the cost of (v_S, v_l) by $(1 - \lambda)\beta > 1$ (this might lead to cost decrease in other edges too), which results in a contradiction. Consequently the set $\mathcal{S} := \{v_S : S \in A \text{ is an opened hub in HST}\}$ is a valid set cover. Observe that the edges between A and B now have cost $\beta\lambda$ and there are t such edges, so we have $|\mathcal{S}| \leq k$. This shows the minimum set cover has cardinality less than or equal to k. □

Similarly we have the following approximation hardness result.

Theorem 2.2.3. *If there is an α-approximation algorithm for the non-metric HST problem with $\lambda = 0$, then there is an α-approximation algorithm for the set cover problem.*

Proof. Given a set cover instance with minimum cardinality k, we generate the HST instance as in Reduction 1 with $\lambda = 0$ and $\beta > \alpha k$. Let T be an α-approximate solution of this instance. We will prove that every edge in T has both endpoints opened as hubs. Suppose not, there exists one edge that is not between two hubs, incurring a cost of at least $\beta + (t - 1)\beta\lambda = \beta$ (since $\lambda = 0$). Since $\beta > \alpha k$, this implies T is not an α-approximate solution, contradicting our assumption.

Let k be the cardinality of a minimum set cover. Next we show the cost of an optimal HST is at most k: we can open hubs at the k vertices that correspond to the optimal set-cover, whose cost is $k + \lambda\beta t = k$. As a result, the cost of T is at most αk. We can therefore obtain an α-approximate set cover solution by selecting those sets opened as hubs. □

We obtain the following corollary by combining Theorems 2.2.1 and 2.2.3.

Corollary 2.2.4. *When $\lambda = 0$, for any $\delta > 0$ it is NP-hard to approximate the non-metric HST problem within a factor of $(1 - \delta) \ln n$.*

Metric HST For notational convenience, we shall denote nodes in A as *A-nodes* and nodes in B as *B-nodes*. We call an A-node *selected* if its corresponding set is included in the set cover solution.

Theorem 2.2.5. *For any $\lambda \in (0, 1)$, the metric HST problem with uniform hub opening cost is NP-hard.*

Proof. We modify Reduction 1 for the non-metric HST. We assign a unit hub opening cost for every node. Recall that the edge weight between the root and an A-node is 0, and the edge weight between an A-node and an B-node is β. We take the metric completion of this graph, i.e., we add all the edges in the complete graph where the cost of an edge is defined as the shortest path length between its two endpoints. based on these edge weights. Recall t is the number of elements. We claim that for $\beta > \max\{\frac{1}{\lambda}, \frac{2}{1-\lambda}\}$, the minimum cost of a HST is $k + t + t\lambda\beta$ if and only if the size of the minimum set cover is k.

For the 'if' part, given a set cover of size k, we install hubs on all B-nodes and selected A-nodes. We connect each A-node to the root by edge cost 0. We connect each B-node to a selected A-node which includes that element by paying $\lambda\beta$. This gives an HST with cost $k + t + t\lambda\beta$.

For the 'only if' part, since the root and all A-nodes are connected by the edges of cost 0, no hub is needed at the root to reduce the edge cost between the root and an A-node. We will ensure that opened hubs among A-nodes exactly represent selected sets. To do so, we need to ensure two things in an optimal HST.

(I) Each edge between an A-node and a B-node is an upper-level edge (i.e., hubs are opened on its both endpoints);

(II) No edge exists between two A-nodes, or two B-nodes.

A sufficient condition for (I) is $\beta > \lambda\beta + 2$ where β is the lower-level edge cost for connecting an B-node to an A-node and $\lambda\beta + 2$ is the upper-level edge cost and the hub opening costs of its two end nodes. This condition also implies that, if an edge joining two B-nodes is used in the optimal HST, it is the upper-level edge because lower-level edges joining two B-nodes are of cost at least 2β ($> \beta > \lambda\beta + 2$). For (II), first notice two A-nodes are already connected via the root by two 0-cost edges. Second, under the condition for (I), it is sufficient to have $2\lambda\beta > \beta\lambda + 1$ where $2\lambda\beta$ is the cost of the upper-level edges joining two elements-nodes and $\beta\lambda + 1$ is the cost for connecting a B-node to an A-node by an upper-level edge by opening a hub on the common A-node. To summarize, we need $\beta > \max\{\frac{1}{\lambda}, \frac{2}{1-\lambda}\}$ which are the conditions in the claim. □

2.2.2 Non-metric Hub Steiner Tree

In this section we reduce the HStT problem to the node-weighted Steiner tree problem defined below.

Definition 2.2.3 (Node-weighted Steiner tree (NWST))**.** Let G be an undirected graph with nonnegative costs assigned to its nodes and edges. Let $R \subseteq V$ be a set of terminals. A Steiner tree for R in G is a

connected subgraph of G containing all the nodes of R. The *node-weighted Steiner tree* (NWST) problem is to find a minimum-cost Steiner tree.

For NWST, Klein and Ravi Klein and Ravi (1995a) showed a greedy algorithm which achieves a logarithmic approximation factor.

Theorem 2.2.6 (Klein and Ravi (1995a))**.** *The NWST problem admits a polynomial-time* $2 \ln k$*-approximation algorithm, where* k *is the number of terminals.*

Below we present our reduction.

Reduction 2. Given an HStT problem as in Definition 2.1.2. We create an NWST instance as follows. Let V' and E' be the node and edge set of the NWST instance respectively. Let $c' : E' \to \mathbb{R}^+$ be the edge weight function. For each node $v \in V$ in HStT, we create a pair of nodes v_h, v_l. Let $V_{\text{upper}} := \{v_h : v \in V\}$ and $V_{\text{low}} := \{v_l : v \in V\}$, where V_{upper} stands for the 'upper-level nodes' and V_{low} for the 'lower-level nodes'. Let $V' = V_{\text{upper}} \cup V_{\text{low}}$. Define the set of terminals $R' := \{v_l : v \in R\}$. For each edge $e = (u, v)$ in HStT, we add to E' all possible edges between these vertices: $(u_h, u_l), (v_h, v_l), (u_h, v_h), (u_l, v_l), (u_h, v_l), (u_l, v_h)$. Edge weights are defined as follows:

$$c'(u_h, u_l) = c'(v_h, v_l) := 0.$$

$$c'(u_l, v_l) = c'(u_h, v_l) = c'(u_l, v_h) := c(u, v).$$

$$c'(u_h, v_h) := \lambda c(u, v).$$

For each $v \in V$, the node weight on v_h is defined as f_v and on v_l as zero. See Figure 2.3 for an illustration.

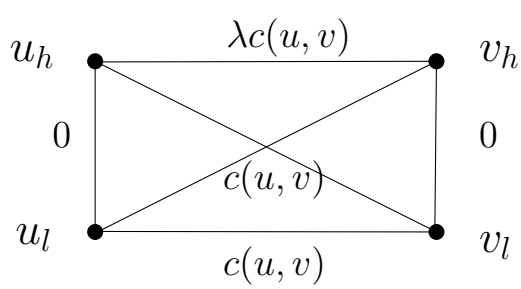


Figure 2.3: Edge weights in Reduction 2

Theorem 2.2.7. *If we have a* γ*-approximation algorithm for the NWST problem, then there exists a* γ*-approximation algorithm for the HStT problem.*

Proof. First, we show that the optimal value of the reduced NWST instance is at most that of the given HStT instance. Let T be a hub Steiner tree T of cost $c(T)$ in the HStT instance. We construct a Steiner tree T' of cost at most $c(T)$ for the reduced NWST instance. For upper-level edges (u, v) in T, we add $(u_h, v_h), (u_h, u_l), (v_h, v_l)$ to T'. For lower-level edges (u, v) in T, we add (u_l, v_l) to T'. It is straightforward to verify that T' has the same cost as T. Next we show T' is indeed a Steiner tree that connects terminals

in R'. Consider any pair of nodes (u, v) in R; Since T is a Steiner tree in HStT, there exists a path that connects u and v in T. Call this path P. We will find a path P' in T' that connects u_l and v_l as follows: for any upper-level edge (a, b) in P, add edges $(a_l, a_h), (a_h, b_h), (b_h, b_l)$ to P'. For any lower-level edge (a, b) in P, add an edge (a_l, b_l). It is easy to see that P' indeed connects u_l and v_l.

Next, we prove the opposite direction. Let T' be a feasible Steiner tree spanning R for the NWST instance. We show that there exists a hub Steiner tree T with cost $c(T) \leq c'(T')$ for the HStT instance. For each upper-level node $v_h \in V_{\text{upper}}$ spanned by T', we install a hub on v in T. For each edge (u_h, v_h) in T' where $u_h, v_h \in V_{\text{upper}}$, we add an upper level edge (u, v) to T. For edges of the form (u_h, v_l) or (u_l, v_l), we add a lower-level edge (u, v). For the remaining edges in T', we do nothing. Arbitrarily delete edges to remove cycles in T as necessary. It is easy to verify that T connects all terminals of R and has cost no more than $c'(T')$. □

As a corollary of Theorems 2.2.6 and 2.2.7, we obtain the following result.

Corollary 2.2.8. *There is a polynomial-time $2 \ln k$-approximation algorithm for the non-metric $HStT$ problem, where k is the number of terminals.*

2.3 Metric Hub Steiner Tree

In the previous section, we reduced the HStT problem to the NWST problem. In this section, we show that, if the edge-costs are metric, the HStT problem can be reduced to the edge-weighted Steiner tree (EWST) problem, the special case of the NWST in which all node costs are zero. The EWST problem admits a number of constant-factor approximations. The currently known best approximation factor is $\rho_{ST} = \ln 4 + \epsilon \approx 1.38$ Byrka et al. (2013); Goemans et al. (2012).

Theorem 2.3.1 (Byrka et al. (2013); Goemans et al. (2012))**.** *For any constant $\epsilon > 0$, there is a polynomial-time $(\ln 4 + \epsilon)$-approximation algorithm for the EWST problem.*

Reduction 3. Let V' and E' be the vertex and edge set of the instance we reduce to. Let $c' : E' \to \mathbb{R}^+$ be the edge weight function. For each node $v \in V$ in HStT, we create a pair of nodes v_h, v_l. Let V' be the set of all newly created nodes. Define the set of terminals $R' := \{v_l : v \in V\}$. For each edge $e = (u, v)$ in HStT, we add to E' the following edges $(u_h, v_h), (u_l, v_l), (u_h, u_l), (v_h, v_l)$. Edge weights are defined as: $c'(u_h, v_h) := \lambda c(u, v), c'(u_l, v_l) := c(u, v), c'(u_h, u_l) := f_u, c'(v_h, v_l) := f_v$. Call the metric completion of this graph $G' = (V', E')$.

For ease of presentation, we define the following partition of E': $H := \{(u_h, v_h) : u, v \in V\}, L := \{(u_l, v_l) : u, v \in V\}, J := \{(v_h, v_l), v \in V\}, K := \{(u_h, v_l), (u_l, v_h) : u, v \in V\}$, where H stands for upper-level edges, L for lower-level edges, J for vertical edges and K for cross edges.

Theorem 2.3.2. *If there exists a γ-approximation algorithm for the EWST problem, then there exists a 2γ-approximation algorithm for the metric HStT problem.*

Proof. First, we show that from a hub Steiner tree T in G, we can construct a Steiner tree T' in G' whose cost is the same as T. Next, we show that for any Steiner tree T' spanning R' in G', we can construct a hub Steiner tree T in G with total cost at most twice the cost of T'.

For the first part, we define a tree T' from T by including all upper-level and lower-level edges in T in addition to each edge of the form (u_l, u_h) that corresponds to installing a hub u in T. Then T' is the required Steiner tree in G'. See Figure 2.4 for an illustration.

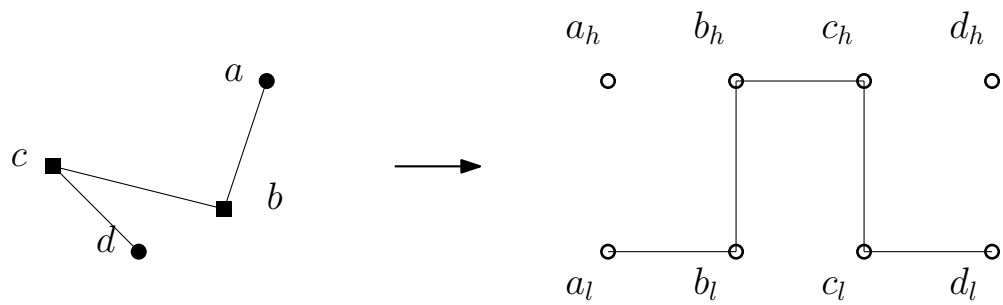


Figure 2.4: Convert a HStT to a Steiner tree where the terminal set $R = \{a, b, c, d\}$. On the left, squares (disks, resp.) indicate hubs (non-hubs, resp.). On the right, the corresponding Steiner tree uses two vertical edges, two lower-level edges and one upper-level edge.

For the second part, we first partition edges of T' into four sets as follows. Define $E_H := H \cap T', E_L := L \cap T', E_J = J \cap T', E_K = K \cap T'$. Recall that for each edge $(u_h, v_l) \in E_K$, there exists a shortest path $P_{u_h v_l}$ from u_h to v_l realizing the distance on this edge which only uses edges from $H \cup L \cup J$. Let $\mathcal{P}$ be the set of such paths, i.e. $\mathcal{P} := \{P_{u_h v_l} : (u_h, v_l) \in E_K\}$. To construct a hub Steiner tree in G, we install hubs $F := \{v : (v_h, v_l) \in E_J \cup \{P \cap J : P \in \mathcal{P}\}\}$. We add in all edges (u, v) such that their copies (u_h, v_h) or (u_l, v_l) is in $E_H \cup E_L \cup \{P \backslash J : P \in \mathcal{P}\}$. Let S be the graph constructed as described above. Since T' is a Steiner tree, S guarantees the connectivity for terminals R. We may assume by short-cutting edges that S is a tree. Figure 2.5 shows an example of our construction. On the left, solid lines represent edges in the Steiner tree. The dashed path between e_l and f_h represents the shortest path between these two nodes. Similarly for the dashed path between f_h and g_l. By definition S contains all solid edges except (e_l, f_h) and (f_h, g_l) which we replace by four dashed edges.

Let S_H be the restriction of S on the upper-level edges (i.e., the edges (u, v) added to S corresponding to an edge $(u_h, v_h) \in H$). S_H may have multiple connected components, each of which may contain *unhubbed nodes* (for which we do not have vertical edges of the form (u_h, u_l) in F). In Figure 2.5, the left bottom tree corresponds to S with two components (subtrees): one containing a single edge (b, c) and the other containing two edges (e, f) and (f, g) where node f is an unhubbed node. For each subtree, by doubling the tree, taking an Eulerian walk and short-cutting edges, we can construct a new subtree on only the hubbed nodes with the cost at most doubled w.r.t. the original subtree. The final solution consists of edges from

all these new subtrees, as well as edges that are contained in S but not in any of the original subtrees. In Figure 2.5, the bottom right shows this solution after postprocessing, which short-cuts the visit to node f.

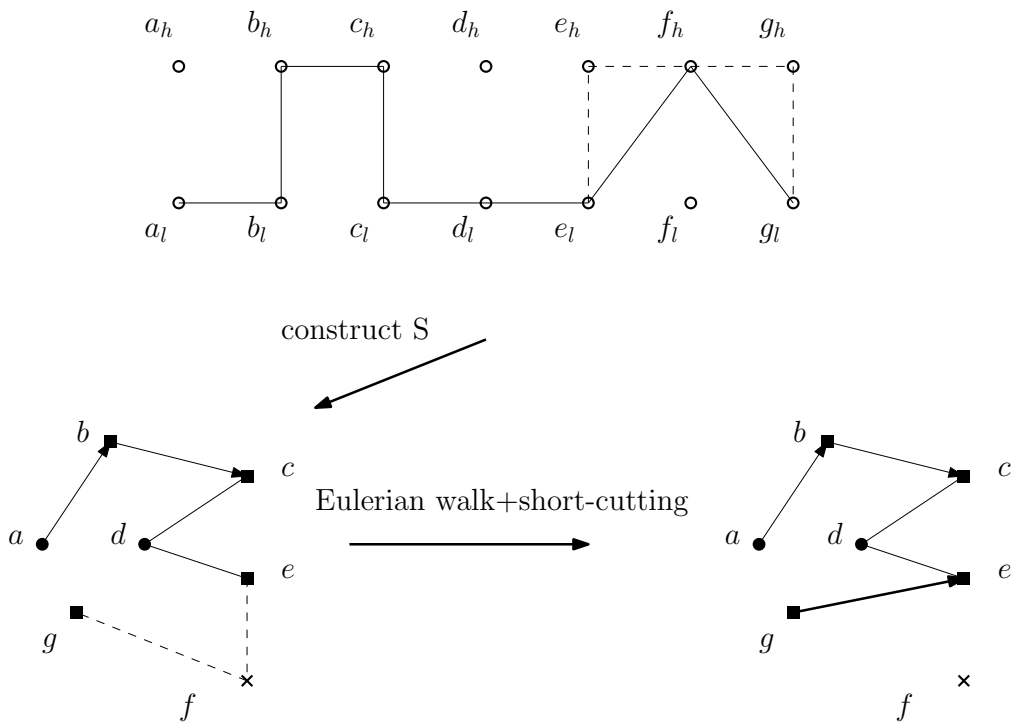


Figure 2.5: Convert a Steiner tree to a HStT where the terminal set $R = \{a, b, c, d, e, g\}$. By replacing solid edges with shortest paths, we construct S (bottom left). Its hub level restriction S_H contains two components (subtrees): one containing a single edge (b, c) and the other containing two edges (e, f) and (f, g) where node f is an unhubbed node. By postprocessing (doubling tree edges, taking Eulerian walks and short-cutting on S_H), we obtain a valid HStT (bottom right).

Notice any two components are connected by our construction, which implies that this solution is connected. Recall S spans the terminal set R. As a result, the solution also spans R. In particular, it means the solution is a valid HStT whose cost is at most twice the cost of the original Steiner tree. The theorem is then proved by combining the two parts.

□

We get the following corollary from Theorems 2.3.1 and 2.3.2.

Corollary 2.3.3. *There is a polynomial-time $2\rho_{ST}$-approximation algorithm for the metric HStT problem, where $\rho_{ST} = \ln 4 + \epsilon$ for any constant $\epsilon > 0$.*

2.4 Approximation Algorithm for HND

In this section, we first design an approximation algorithm for the CHLP (see Definition 2.1.1) and use this to design an approximation algorithm for the HND.

Approximation algorithm for CHLP. For the CHLP we design a constant approximation algorithm by connecting it to the *uncapacitated facility location* (UFL) problem. For our purposes, the UFL problem is defined as follows.

Definition 2.4.1 (Uncapacitated facility location (UFL))**.** Given a graph $G = (V, E)$ with edge cost $c(i, j)$ for each edge $(i, j) \in E$, the demand d_i for each node $i \in V$, and the hub open cost f_i for $i \in V$, the uncapacitated facility location problem is to find a subset of nodes $H \subset V$ to open hubs, and the node-hub assignment function $\pi : V \to H$ such that the cost $\sum_{i \in H} f_i + \sum_{i \in V} d_i c(i, \pi(i))$ is minimized.

For the CHLP, we can assume WLOG that each truck has unit capacity by scaling all demand values by M, the capacity of the truck. In contrast with the UFL that defines d_v as the demand from node v, the CHLP instead defines $W(i, j)$ as the amount of flow from i to j. The objective function of the CHLP can be written as follows:

$$\begin{aligned}
&\sum_{i,j \in V} [c(i, \pi(i)) + \lambda c(\pi(i), \pi(j)) + c(\pi(j), j)] W(i, j) + f(H) \\
=&\sum_{i \in V} c(i, \pi(i)) \sum_{j \in V} W(i, j) + \sum_{j \in V} c(\pi(j), j) \sum_{i \in V} W(i, j) + \lambda \sum_{i,j \in V} c(\pi(i), \pi(j)) W(i, j) + f(H) \\
=&\sum_{i \in V} (O_i + D_i) c(i, \pi(i)) + \lambda \sum_{i,j \in V} c(\pi(i), \pi(j)) W(i, j) + f(H)
\end{aligned}$$

In particular, if we set d_i in Definition 2.4.1 equal to $O_i + D_i$ for each $i \in V$, we see that the UFL problem is a special case of the CHLP which ignores the cost from the upper level network.

Our approximation algorithm for the CHLP is as follows: we first run the approximation algorithm for the corresponding UFL instance and obtain an approximate solution. Then we fix hub opening and assignment decisions based on the approximate solution and route the upper level demand via a clique network on hub nodes.

Lemma 2.4.1. *The approximation ratio for CHLP is* $(1 + 2\gamma_{UFL})$ *where* γ_{UFL} *is the best known approximation ratio for metric UFL.*

Proof. Let c_l, c_u, c_o be the cost of lower level, upper level and open hubs induced by π_A. Since the CHLP instance reduces to UFL when ignoring upper level cost, we have $Z_{UFL} \leq Z_{CHLP}$. By triangle inequality

we have the following bound on c_u:

$$\begin{aligned}
c_u = &\sum_{i,j\in V} \lambda c(\pi_A(i), \pi_A(j))W(i,j) \\
\leq &\lambda \sum_{i,j\in V} [c(\pi_A(i), i) + c(i,j) + c(j, \pi_A(j))]W(i,j) \\
\leq &\lambda \sum_{i\in V} (O_i + D_i)c(i, \pi_A(i)) + \lambda \sum_{i,j\in V} c(i,j)W(i,j) \\
\leq &\lambda Z_A + \lambda \sum_{i,j\in V} c(i,j)W(i,j)
\end{aligned}$$

Notice $Z_A \leq \gamma_{UFL} Z_{UFL} \leq \gamma_{UFL} Z_{CHLP}$ and the second term $\lambda \sum_{i,j\in V} c(i,j)W(i,j)$ is the flow routing cost when each node is an open hub, which is a trivial lower bound on Z_{CHLP}. Altogether $c_u \leq \lambda\gamma_{UFL} Z_{CHLP} + Z_{CHLP}$. On the other hand, notice $c_l + c_o = Z_A \leq \gamma_{UFL} Z_{CHLP}$. To sum up, we have $c_u + c_l + c_o \leq (1 + \gamma_{UFL} + \lambda\gamma_{UFL})Z_{CHLP} \leq (1 + 2\gamma_{UFL})Z_{CHLP}$. □

Remark 2.4.1. The above proof easily extends to variants of CHLP problem, e.g. capacitated hub location variant by reducing to its corresponding facility location counterpart.

Next we present our approximation result for the HND problem. The main theorem is the following.

Theorem 2.4.2. *There exists an $O(\log n)$ approximation algorithm for HND in general metric case on n-node graphs.*

The main idea of our algorithm is to reduce the HND problem to the CHLP via light graph spanner defined below.

Definition 2.4.2. Let $G = (V, E, w)$ be a weighted graph with weights $w(\cdot)$. For any subgraph $G' = (V', E', w)$ of G, let $dist_{G'}(u, v)$ be the weighted distance from u to v in G'. For a spanning subgraph $G' = (V, E', w)$, let Stretch$(G') = \max_{v,u\in V}\{dist_{G'}(v,u)/dist_G(v,u)\}$. The subgraph G' is said to be a κ-spanner for G if Stretch$(G') \leq \kappa$. G' is said to be α-light if $w(G') \leq \alpha w(MST)$, where MST is a minimum spanning tree in G.

Mansour and Peleg (1994) showed a simple greedy algorithm to construct an $O(\log n)$-light, $O(\log n)$-spanner. Based on this spanner, we show the following result:

Lemma 2.4.3. *Given an $O(f(n))$-approximation algorithm for the CHLP, there exists an $O(f(n)\log n)$-approximation algorithm for the HND problem.*

Proof. Let Z_{HND}, Z_{CHLP} be the optimal total cost for HND and CHLP respectively. Clearly $Z_{CHLP} \leq Z_{HND}$ since HND has additional integrality constraints on the upper level compared to CHLP. Let T_A be the approximate solution constructed by the given approximation algorithm for CHLP. Recall a valid HND

solution consists of open hubs, lower and upper level network. We construct such a solution by opening the same set of hubs as T_A and thus the same lower level network. For the upper level network, let H be the complete graph induced by open hubs. We construct an $O(\log n)$-light $O(\log n)$-spanner H' of H as in Mansour and Peleg (1994) and send all upper level flows along the shortest path induced by H'. We bound the upper level routing cost c_u on H' below.

For $i, j \in V(H)$, let $W(i,j)$ be the flow from i to j. Let $c(i,j)$ be the distance between i, j in H. $P(i,j)$ be the shortest path from i to j in H'. Let $c(P(i,j))$ be the length of the shortest path. Recall that we assumed each truck has unit capacity. For each edge $e \in E(H')$, let $x(e)$ be the total flow along edge e, i.e. $x(e) = \sum_{i,j \in V(H'): e \in P(i,j)} W(i,j)$. Let $c(MST(H))$ be the cost of an MST on H where edge weight equals the distance between two endpoints. Then

$$
\begin{aligned}
c_u &= 2 \sum_{e \in E(H')} \lambda \lceil x(e) \rceil c(e) && (2.1)\\
&\leq 2\lambda \sum_{e \in E(H')} (x(e)+1) c(e) && (2.2)\\
&\leq 2\lambda \sum_{e \in E(H')} x(e) c(e) + O(\lambda \log n) c(MST(H)) && (2.3)\\
&= 2\lambda \sum_{i,j \in V(H')} W(i,j) c(P(i,j)) + O(\lambda \log n) c(MST(H)) && (2.4)\\
&\leq O(\lambda \log n) \sum_{i,j \in V(H')} W(i,j) c(i,j) + O(\lambda \log n) c(MST(H)) && (2.5)
\end{aligned}
$$

Inequality 2.3 holds because H' is $O(\log n)$-light. Equality 2.4 follows by the definition of $x(e)$ and a change of summation. Inequality 2.5 follows because $c(P(i,j)) \leq O(\log n) c(i,j)$ for the $O(\log n)$ spanner H'. Note $\sum_{i,j \in V(H')} W(i,j) \lambda c(i,j)$ is exactly the upper level routing cost of our $f(n)$-approximate solution T_A to CHLP and $\lambda MST(H)$ is a lower bound for this cost since any valid routing requires connectivity. To wrap up, we have $c_u \leq O(\log n) f(n) Z_{CHLP}$.

Recall the total cost of an HND instance consists of the cost of opening hubs, lower level and upper level routes. The sum of the first two parts are exactly the costs of the corresponding part in T_A. Therefore the result follows with the above bound on the upper level cost and the fact that $Z_{CHLP} \leq Z_{HND}$. □

Now it is easy to derive the proof of the main theorem 2.4.3.

Proof of Theorem 2.4.2. Lemma 2.4.1 shows that $f(n)$ in lemma 2.4.3 is at most $(1+2\gamma_{UFL})$. The theorem follows by replacing $f(n)$ in lemma 2.4.3 by $1 + 2\gamma_{UFL}$, where $\gamma_{UFL} = 1.488$ (Li, 2011). □

2.5 Heuristic Algorithms for HND

In this section, we propose an evolutionary framework based on theoretical insights from Section 2.4 and a novel MIP model for the HND, which will be outlined in Section 2.5.1. Our evolutionary framework starts with a set of initial solutions, typically referred to as the initial population. In our implementation, the initial population consists of solutions from two heuristic algorithms described in Section 2.5.2 – one based on the MIP model and the other based on the UFL problem. Each individual in this population has a corresponding genetic representation. In our implementation the genetic representation of each individual is a binary vector indicating hub locations and client-hub assignments. In Section 2.5.3, starting from the initial population, the framework iteratively performs two types of operations, crossover and mutation, to improve solution quality. A crossover step creates new individuals by combing two or more 'genes' in a meaningful way. A mutation step creates new individuals by modifying the 'gene' of one individual. This phase is terminated when population diversity decreases below a user-specified threshold. Finally, we generate and solve restricted MIP instances by fixing hub locations from the genes of final individuals.

2.5.1 MIP Model for HND

We adapt a flow-based model for the hub location problem (Contreras and Fernández, 2012) to take the upper level network into consideration. Our MIP formulation is presented below. W_{ij} is the amount of flow from i to j, O_i is the total out-flow from i, D_i is total in-flow into i, c_{ij} is the lower level cost of sending one unit flow along edge (i, j), f_i is the hub opening cost at i, M is the capacity of an FTL truck. z_{ip} is the indicator of assigning i to p and we use z_{ii} as an indicator for opening hub i. f^i_{pq} is the fraction of *total* out-flow from i along arc pq and t_{pq} is the number of FTL trucks running roundtrips on edge (p, q).

Two types of costs are considered, namely hub opening cost and routing cost. The first part equals $\sum_i f_i z_{ii}$. The second part consists of upper and lower level routing cost. From previous section the lower level network is assumed to be a collection of star networks, therefore its cost equals $\sum_i (O_i + D_i) \sum_p z_{ip} c_{ip}$. The upper level cost is due to running roundtrip trucks on each edge: $\sum\limits_{p<q} 2\lambda M c_{pq} t_{pq}$.

$$\min \quad \sum_i f_i z_{ii} + \sum_i (O_i + D_i) \sum_p z_{ip} c_{ip} + \sum_{p<q} 2\lambda M c_{pq} t_{pq} \tag{2.6}$$

$$\text{s.t.} \quad z_{ip} \leq z_{pp}, \quad \forall i, p \tag{2.7}$$

$$\sum_p z_{ip} = 1, \quad \forall i \tag{2.8}$$

$$f^i_{pq} \leq z_{pp}, \quad f^i_{pq} \leq z_{qq}, \quad \forall p, q \tag{2.9}$$

$$(\sum_q f^i_{pq} - \sum_q f^i_{qp}) O_i = O_i z_{ip} - \sum_q W_{iq} z_{qp}, \quad \forall i, p \tag{2.10}$$

$$M \cdot t_{pq} \geq \sum_i f^i_{pq} O_i, \quad M \cdot t_{pq} \geq \sum_i f^i_{qp} O_i, \quad \forall p < q \tag{2.11}$$

$$f^i_{pq} \geq 0, \quad z_{ip} \in \{0, 1\}, \quad t_{pq} \in \mathbb{Z}^+ \tag{2.12}$$

Main constraints (2.10) imply flow conservation: the left hand side is the net flow originating from i at vertex p – if i is assigned to p, the net flow equals i's total demand O_i minus i's demand to those assigned to p which is $\sum_q W_{iq} z_{qp}$; if i is not assigned to p, the net flow should be zero (in equal out) minus i's demand to those assigned to p which is again $\sum_q W_{iq} z_{qp}$. Constraints (2.7), (2.8) are standard assignment constraints. Constraints (2.9) ensure the upper level network consists of only hub nodes. Constraints (2.11) ensures the total capacity on edge (p, q) to route the required amount of flow. This is where we incorporate the 2-cycle constraint that all truck routes are in anti-parallel arcs, which is critical to making the problem practical so that FTL trucks may be sourced, maintained and operated from home hubs.

2.5.2 Heuristic Algorithms for Initial Population

The initial population is constructed by the following two heuristic algorithms, both of which are inspired by the importance of hub locations for designing approximation algorithms (Section 2.4).

- *Prioritized Search* (PS). We first ignore all integrality constraints except those on variables representing hub locations. We search for good hub locations by solving the resulting (relaxed) MIP model within a time limit chosen a priori. Then we fix hub locations, restore the remaining integrality constraints and solve the resulting problem to near-optimality.
- *Facility Location-based Search* (FLS). We search for good hub locations and client-hub assignments by solving a family of UFL instances.

Prioritized Search

Recall the proof of Theorem 2.4.2 suggests the importance of hub locations. Based on this, we propose a simple MIP-based heuristic algorithm prioritizing the search for near-optimal hub locations. We first relax all integrality constraints except those on variables related to hub locations. We solve the resulting (relaxed) MIP model within a predetermined time limit and store the best choice of hub locations. We then fix the hub locations, restore those relaxed integrality constraints and solve the restricted problem to near-optimality. For ease of notation, we define these two related problems below:

Definition 2.5.1. The *hub choice HND* (HC-HND) problem is a relaxed version of the HND, where all variables except those representing hub locations in model 2.6 are relaxed to be fractional, i.e., we only require hub opening decisions z_{ii}'s (defined in Sect. 2.5.1) to be integral.

Definition 2.5.2. The *fixed hub HND* (FH-HND) problem is the HND problem restricted to a given set of open hubs.

Our algorithm outlined in Algorithm 1 starts by solving the HC-HND until the MIP gap α_1 is achieved. Using Gurobi 7.5 solution pool option, we store the best k sets of hub locations and solve the corresponding the FH-HND problem induced by each set of hub locations. In our implementation, we set $\alpha_1 = \alpha_2 = 0.02, k = 2$.

Algorithm 1: Prioritized search

Input: α_1 : optimality gap for HC-HND; α_2 : optimality gap for FH-HND; k : number of hub choices to store

Output: P : a set of solutions

```
Solve HC-HND to optimality gap α1
Store in S k best sets of hub locations from the solution pool
for each set of hub location in S do
    Solve FH-HND induced by the hub location to optimality gap α2
```

Facility Location-based Search

We design another heuristic method for finding good solutions for larger instances. Inspired by our proof ideas for CHLP in Lemma 2.4.1, we solve a family of parametrized UFL instances to obtain a set of near-optimal solutions. Recall Lemma 2.4.1 essentially uses an UFL instance to construct an upper bound for CHLP. Its cost is directly connected to the cost of lower level and hub opening. However, the cost of upper level network is overlooked. We deal with this problem in a heuristic fashion: the routing cost of UFL, same as the cost of lower level network for HND is multiplied by a factor $\theta > 0$, which results in a parametrized family of UFL instances, denoted by UFL(θ). A finite set of UFL instances is thus generated by varying θ over a set of discrete values. We then solve each instance and obtain the corresponding assignment and hub opening decisions. This method is outlined in Algorithm 2. In our implementation, the set of values for θ is $\{0.2, 0.4, \ldots, 1.6\}$.

Algorithm 2: Facility location-based search

Input: L : a list of positive values, k : number of feasible solutions to store

Output: P : a set of solutions

```
P ← ∅ ;
for θ ∈ L do
    Solve UFL(θ) and store hub locations and client-hub assignments;
Add the best k solutions to P
```

2.5.3 Evolutionary Framework

A generic evolutionary framework makes use of two important operations: the crossover operation and the mutation operation. The crossover operation is aimed at generating better solutions from a solution pool. This corresponds to the concept of exploitation in a generic search algorithm. The mutation operation is aimed at diversifying the current population, therefore corresponding to the concept of exploration in a generic search mechanism. Unlike traditional evolutionary frameworks where crossover and mutation are inexpensive operations, in order to maintain MIP feasibility and cater to the specific problem domain, we adapt the idea of *very large neighborhood search*. More specifically, in the crossover step a set of individuals are selected from the pool. We fix variables whose values agree in all chosen individuals and allow the values for other variables to be determined by solving a restricted MIP. Similarly, in the mutation step a random subset of variables are fixed. The remaining variables are determined by solving a restricted MIP. The solving process of the restricted MIP in both operations may be terminated early to limit the computational cost. A new individual is added to the population as soon as it is generated due to the expensiveness of each operation.

The probability for either crossover or mutation to happen is dynamically adjusted according to the population diversity which is calculated by the average L^1 distance of hub location variables between the best individual and others. We choose to measure the distance of only hub location variables because once they are fixed, the remaining problem can then be solved to a desirable optimality gap in a reasonable amount of time. The threshold for changing mutation and crossover probability is calculated as the ratio of current diversity over previous diversity. If it is less than the current crossover probability, we increase mutation probability and decrease crossover probability. If it is greater than the reciprocal of current mutation probability, we decrease mutation probability and increase crossover probability. During each iteration we always select two individuals for crossover. The first individual is selected at random, the second is selected at random among those better than the first one, which introduces a slight bias for selecting better individuals. Every five iterations all individuals are selected for crossover, i.e. we fix those values where all individuals agree and solve a restricted MIP. The algorithm terminates when the population diversity drops below a certain threshold. In our implementation, the initial mutation and crossover probability are set to be 0.7 and 0.3 respectively. The initial increase or decrease of this probability is 0.1, which is decreased by a factor of 0.25 each time the probability is dynamically adjusted. The termination diversity threshold δ is set to be 1. We use Gurobi 7.5 to solve all MIP models. All optimality gaps stated below are calculated w.r.t. the optimal value of each LP relaxation.

Algorithm 3: Evolutionary framework

Input: P : a set of solutions ('individuals'); p_m: mutation probability; p_c crossover probability; δ: diversity threshold for termination

```
while Population diversity > δ do
    Perform mutation with probability p_m;
    Perform crossover with probability p_c;
    Calculate the current population diversity;
    Adjust mutation and crossover probability based on the change of diversity
```

2.6 Numerical Experiments

In this section, we first describe our dataset and parameter settings in Section 2.6.1. We then compare the population quality from two heuristic algorithms proposed in Section 2.5.2. Finally we report the solution quality from our evolutionary framework. Computational experiments were run on a Lenovo laptop with 2.80GHz Dual-Core Intel Core i7 and 24GB memory.

2.6.1 Data and Parameter Description

The network data used in our computational experiments is based on a real-world LTL trucking network, consisting of approximately $560k$ ft^3 of volume to be shipped between 100 locations. Vehicle and facility costs are set to industry averages. More precisely, we assume a fully loaded cost of USD 1.50 per mile for a standard 40-ft trailer and fully loaded daily operating cost (administrative labor, maintenance, rent or depreciation of facility and equipment) of USD 1000 per transshipment hub. We note that our simplifying assumption here that facility cost is independent of the facility size and volume handled may seem somewhat unrealistic - however, in practice facility cost can be approximated reasonably well by a simple linear combination of some fixed cost component and a (non-location dependent) unit cost per unit capacity required for the volume handled. As the latter is more or less a constant in our objective (items will be handled in exactly one sending and receiving hub and cross-docking operations along the mainhaul route tend to require less work than the initial receiving and final dispatching), it can be justified even in this realistic setting to consider only the fixed portion of the facility cost in an uncapacitated facility location problem. In order to provide some intuition of the typical tradeoff between facility and upper level transport cost in a real-life network, in our setting opening an additional facility is equivalent in cost to $600 - 700$ miles of daily FTL mainhaul transport.

As stated before, transport on the lower level (pickup/delivery) network is typically done in milkruns - sometimes also combining pickup and delivery into joint runs and using a combination of 40-ft and smaller vehicles depending on the customers served (not every customer address will be able to physically accomodate delivery by a 40-ft truck). In order to simplify the problem, we assume here that 20-ft delivery

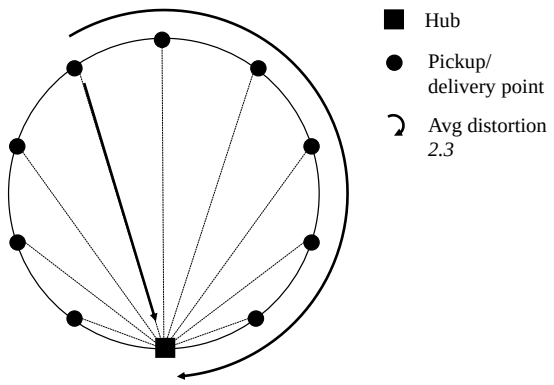


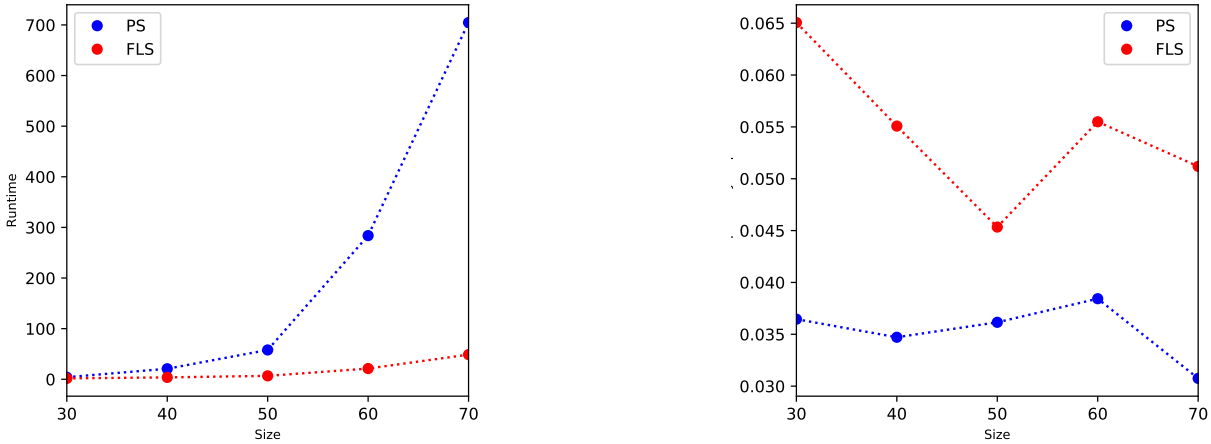
Figure 2.6: Star network as an approximation to milkruns

trucks with a maximum capacity of 1000 ft^3 and fully loaded cost of 1.30 USD per mile are deployed on joint pickup/delivery milkruns and will maintain an average utilization level of 60% over the course of each milkrun. This yields an average cost of 0.21 cents per mile per ft^3 of product moved. Finally, assuming that on average the distance traveled between pickup/delivery location and hub is roughly 2.3 times the direct distance (assuming 15 stops per milkrun equally distributed on a circle),we arrive at a cost approximation of 0.48 cents per ft^3 per mile (direct distance) for our simplified star-topology model of the lower level network. Thus, lower level LTL transport is approximately $5-6$ times as expensive as the upper level FTL transport on a per cubic feet per mile basis.

2.6.2 Comparison of Heuristic Search

We compare the performance of the two heuristic algorithms – PS and FLS proposed in Section 2.5.2. We measure the performance from two aspects: runtime and optimality gap. We perform numerical tests on problem size of $30, 40, \ldots, 70$. For each size, we generate 10 instances by randomly sampling locations from the entire dataset. Average runtime and optimality gap are reported in Fig. 2.7. For the FLS heuristic, for each test instance, we report the best solution found among a set of θ values (see Section 2.5.2). Fig. 2.7a shows that the average runtime of PS is longer than FLS for all test instances. Compared to the runtime of FLS, PS cannot scale to the real-world problem size, largely due to the fact that uncapacitated facility location problem can be solved efficiently by the modern MIP solver while solving HC-HND can be a bottleneck in PS heuristic. On the other hand, Fig. 2.7b shows that on average PS finds better solutions than FLS, which is also in line with our expectation since FLS relies on the UFL model while PS relies on partial linear relaxation of the HND model, which tends to be a more accurate representation of the original problem.

Next we study the *closeness* between the objective value of the HC-HND and the best objective value of the



(a) Runtime (s)

(b) Optimality Gap

Figure 2.7: Average runtime and optimality gap for two heuristic search algorithms (PS: prioritized search; FLS: facility location-based search)

induced FH-HND, defined as the absolute difference between these two values divided by the larger of the two. Fig. 2.8 reports that the closeness averaged over all instances is small, which means that little is lost by relaxing the integrality of assignement variables and the upper level decisions. It also indicates that hub locations play an important role in the HND problem, which is in line with the proof of Theorem 2.4.2.

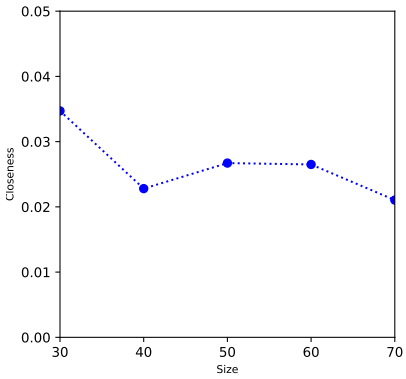


Figure 2.8: Average closeness between the objective value of HC-HND and the best objective value of the induced FH-HND

Finally we run the above two heuristic algorithms on our entire dataset of 100 nodes. FLS is able to find a feasible solution within 5% of optimality gap in less than 3 minutes, whereas it takes PS more than an hour to solve the LP relaxation.

2.6.3 Improvement from Evolutionary Framework

In this section we investigate the effectiveness of our improvement heuristic on the same samples generated in the previous section and the full real instance on 100 locations.

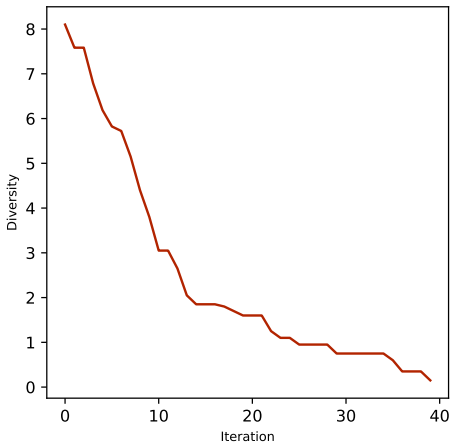
We store solutions generated by both heuristic algorithms in the previous section as the initial population and run our evolutionary algorithm based on this set. The improvement and runtime is shown in Tab. 2.2. We are able to obtain minor improvement on solution quality when the starting solutions are already very close to optimality, which is competitive to the performance of Gurobi (last column) if we allow it to run with the best solution in the starting population as a known incumbent for the same amount of time after the root relaxation finishes.

Size	Initial Gap (%)	Final Gap (%)	Time (s)	Gurobi Gap (%)
30	6.50	4.21	2.93	6.50
40	5.50	3.52	15.54	2.71
50	4.53	3.78	94.91	3.91
60	5.55	4.29	124.60	3.48
70	5.12	3.52	167.71	4.73

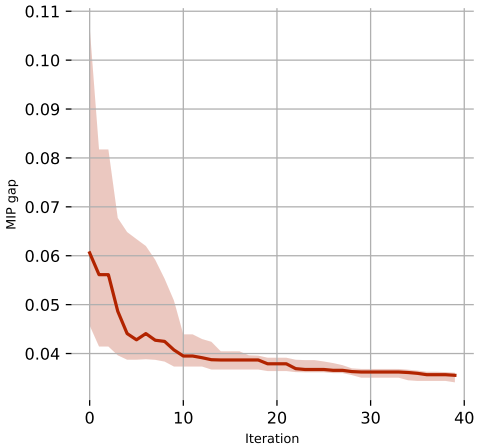
Table 2.2: Improvement on random samples

For the real-world instance, as a baseline approach we first run Gurobi solver with its parameter 'MIPFocus' set as 1 so that the solver focuses on finding feasible solutions. For our particular problem, it takes more than 80 minutes to solve the root LP relaxation. Then the solver spent more than an hour before a feasible solution of around 10% gap is found.

In comparison, we start by generating a set of solutions using facility location-based search. This process takes less than 10 minutes. Next we run the improvemen heuristic on this set of solutions. The population diversity is shown in Figure 2.9a together with a comparison of individual optimality gap between the start and the end of this genetic algorithm in Figure 2.9b. Y axis indicates the optimality of our population at each iteration which is computed with respect to LP relaxation of our MIP model. The shaded area indicates gap values between 25 percentile and 75 percentile. The population is shown to start with a median of about 6% and gradually converge to a median of below 4% over about 40 runs of our algorithm in about half an hour. We remark this step can be parallelized to further speed up the computation. Finally we pick the best three individuals and use the set of open hubs in each individual's genetic representation to generate three FH-HND instances. After solving those instances the best optimality gap decreases to 2.87%.



(a) Population Diversity



(b) MIP gap of the population

2.7 Conclusion

In this paper, we proposed a new theoretical model for a common logistics network problem. The novelty of our model is the tradeoff between reduced per unit transportation cost in the upper aggregated level and the cost of opening and operating hubs in this level. Furthermore, the difficulty arises from the combination of segments in the upper and lower levels of the network for demand routing and partitioning of the demands across hubs.

We studied two special variants of the general problem involving Steiner trees and simpler upper level networks, respectively. For the hub Steiner tree problem, we analyzed the hardness and presented approximation algorithms. Hardness results rely on reductions from the set cover problem, and approximation algorithms rely on reductions to the node-weighted or edge-weighted instances. For the clique hub location problem, we presented a constant-factor approximation algorithm using uncapacitated facility location, and combined this with a light spanner to derive a logarithmic approximation algorithm for the general problem.

Inspired by our theoretical analysis, we develop matheuristics that combine evolutionary algorithms, very large neighborhood search and mixed integer programming. Using realistic data, we demonstrate that our approximation techniques provide very good starting solutions for further heuristic search. Our methods produce solutions with an optimality gap of less than 3% in a reasonable amount of time.

Chapter 3

Combinatorial Heuristics for Inventory Routing Problems

3.1 Introduction

The Inventory Routing Problem (IRP) arises from Vendor Managed Inventory systems in which a product supplier and its retailers cooperate in the inventory planning. First, the retailers share with the supplier the demand patterns for its product and the storage costs for keeping early deliveries per retailer location. Then the supplier is responsible for planning a delivery schedule that serves all the demands on time. Naturally, the supplier wishes to minimize its routing cost and storage cost over the time horizon. This optimization problem is called IRP and has been studied extensively in, for example, Burns et al. (1985); Campbell et al. (1998); Campbell and Savelsbergh (2002); Chan et al. (1998).

In the classical single-depot IRP, a set of client demand locations in a metric containing the depot is given, and for a planning horizon of T days, a daily demand at each client location is specified. The goal is to come up with vehicle routing schedules in each of the T days to stock the client demands before they materialize. However, early stocking at a location incurs a location- and duration-specific *inventory* holding cost that are also specified. If we assume the daily replenishing vehicle has infinite capacity, the distance traveled by the vehicle in a daily route translates to a *routing* cost. The goal of IRP is to find daily vehicle schedules for the T days that deliver enough supply at each location to meet all daily demands and minimizes the sum of inventory holding costs for units supplied ahead of their demand and the daily routing costs of the vehicle, over the T days.

A generalization of IRP is the Capacitated Inventory Routing Problem (CIRP), which has K vehicles available per day and imposes a vehicle capacity U on the amount of demand that can be delivered per vehicle from the depot. We assume the demands can be split and served by more than one delivery.

Problem Definition. We now give the formal definition of IRP. In *IRP*, we are given a complete graph $K_n = (V, E)$ whose vertices are potential locations of clients and whose edge weights are determined by a *metric* $w : E \to \mathbb{R}_{\geq 0}$ ($w_{xz} \leq w_{xy} + w_{yz}$ for all $x, y, z \in V$). In a *graph metric*, the distance between every pair of vertices is the length of the shortest path between them in a given weighted graph. There is a depot $r \in V$ from which a vehicle of infinite capacity loads supply to drop off to clients. Thus, in this basic uncapacitated version, the vehicle may carry any amount of supply from the depot at each time. We have a discrete time horizon $[T] := 1, \ldots, T$ over which client $v \in V$ demands $d_t^v \geq 0$ units of supply to be delivered to it by time t. For each client $v \in V$, demand time $t \in [T]$, and potential serving time $s \leq t$, storing one unit of supply at v during time $[s, t]$ incurs a holding cost of $h_{s,t}^v$. We denote by $D(V \times [T])$ the set points (v, t) such that $d_t^v > 0$. When the context of V and $[T]$ are clear, we use D and $D(V \times [T])$ interchangeably. We call such points *demand points*. The objective is to select a tour from r through a subset of clients per time $t \in [T]$ to satisfy every demand point (no late delivery allowed) so that the total routing cost and holding cost over $[T]$ is minimized. Denote by $H_{s,t}^v$ the holding cost incurred if d_t^v is served at time s, i.e., $H_{s,t}^v = h_{s,t}^v d_t^v$. We remark that in the uncapacitated case, there is always an optimal solution such that each d_t^v is served at a single time, for if d_t^v is delivered in separate portions at times $s_1 < \ldots < s_l$, then the total cost does not increase if we move all of d_t^v to be delivered at time s_l. This is due to the infinite capacity available at the delivery vehicle.

In *CIRP*, we are given an additional constraint that K vehicles are available per time and each vehicle may carry at most U units from the depot per trip and at most one trip per vehicle is allowed per time. A feasible solution consists of the routes (with multiplicity over the edges) per day $s \leq T$ and the amount delivered to each $v \in V$ on day s such that all demands are satisfied on time.

Although existing computational research on IRP has been extensive, the instance sizes solved are still limited. The state-of-the-art method (Archetti et al., 2017) is able to solve problems of 200 clients over 6 days, detailed in Section 3.2. There have not been new conceptual ideas beyond refinements of traditional integer programming methods e.g. branch and cut (Archetti et al., 2007), branch-cut-and-price (Desaulniers et al., 2016), and matheuristics (Archetti et al., 2012, 2017).

A related problem, called Prize-Collecting Steiner Tree (PCST), will be crucially used in our heuristics for obtaining IRP solutions. The *Prize-Collecting Steiner Tree* problem has as input a graph $G = (V, E)$ with root r, edge weights $w : E \to \mathbb{R}_{\geq 0}$ and vertex penalties $\pi : V \to \mathbb{R}_{\geq 0}$. The goal is to find a tree rooted at r visiting some subset of vertices minimizing the total edge cost of the tree and the penalties of vertices not spanned by the tree.

The unifying theme of our heuristics is to reduce the search space of IRP by creating and solving PCST instances as intermediate steps to determine which clients to visit each day. PCST is a suitable intermediate problem because it is much faster to solve than IRP since it does not involve the inter-temporal constraints of IRP. Although it is NP-hard, Leitner et al. (2020) uses a dual-ascent-based branch-and-bound approach to very quickly solve PCST problems to near-optimality over $200,000$ nodes. Additionally, PCST is able

to capture the challenge of IRP's trade-off between holding cost and routing cost by using the trade-off between routing cost and penalty cost in its own objective, even though it does not have the multi-period nature of IRP. Each of our heuristics will convert the holding cost of IRP over the whole planning horizon to penalties in PCST so that the PCST solutions eventually form a good IRP solution. Next, we explain in more detail our contributions to applying ideas from PCST solutions in deriving better computational results for IRP.

Contributions.

1. We exploit a conceptual reduction proposed by Fukunaga et al. (2014) from periodic IRP to PCST and extend the ideas to the general IRP.

2. We design a new suite of algorithms for general IRP using this reduction to look for local improvements. In particular, we define a very large neighborhood search step and reduce it to a PCST problem.

3. We implement the algorithms and compare their performance using a data generation model similar to Archetti et al. (2012) but without the capacity constraints on the vehicles and clients, resulting in much faster solution time while still obtaining low optimality ratio.

4. Extending the local search ideas for IRP, we refine the algorithms to solve the capacitated IRP (CIRP) by penalizing any infeasibility until reaching a solution satisfying the capacities. We implement the algorithms on randomly generated instances of CIRP using the same data generation model and obtain similar results, with slightly larger ratios than the uncapacitated case.

Techniques. Our heuristics are inspired by the approximation algorithm for the periodic IRP in Fukunaga et al. (2014). In the periodic IRP, we are given additionally a set of frequencies $f_0, \ldots, f_k$ to assign to the clients. Assigning a client v to frequency f means that v must be visited exactly every f days. A feasible solution will choose a frequency from the available set for each client and produce a delivery schedule that obeys the assigned frequencies. Thus, the periodic IRP is more restrictive than the general IRP we consider here. Fukunaga et al. (2014) exploits the restriction of periodic schedules by reducing the periodic IRP to the Prize-Collecting Steiner Tree (PCST) problem such that the holding costs are simulated by the penalties of PCST. The idea of the reduction is to create a copy of the input graph per frequency f_i. For the ith copy of the graph, scale the edge costs by roughly T/f_i because the clients assigned frequency f_i are to be visited $\lfloor T/f_i \rfloor$ many times in the schedule. Denote by v_j the copy of v in the jth graph. To capture the holding costs of periodic IRP, they set the penalties $p(v_j)$ so that $\sum_{j=0}^{i-1} p(v_j)$ is the holding cost for v if v is visited every f_i days (Section 3.3.1 contains an example of how we adapt this to our setting). In this way, the connection cost and penalty cost of the PCST instance correspond to the routing cost and holding cost of the periodic IRP instance. See Fukunaga et al. (2014) for more details.

We adapt their idea to propose three types of heuristics that each take advantage of solving the easier PCST problem. First, our local search heuristics use PCST to quickly perform large neighborhood search among

the potential improvements per round. In our work, we designed a new greedy construction heuristic for (uncapacitated) IRP using a reduction of the greedy step to a PCST problem. Our greedy heuristic uses PCST to determine the best density demand set to cover each round. The density of a set of demands for a specified delivery day is the ratio of the total cost to satisfy the demand on the delivery day to the total units of demands satisfied.

We also adapt the primal-dual method to design a primal construction heuristic for uncapacitated IRP. While the algorithm proceeds using a reverse waveform method introduced in earlier work of Levi et al. (2006), we introduce a new additional step of choosing routes in each period by solving an appropriately defined PCST. The primal-dual heuristic uses PCST to guide the growth of the dual values and identify the set of demands to serve. Since the performance of greedy and primal dual is dominated by that of the local search heuristics, we defer the details of greedy and primal dual heuristics to the Appendix A.1 and A.2. We describe the motivation for using PCSTs in Section 3.3.

For the capacitated case, we identify a set of knapsack constraints to add to the PCST instance. The Lagrangian relaxation is solved via the classic subgradient ascent method to search for a high-quality feasible solution. Each Lagrangian subproblem corresponds to a perturbed PCST instance which can be solved very quickly to optimality using the solver developed by Leitner et al. (2016).

Results. We measure the quality for a solution value UB by its optimality gap $(UB - LB)/UB$ where the lower bound LB is computed from a mixed integer programming (MIP) formulation (the details can be found in the Online Supplement). The MIP may be terminated early due to certain criterion, which is detailed in Section A.3 and Section 3.6. For uncapacitated IRP, the MIP is terminated when the optimality gap reaches 10%. For capacitated IRP, the MIP is terminated at one hour.

Uncapacited IRP: Among a suite of combinatorial heuristics we implemented, the Prioritized Local Search outperforms others. Further extensive computational study on this heuristic shows that it achieves a median gap between 5% to 7% across all instances and solves faster than the MIP by more than two orders of magnitude.

Capacitated IRP: Given the effectiveness of the Prioritized Local Search heuristic, we investigated the performance of a modified version of it, which obeys the truck capacity. We found that the optimality gap decreases as we increase the number of clients and the length of the time horizon. In particular, we were able to achieve a median gap lower than 20% when the number of locations is 70 and the length of the time horizon is 9 days, while the MIP is unable to find any reasonable solution within an hour when the number of trucks is 3 or 4.

Paper Outline. The remaining sections are organized as follows. In Section 3.2, we describe related work. Section 3.3 describes three local search heuristics for uncapacitated IRP. Section A.3 shows the computational results of the various heuristics across a range of input parameters on uncapacitated IRP instances. Section 3.5 provides modified local search heuristics for CIRP. Finally, Section 3.6 shows the

results of the local search for CIRP.

3.2 Related Work

Methods for Capacitated IRP. For the capacitated single-vehicle deterministic IRP, Archetti et al. (2007) introduced and found exact solutions for *small* benchmark instances of up to 50 clients over 3 days and up to 30 clients over 6 days. Later Archetti et al. (2012) gave a heuristic combining tabu search with MIPs that found near-optimal solutions to the small instances. They also improved upper bounds on *large* instances of 200 clients over 6 days in the case of order-up-to-level policy. Recently Avella et al. (2018) provided reformulations under the maximum level policy with new valid inequalities and effective separation methods. For the more general multivehicle case, Archetti et al. (2014) present and compare several formulations as well as valid inequalities for capacitated IRP. Adulyasak et al. (2014) present branch-and-cut algorithms for a more general setting called multivehicle production and inventory routing problem. Desaulniers et al. (2016) provide a branch-price-and-cut algorithm, finding 54 new optima out of the 238 instances from Archetti et al. (2012) with unknown optima. Archetti et al. (2017) give a metaheuristic that solves MIPs both before and after tabu search. To initialize a solution, they formulate MIPs of different strengths and choose the MIP based on the instance size, stronger MIP for smaller instances. Then the tabu search adds, deletes, or moves visits. If the instance was small, the MIP after the tabu search fixes some variables to integer values based on how often the variable is 0 or 1 among the solutions from the search. For large instances, the routes from the tabu solutions are included as route variables in the MIP. They were able to improve the upper bound on 224 of the 240 large instances. Adaptive large neighborhood search has been used on IRP (Coelho et al., 2012a) and Multi-vehicle IRP (Coelho et al., 2012b). Large neighborhood search has also been investigated on other variants of IRP (Shirokikh and Zakharov, 2015; Aksen et al., 2014; Goel et al., 2012). The use of Lagrangian methods for discrete optimization is extensively studied in the literature. We refer the interested reader to the two excellent surveys by Fisher (1981) and Shapiro (1979) and references therein. For the capacitated IRP, we follow the classic approach outlined in Held et al. (1974) for the Lagrangian relaxation using subgradient ascent to search for a feasible solution. Lagrangian methods have been applied to variants of IRP, see e.g. Chien et al. (1989), Shen and Qi (2007) and Yu et al. (2008).

To the best of our knowledge, there are no computational studies specifically geared towards the uncapacitated IRP considered here. However, as a starting point, we test our heuristics on uncapacitated instances having the same parameter values (except for the vehicle and inventory capacities that we ignore) as the large instances of size 200 by 6 in Archetti et al. (2012). In the case of capacitated IRP (CIRP), our results are not directly comparable with the aforementioned existing results since we only consider the truck capacity, but not the inventory capacity. We tested on instances of size up to 70 locations, 9 days and 4 vehicles including vehicle capacities (but not inventory capacities), with all other parameters remaining the same as Archetti et al. (2012). To evaluate the quality of our heuristics, we need to reduce the size of the instances in order to

obtain lower bounds within reasonable time from the MIP for CIRP.

Prize-Collecting Steiner Trees and Tours. Since we will use the solvers for PCST in our implementation, we also review previous work on PCST. Specifically, we use the dual-ascent-based branch-and-bound solver of Leitner et al. (2020) for PCST, which reaches near-optimality and performs the fastest on many categories of instances from the DIMACS Challenge. Prior to this, Ljubić et al. (2006) and Fischetti et al. (2017) provided the most competitive algorithms to find exact solutions for PCST. Whenever we obtain a tree from the solver (Leitner et al., 2020), we convert the tree to a tour by running the TSP solver Concorde (Cook, 2015) on the subset of vertices spanned by the tree.

Theoretically, our greedy algorithm will solve a PCST problem to find low density set covers. The solution to each PCST instance will come from the primal dual algorithm for PCST (Goemans and Williamson, 1995), whose analysis is used in eventually proving a logarithmic factor guarantee of our greedy algorithm for IRP.

Besides our application of PCST to IRP, PCST and the related Prize-Collecting Steiner Forest (PCSF) problem have been applied in a variety of contexts. For example, PCST was used to extract seismic features when the features form a small number of strong connected components (Schmidt et al., 2015). In a similar approach, the sparse recovery problem was solved by reducing to PCSF (Hegde et al., 2016). In computational biology, solving PCSF identified high-scoring hits from RNA-interference that are connected (Berger et al., 2013), linked yeast hits with hidden human genes (Khurana et al., 2017), and determined simultaneously acting pathways in biological networks (Tuncbag et al., 2013).

Approximation Algorithms for IRP. On the theoretical side, approximation algorithms for special cases of IRP have been studied but mostly for the uncapacitated versions that we study here. IRP on general metrics has a $O(\frac{\log T}{\log \log T})$-approximation by Nagarajan and Shi (2016) and an $O(\log N)$-approximation by Fukunaga et al. (2014). For variants of periodic IRP, Fukunaga et al. (2014) provide constant approximations. Another special case of IRP is the joint replenishment problem (JRP). JRP is equivalent to IRP on a two-level tree metric, where the first level has one edge of cost K_0 and the second level consists of children of the first level, with an edge of cost K_i for each commodity i. JRP is known to be NP-hard (Arkin et al., 1989) and APX-hard (Nonner and Souza, 2009). On the positive side, Levi et al. (2006) give a 2-approximation via a primal dual approach. The approximation factor was reduced to 1.8 in Levi et al. (2008) by LP rounding. Bienkowski et al. (2014) improve the approximation factor further to 1.791 by randomized rounding.

3.3 Uncapacitated Local Search Heuristics

To solve the uncapacitated problem, we consider seven combinatorial heuristics: DELETE, ADD, Prioritized Local Search, greedy, pruned greedy, primal dual and pruned primal dual. Among them, the Prioritized

Local Search and ADD, which are based on solving a family of PCST problems, are shown to be the most competitive by our computational experiments in the next section.

We briefly describe how a solution to the PCST problem can be used to implement a local search that identifies tours to add. We use the total edge cost of a tree as a proxy of routing cost since it is generally much easier to construct a tree than a tour. At each iteration of our heuristic, we encode as the penalty in a PCST instance the change in the holding cost of adding or removing a visit to a client on a day. An optimal PCST solution on that day will trade-off the holding cost change with the routing cost change. To obtain an IRP solution at the end, we convert each day's tree into a tour visiting the clients spanned by the tree by calling Concorde on the graph induced by the spanned clients. We will apply the same conversion from trees to tours in all other heuristics as well.

In Section 3.3.1, we introduce a local search procedure that applies only DELETEs. Section 3.3.2 describes a local search procedure applying only ADDs. In Section 3.3.3 we introduce the DELETE-ADD operation and define a more refined local search that applies all three operations in order of complexity.

3.3.1 DELETE

In the delete-only local search, we start with a feasible solution that visits everyone everyday and delete an entire day's visit as long as it makes an improvement.

We define the following notations needed for the algorithm. Let $l(v, s)$ be the latest visit before day s that visits v. Denote by T_s the existing tree on day s in the current step of the algorithm. For a subgraph $F \subseteq G$, let $E(F)$ and $V(F)$ be the edge set and vertex set of F, respectively. We use the vector $\mathbf{T}$ to represent the current set of existing trees on each day throughout the time horizon. Let $\hat{t}(v, s)$ be the latest day t such that $\mathbf{T}$ does not visit v during the time interval $[s + 1, t]$. Notice that if a visit to v on day s is removed, then each demand (v, t) with $t \in [s, \hat{t}(v, s)]$ would incur an extra holding cost of $H^v_{l(v,s),t} - H^v_{s,t}$. So we set the penalty of removing v on day s to be

$$\pi_s(v) := \begin{cases} \sum_{t=s}^{\hat{t}(v,s)} (H^v_{l(v,s),t} - H^v_{s,t}) & \text{if } v \in T_s \\ 0 & \text{else.} \end{cases} \tag{3.1}$$

If the tree T_s on day s is deleted, then the routing cost decreases by $c(E(T_s))$ and the holding cost increases by $\pi_s(V(T_s))$. So the total change in cost for deleting the tree on day s is $\Delta_{DELETE}(s) := -c(E(T_s)) + \pi_s(V(T_s))$. Finally, the operation $DELETE(s)$ removes the existing tree T_s on day s.

Denote by $c(\mathbf{T})$ the total cost of a solution $\mathbf{T}$. The *improvement ratio* of an operation is the magnitude of the change in cost induced by the operation divided by the total cost of the current feasible solution. In all of the local search heuristics, whenever we scan through the time period to find an improving step, we

will make the improving operation on the day that gives the best improvement while keeping the solution feasible. To avoid potentially long running time, we shall stop looking for improvements whenever the best possible improvement ratio is smaller than some small value, typically 0.01.

Now, we formally define the DELETE algorithm.

Algorithm 4: Local Search with DELETE

```
Initialize a feasible solution T
Let t' = argmin_{s∈[1,T]} Δ_DELETE(s)
while |Δ_DELETE(t')| / |c(T)| ≥ 0.01 do
    DELETE(t')
```

The initial feasible solution here is a "full" solution, where each day consists of a minimum cost tree that visits everyone.

3.3.2 ADD

In the add-only local search, we start with a feasible solution and we find an optimal subset of clients to add to the current subset on some day as long as it improves the total cost. To find the best subset of clients to add on a given day s, we solve an appropriately constructed PCST problem whose cost captures the rewards in terms of savings in holding cost when visits are added and the extra routing cost to connect the added visits.

We use the same definitions of $l(v, s)$, T_s, and $\hat{t}(v, s)$ from Section 3.3.1. However, the penalties now apply to the clients *not* visited on the day we are adding visits. In particular, if a visit to v on day s is added, then every demand point (v, t) with $t \in [s, \hat{t}(s, v)]$ *saves* a holding cost of $H^v_{l(v,s),t} - H^v_{s,t}$. So we set the penalties as follows.

$$\pi_s(v) := \begin{cases} \sum_{t=s}^{\hat{t}(v,s)} (H^v_{l(v,s),t} - H^v_{s,t}) & \text{if } v \notin T_s \\ 0 & \text{else.} \end{cases} \tag{3.2}$$

The total change in cost is $\Delta_{ADD}(s) := c(E(PCST_s)) - \pi_s(V(PCST_s))$, where $PCST_s$ denotes an optimal PCST solution on the instance G with penalty function π_s and edge weights defined as follows.

$$w_s(e) := \begin{cases} c(e) & \text{if } e \notin T_s \\ 0 & \text{else.} \end{cases} \tag{3.3}$$

The reason we set edge costs to 0 for the edges in T_s is that the existing tree should be free to use for

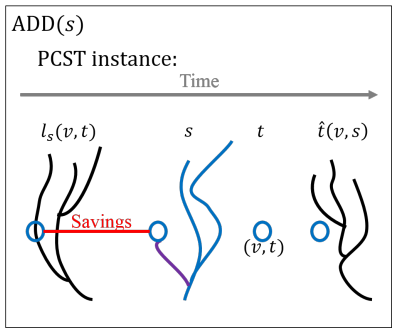


Figure 3.1: Here, we consider an ADD operation on day s. If a client v is added to the existing tree on day s (shown in blue), then any demand point at v with deadline day t within s and $\hat{t}(v,s)$ would benefit from the extra visit. The red segment represents the amount of holding cost that would be reduced for (v,t) if v was visited on day s instead of day $l(v,s)$. To reach v from the existing tree, the extra routing required is represented by the purple path connecting the blue tree to v.

connecting to the vertices that the PCST adds to the visit set. Notice that minimizing $c(E(PCST_s)) - \pi_s(V(PCST_s))$ is the same as minimizing this quantity after adding a fixed constant value $\pi_s(V)$. After this addition, the total minimization objective becomes $c(E(PCST_s)) + \pi_s(V \setminus V(PCST_s))$. Thus solving the PCST with its original objective function is consistent with minimizing $\Delta_{ADD}(s)$. The operation $ADD(s)$ adds the tree $E(PCST_s)$ to T_s covering the extra clients $V(PCST_s)$. Figure 3.1 shows how the penalty at each client captures the savings in holding cost if the client is added on a specified day.

Note that the neighborhood for the improvement step is of exponential size since each step decides which subset of vertices to add to the current tree for some fixed day. Creating and solving the appropriate PCST instance enables us to find the best improvement step quickly in practice. Unlike the DELETE step where the PCST instance evaluates good possibilities for deletes (of which there are is at most one tree per day), the ADD step uses the auxiliary PCST instance to carry out the very large neighborhood search efficiently, and represents a key innovation in our algorithm. Algorithm 5 formally describes the ADD algorithm, where the initial feasible solution used is a near-empty solution, which visits everyone only on day 1.

Algorithm 5: Local Search with ADD

Initialize a feasible solution $\mathbf{T}$
Let $t' = \text{argmin}_{s \in [1,T]} \Delta_{ADD}(s)$
while $\frac{|\Delta_{ADD}(t')|}{c(\mathbf{T})} \geq 0.01$ **do**
  $ADD(t')$

Proof. By the optimality of $\mathbf{T}$, there cannot exist a subset of nodes which after adding, the solution value is lower than the value of $\mathbf{T}$. By the definition of algorithm 5, we would be able to find a subset of nodes to add which gives the same (optimal) value as $\mathbf{T}$. □

3.3.3 Prioritized Local Search

For the prioritized local search, we start with the final solution from the ADD local search of Section 3.3.2. Then we try three operations in order of complexity. First, we look for a day for which DELETE improves the cost while preserving feasibility. If no such day exists, we look for one that ADD improves the cost on. If still no such day exists, we look for a pair (s_1, s_2) of days such that the net change in cost of $DELETE(s_1)$ followed by $ADD(s_2)$ is negative and the operation leaves a feasible solution. As long as any of the three operations makes an improvement, we continue updating the solution. Now we formally define the final pairwise operation $DELETE - ADD(s_1, s_2)$.

In the previous sections, the cost change $\Delta_{DELETE}(s_1)$ and $\Delta_{ADD}(s_2)$ were each computed relative to the existing trees $\mathbf{T}$. Here, $\Delta_{DELETE}(s_1)$ will be defined relative to the existing trees $\mathbf{T}$, but $\Delta_{ADD}(s_2)$ will be defined relative to the leftover trees after deleting T_{s_1} from $\mathbf{T}$ since we want to find the cost of adding to day s_2 right after deleting everything from day s_1. To keep the context of which solution the cost changes are computed on, we denote the new cost changes by $\Delta_{DELETE}(\mathbf{T}, s_1)$ and $\Delta_{ADD}(\mathbf{T} - T_{s_1}\mathbf{e_{s_1}}, s_2)$. Then the change in cost for the pairwise $DELETE - ADD$ is $\Delta_{DA}(s_1, s_2) := \Delta_{DELETE}(\mathbf{T}, s_1) + \Delta_{ADD}(\mathbf{T} - T_{s_1}\mathbf{e_{s_1}}, s_2)$.

We test prioritized local search starting with the solution from greedy and primal dual, respectively, as the initial feasible solution.

In the prioritized local search, the cost-minimizing pair of days for DELETE-ADD often has s_1 coinciding with s_2, although they are different occasionally. Most of the time s_1 coincides with s_2 since having both operations on the same day allows optimizing for the best visit set for that day. Note that $\Delta_{DA}(s, s) \leq 0$ for all s as the operation would choose a set that has better or equal cost as the current set on day s. As a heuristic to reduce the run time of prioritized local search, we also test the restricted version of it that only applies DELETE-ADDs to the same day, i.e., $DELETE - ADD(s, s)$.

3.4 Uncapacitated IRP Results

In this section, we examine the solution quality of our heuristic methods. We measure the quality for a solution value UB by its optimality gap $(UB - LB)/UB$ where the lower bound used for computing this gap is obtained by solving a MIP, whose detail can be found in Appendix A.3. We first show in Section 3.4.2, via a set of preliminary tests, that the Prioritized Local Search and ADD are the most competitive heuristics among others, including DELETE, greedy, pruned greedy, primal dual and pruned primal dual. Since the Prioritized Local Search builds upon ADD as explained in Section 3.3.3, we proceed in Section 3.4.3 to focus solely on the former and conduct extensive computational tests to study its performance as multiple problem parameters vary.

3.4.1 Experimental Setup

All the heuristics were implemented in C++ on an Intel Xeon processor X5680, 3.33 GHz machine with 8 GB RAM. Each heuristic uses 1 thread. A copy of the code and data used to conduct the experiments is available on Github at https://github.com/yjiao18/EuclideanIRP. The MIP was solved by Gurobi Version 7 on default settings using 8 threads. We report the runtime directly without accounting for the number of threads.

We follow the same data generation model of the largest instances tested in Archetti et al. (2012), except that our model has no capacity constraints at vehicles and store locations. More specifically, below is how our data sets were generated.

- The location of each client is sampled uniformly from a 500×500 grid
- The holding cost per day at each client is sampled uniformly from the interval $[0.1, 0.5]$
- In addition we introduce a parameter H as the scaling factor for the holding cost, i.e. the objective function is $r(S) + H \times h(S)$, where $r(S)$ is the routing cost and $h(S)$ is the holding cost of a solution S, Therefore a larger H is oriented towards a solution that pays more attention to optimizing holding cost.

3.4.2 Comparison of Different Heuristics

Recall we use N and T to denote the number of clients and days respectively. To compare the performance between different heuristics, we create three data sets, each with one parameter vary in a meaningful range. For each data set, we generated 100 test instances and computed the lower bound via a MIP (detailed in Appendix A.3, which was set to terminate when the MIP gap reaches 10%.

- In the first set, N varies from 110 to 160 in increments of 10 while H is fixed to 2.6 and T fixed to 6. We picked $H = 2.6$ because results on varying H indicated that the highest gaps of the heuristics' occur around $H = 2.6$, which means the trade-off between routing cost and holding cost is hardest there.
- In the second set, T varies from 6 to 18 in increments of 2 while $N = 50$ and $H = 2.6$. We kept $H = 2.6$ for the aforementioned reason and chose a smaller N to accommodate the increased computation time that the MIP requires on instances with large T.

In the Online Supplement, we describe the results from letting H vary from 0.01 to 6.01 in increments of 0.5 while fixing N at 100 and T at 6. We chose $N = 100$ and $T = 6$ to keep the base values the same as large instances in Archetti et al. (2012).

For each test, we solve 100 instances using each heuristic and report the boxplot of corresponding optimality gaps in Figures A.6 and A.8. The main finding is that the heuristics achieving the best gap and solution time are the Prioritized Local Search and ADD. Below we explain in more detail our findings when we vary each parameter.

Varying N

Here, T and H are fixed to 6 and 2.6, respectively. The number of clients N varies from 110 to 160 at increments of 10. Due to limited computation time, results which require MIP values are restricted to $N \leq 140$. The heuristics in order from lowest to highest gap are Local Search with ADDs and Prioritized Local Search, Pruned Primal Dual, Local Search with DELETEs, Pruned Greedy, Primal Dual, and Greedy. The gap values of the heuristics did not form any particular patterns with respect to N. Results of the gap are detailed below.

- Local search with ADDs and Prioritized Local Search both have gaps of 1.04.
- Local search with DELETEs has slightly higher gap than Pruned Primal Dual, from 1.09 to 1.1.

Varying T

Here, N and H are fixed to 50 and 2.6, respectively. The number of days T varies from 6 to 18 in increments of 2. Results that involve MIP values are available only up to $T = 16$ due to the MIP's high computation time. In order from lowest to highest gap, the heuristics are Local Search with ADDs and Prioritized Local Search, Local Search with DELETEs, Pruned Primal Dual, Pruned Greedy, Primal Dual, and Greedy. Results for the gap are detailed below.

- Local search with ADDs and Prioritized Local Search both have gaps ranging from 1.03 to 1.08.
- Local search with DELETEs' gap does not exhibit any trend with respect to T. The gap ranges from 1.03 to 1.09.

3.4.3 Comparison between the Prioritized Local Search and the Baseline MIP

Since Section 3.4.2 shows the Prioritized Local Search outperforms other heuristics, we proceed to study its solution quality as we vary two parameters N and T while keeping H fixed. We choose to vary N and T because the performance of both the MIP and the heuristic is more sensitive to these two parameters. More specifically, here are the range of values we tested.

- The number of clients $N = 40, 50, 60, 70$
- The number of days $T = 3, 5, 7, 9$

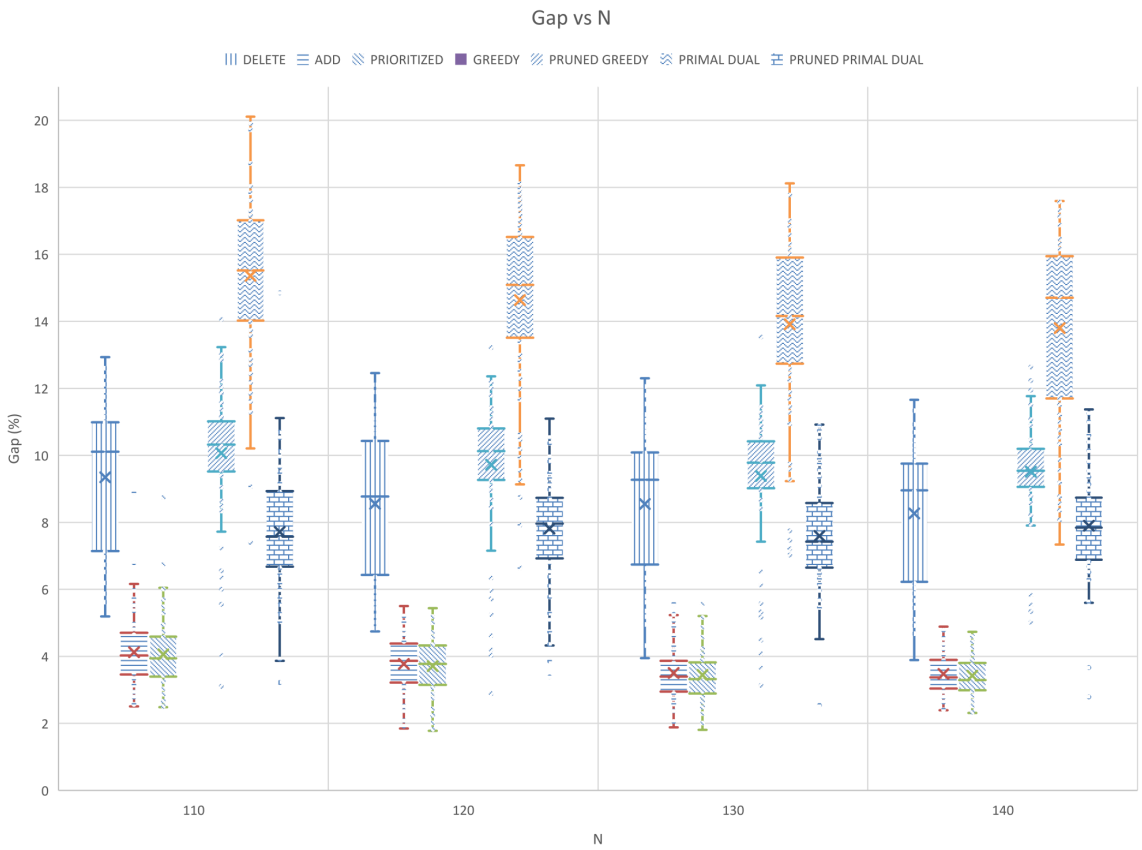


Figure 3.2: Comparison of methods for the uncapacitated case by varying the number of locations N (colors online). The fixed parameters have values $H = 2.6$ and $T = 6$. The vertical points span the gap values of the 100 instances tested for each value of N. The gaps for Delete, Add, Prioritized Local Search, Pruned Greedy, Primal Dual, and Pruned Primal Dual are shown in vertical stripes, horizontal stripes, downward diagonal stripes, upward diagonal stripes, zig zags, and bricks, respectively. Greedy is omitted in this plot due to its high gap relative to the other heuristics. See Appendix A.3 for the plot including all heuristics.

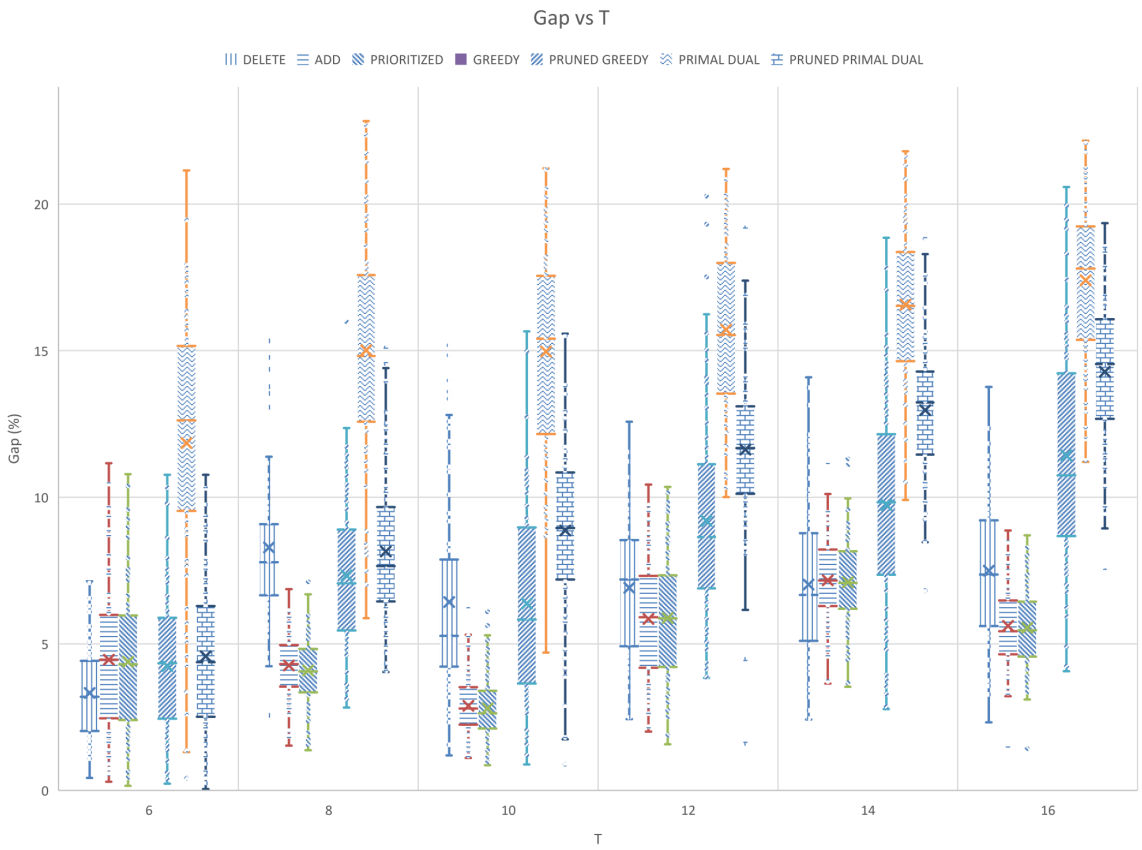


Figure 3.3: Comparison of methods for the uncapacitated case by varying the time horizon T (colors online). The fixed parameters have values $N = 50$ and $H = 2.6$. The vertical points span the gap values of the 100 instances tested for each value of T. The gaps for Delete, Add, Prioritized Local Search, Pruned Greedy, Primal Dual, and Pruned Primal Dual are shown in vertical stripes, horizontal stripes, downward diagonal stripes, upward diagonal stripes, zig zags, and bricks, respectively. Greedy is omitted in this plot due to its high gap relative to the other heuristics. See the Online Supplement for the plot including all heuristics.

For each parameter setting, we randomly generated 10 instances following the procedure in Section 3.4.1. The gap of the Prioritized Local Search is computed with respect to the lower bound found by the MIP. Each MIP is terminated when either the MIP gap reaches 10% or the solution time exceeds one hour. The speedup factor is computed as the ratio between the solution time of the MIP and the Prioritized Local Search. In Figure 3.4 our result is reported in a 4×4 grid where each cell corresponds to the gap comparison for an (N, T) pair. Similarly the speedup factor is reported in Figure 3.5. These two figures jointly summarize the relative performance of the heuristic against the MIP since the speedup factor must be evaluated along with the corresponding gaps provided by these experiments.

Recall each MIP terminated whenever the gap reaches 10%, or the solution time exceeds one hour. Figure 3.4 shows that, for any (N, T) pair, the median gap of the Prioritized Local Search is smaller than the median MIP gap and remains between 5% to 7%. The gap of our heuristic remains stable and does not show any worsening pattern as N and T increases. On the other hand, the median MIP gap is between 8% to 10% except for instances of the largest size where $N = 70$ and $T = 9$ shown in the bottom-right corner. In fact, the MIP gap of this test displays a large variation from 10% to 97%, with a median of 19.38%, whereas the average gap of our heuristic is 5.79%.

Figure 3.5 shows that the runtime of our heuristic. It can be seen that the runtime increases slightly as N and T increases, and all instances are finished within 10 seconds.

3.4.4 True Optimality Gap when MIP is solved exactly

Recall each MIP is set to terminate before the optimal solution is found. Therefore we computed the gap $(UB - LB)/UB$ of our heuristic for those data sets where an optimal solution could be found within reasonable time. Table 3.1 reports this gap for four small data sets. It shows that median gap of our heuristic is between 3% to 4%. We remark that no instance is solved to optimality by our heuristic.

N	T	Min Gap (%)	Median Gap (%)	Max Gap (%)
30	3	0.26	3.68	6.04
30	5	1.62	3.32	7.80
40	3	0.82	3.86	5.48
40	5	1.98	3.32	5.48

Table 3.1: Gap $(UB - LB)/UB$ of the Prioritized Local Search heuristic for uncapacitated instances solved to optimality. N is the number of clients and T is the number of days.

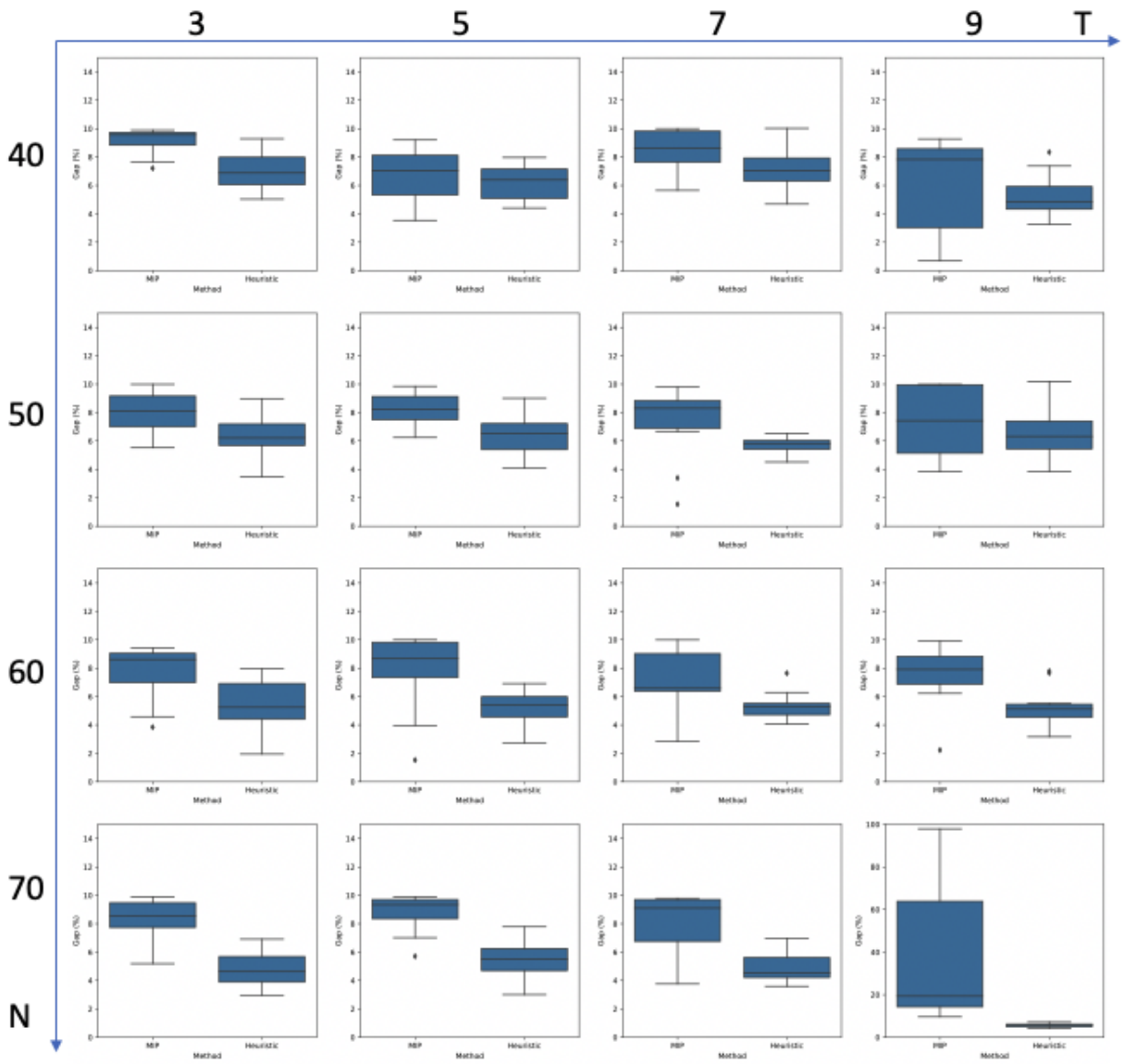


Figure 3.4: Comparison of methods for the uncapacitated case. A 4×4 grid of the gap comparison when varying N and T. Starting from the top-left corner, N increases from 40 to 70 at an interval of 10 along the vertical axis and T increases from 3 to 9 at an interval of 2 along the horizontal axis. The scales of y-axis are all the same except for the bottom-right figure.

3.5 Capacitated Local Search

Recall in Section 3.4.2 the Prioritized Local Search is shown to outperform other heuristics. Since the Prioritized Local Search is built upon two heuristics: ADD and DELETE, we explain how to modify them to incorporate the vehicle capacity. It is not hard to see that ADD and DELETE defined for uncapacitated instances can lead to infeasibility when vehicles have capacity constraints. Indeed, adding too many nodes

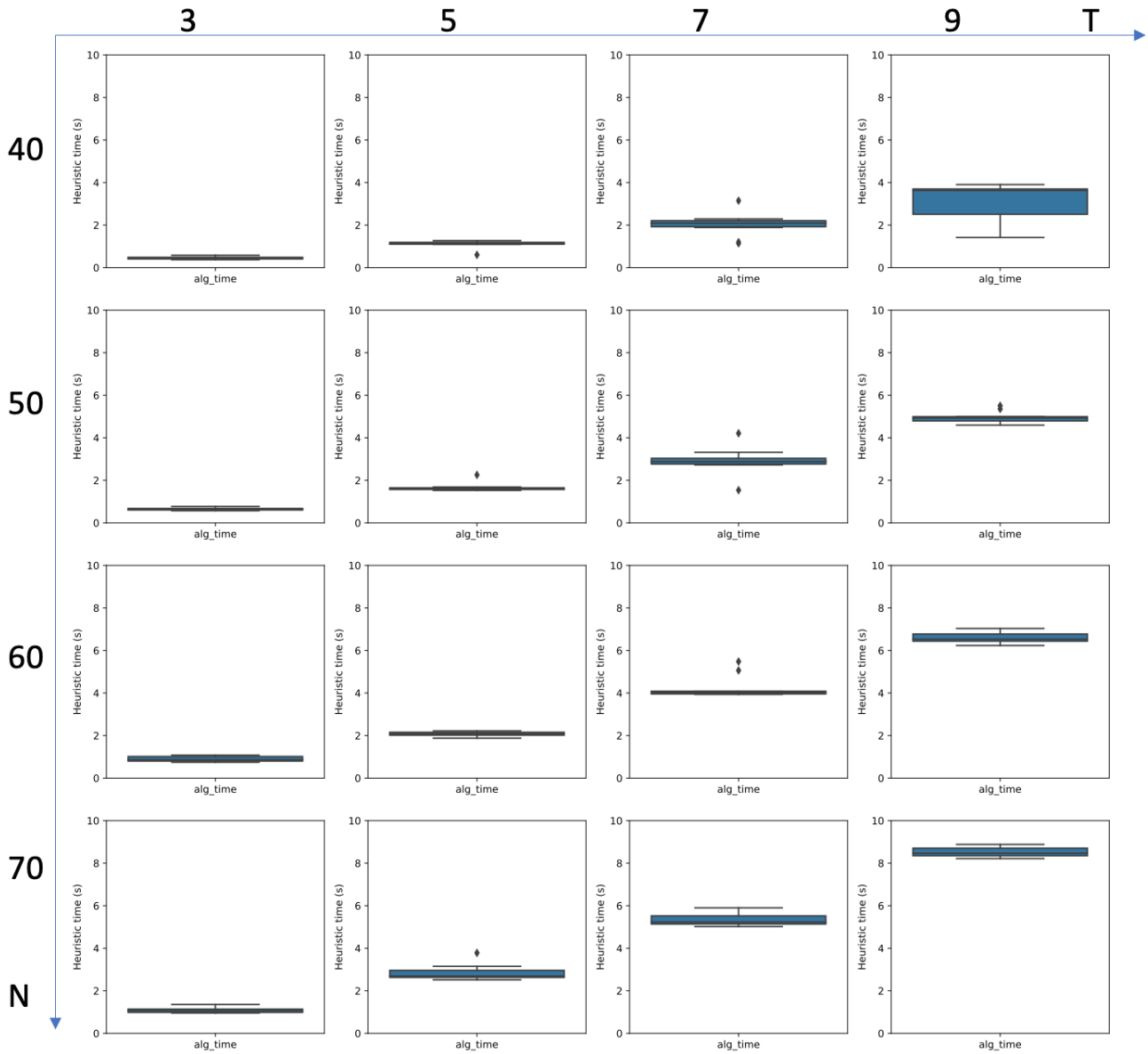


Figure 3.5: Runtime of the heuristic for the uncapacitated case. A 4×4 grid of the runtime of the heuristic in seconds when varying N and T. Starting from the top-left corner, N increases from 40 to 70 at an interval of 10 along the vertical axis and T increases from 3 to 9 at an interval of 2 along the horizontal axis.

on a particular day could make the total demand of that day exceed the total truck capacity. Similarly deleting too many nodes could lead to infeasibility on previous days. To restore feasibility, we modify ADD and DELETE based on Lagrangian relaxation. At a high level, for a fixed operation on a fixed day, we associate a knapsack constraint with each day where the capacity can be violated. Then we dualize the set of knapsack constraints and iteratively update corresponding multipliers via subgradient ascent. We remark that each iteration has a nice interpretation as solving an instance of PCST with modified penalties. The

algorithm terminates when either we reach feasibility or the maximum number of iterations allowed. If the obtained solution is still not feasible, we do not make any change to the current day and move on to perform local search for the next day. The next two sections explain the modifications for ADD and DELETE in more detail. Section 3.5.3 combines ADD and DELETE to carry out each local search step. Finally Section 3.5.4 converts PCST solution into tours which respect the capacity constraint.

3.5.1 Capacitated ADD

We use the same notation as Section 3.3.2. ADD finds a set of additional clients to deliver on a given day s. Since only day s may increase its total demand after ADD, this operation can only cause day s to be infeasible. We associate a knapsack constraint for day s. For $v \notin V(T_s)$ let $x_v = 1$ if v is added and 0 otherwise. Let $y_e = 1$ if edge e is used. Let δ_s be the slack for day s which is the difference between total capacity and the total demand on day s and D_v be the demand to deliver if v is added on day s. Let π_s be the same penalty defined in Section 3.3.2. We formulate the PCST problem with a knapsack constraint as follows:

$$\min \quad \sum_{e \in E} c_e y_e + \sum_{v \notin V(T_s)} \pi_v (1 - x_v) \tag{3.4}$$

$$\text{subject to} \quad \text{PCST constraints} \tag{3.5}$$

$$\sum_{v \notin V(T_s)} D_v x_v \leq \delta_s \tag{3.6}$$

where (3.5) refers to classic cut type PCST constraints, see e.g. Bienstock et al. (1993). We next dualize constraint (3.6) and obtain the following Lagrangian function:

$$L(\lambda) = \min\{\sum_{e \in E} c_e y_e + \sum_{v \notin V(T_s)} (\pi_v - \lambda D_v)(1 - x_v) - \lambda \delta_s + \lambda \sum_{v \notin V(T_s)} D_v, \text{ subject to PCST constraints}\}$$

where λ is the dual multiplier. Below we explain why the classic subgradient method by Held et al. (1974) can be used to guide us towards feasibility. The subgradient method updates the dual multiplier after solving the Lagrangian subproblem in an iteration. The generic update rule is

$$\lambda^{k+1} = \lambda^k + \alpha_k (Ax^k - b) \tag{3.7}$$

where k is the iteration number, the dualized constraint is $Ax \leq b$ and α_k is the step size to be discussed later. In our case, the Lagrangian subproblem is an instance of PCST with penalty of node $v \notin V(T_s)$ *decreased* by λD_v to discourage adding too many nodes. Notice this subproblem can be solved very fast in practice via the solver by Leitner et al. (2020). Therefore the proposed algorithm can be seen to decrease penalty until we reach feasibility or the maximum number of iterations allowed. The latter case indicates

the slack of the current day s may be too small to add some more nodes, so we do not make any change to the current day.

The step size used most commonly in practice is (justification of this formula is given in Held et al. (1974)):

$$\alpha_k = \frac{\mu_k(\bar{L} - L(\lambda^k))}{\|Ax^k - b\|_2^2} \tag{3.8}$$

where μ_k is a scalar satisfying $0 < \mu_k \leq 2$ and $\bar{L}$ is an upper bound on the Lagrangian relaxation value, frequently obtained by applying a heuristic to the problem. Often the sequence μ_k is determined by setting $\mu_0 = 2$ and halving μ_k whenever $L(\lambda)$ has failed to increase in some fixed number of iterations. We start with $\mu_0 = 2$ and use the above step size strategy.

3.5.2 Capacitated DELETE

Similar to capacitated ADD, we associate a knapsack constraint with each day where the total demand on that day can exceed total capacity. Because deletion could make multiple previous days infeasible, we have multiple knapsack constraints. More specifically, let Γ_s be the set of latest visit days of visited nodes on day s. Let $\mathcal{P}$ be the partition of $V(T_s)$ based on Γ_s, i.e., for $t \in \Gamma_s$, $\mathcal{P}_t := \{v \in V(T_s) :$ latest visit day of v, $l(v,s) = t\}$. Let δ_t be the slack of day $t \in \Gamma_s$. Let x, y be the same indicators as in capacitated ADD. The PCST problem with knapsack constraints can be written as:

$$\min \quad \sum_{e \in E} c_e y_e + \sum_{v \in V(T_s)} \pi_v (1 - x_v) \tag{3.9}$$

$$\text{subject to} \quad \text{PCST constraints} \tag{3.10}$$

$$\sum_{v \in \mathcal{P}_t} D_v (1 - x_v) \leq \delta_t, \quad \forall t \in \Gamma_s \tag{3.11}$$

The Lagrangian subproblem becomes:

$$L(\lambda) = \min\{\sum_{e \in E} c_e y_e + \sum_{t \in \Gamma_s} \sum_{v \in \mathcal{P}_t} (\pi_v + \lambda_t D_v)(1 - x_v), \text{ subject to PCST constraints}\}$$

We start with the initial penalty defined in Section 3.3.1 and each Lagrangian subproblem solves a PCST instance with *increased* penalties which discourages deleting too many nodes. Dual multipliers are updated in the same fashion as (3.7) and (3.8). The algorithm terminates when we reach feasibility or the maximum number of iterations. The latter case indicates the slack of the previous affected days may be too small to receive more demand due to deletion on day s, so we do not make any change to the current day s.

3.5.3 Prioritized Local Search

We start with the following initial solutions: on a fixed day, we either visit no one or visit everyone, and we do not make any visit until the accumulated demand over time exceeds the total capacity of trucks. We then perform local search on this initial solution: for ADD and DELETE, we respectively choose the day that would result in the best improvement to perform the operation. We then choose the better of the two operations. Notice we only perform one of the two operations instead of both of them sequentially – the latter option is not more effective than the former one based on our initial testing. We terminate the local search once the improvement ratio becomes less than 1% or we have run more than 1000 iterations.

3.5.4 From PCST to Feasible Subtours

Notice once the set of visits is fixed, the routing plan on each day reduces to an instance of the capacitated vehicle routing problem (CVRP). We use the following simple heuristics to solve the CVRP for each day. We break up the TSP tour obtained from Concorde into several subtours: visit the nodes in the order of the tour and form a new subtour when the accumulated demand has exceeded the total capacity. For the subset of nodes inside this new subtour, we then re-run the concorde solver and get a possibly cheaper subtour.

3.6 Capacitated IRP Results

In this section, we examine the solution quality of our heuristic method. We measure the quality of a solution value UB by its optimality gap $(UB - LB)/UB$ where the lower bound used for computing this gap is obtained by solving a MIP by the branch-and-cut algorithm – details can be found in Appendix A.3. Recall from Section A.3 that N, T, H are the number of clients, the number of days and the holding cost multiplier respectively. In this section we introduce two new parameters – K to denote the number of available vehicles and U to denote the vehicle capacity.

3.6.1 Experimental Setup

Heuristics were implemented on the same machine setup as the uncapacitated case (Section 3.4.1). A copy of the code and data used to conduct the experiments is available at `https://bitbucket.org/OrionT/irp-code/src/master/`.

3.6.2 Parameter Settings

In addition to the number of clients N and the number of days T, we add parameters K as the number of vehicles and U as the vehicle capacity. We assume each vehicle is homogeneous. The ranges of these parameters are detailed below.

- Time horizon $T = 3, 5, 7, 9$
- Number of clients $N = 40, 50, 60, 70$
- Number of vehicles $K = 1, 2, 3, 4$
- The capacity of a vehicle is set to the average demand per day multiplied by 2 divided by the number of vehicles; Here multiplying by 2 is chosen in a heuristic fashion since we expect a vehicle delivery roughly every two days.

For each parameter setting, we randomly generate 10 instances following the procedure in Section 3.4.1.

3.6.3 Performance Evaluation

In Figure 3.6, we report the optimality gap of the Capacitated Prioritized Local Search and the MIP. Figure 3.7 shows the runtime of our heuristic method. The termination criterion for the MIP is one hour. Below we summarize our findings.

- In terms of the median gap, the MIP generally outperforms our heuristic method when $N \leq 60$ and $T \leq 5$.
- For each fixed N and fixed K, the median gap of the heuristic generally decreases as T increases. This is due to the larger neighborhood resulting from larger T, which provides more opportunities for the local search to improve the cost of the solution.
- Our heuristic method tends to outperform MIP when the problem size becomes larger. More specifically, for $K \geq 3$, $N \geq 60$ and $T \geq 7$, the heuristic finds a better solution than the MIP. This is again because, as we increase N and T, the size of the neighborhood considered by our heuristic becomes larger, while the MIP becomes more difficult to solve. As a result, our heuristic is likely to find better solutions for larger sized instances.
- Our heuristic achieves a steady runtime of a few seconds across all test instance. The MIP generally terminates at its time limit of one hour and is thus omitted from Figure 3.7.

Therefore it is recommended to apply our heuristic method when the problem size becomes larger, because it is more likely to find a good solution due to the larger size of the neighborhood that can be searched, and

requires significantly less time to find a good solution than the MIP.

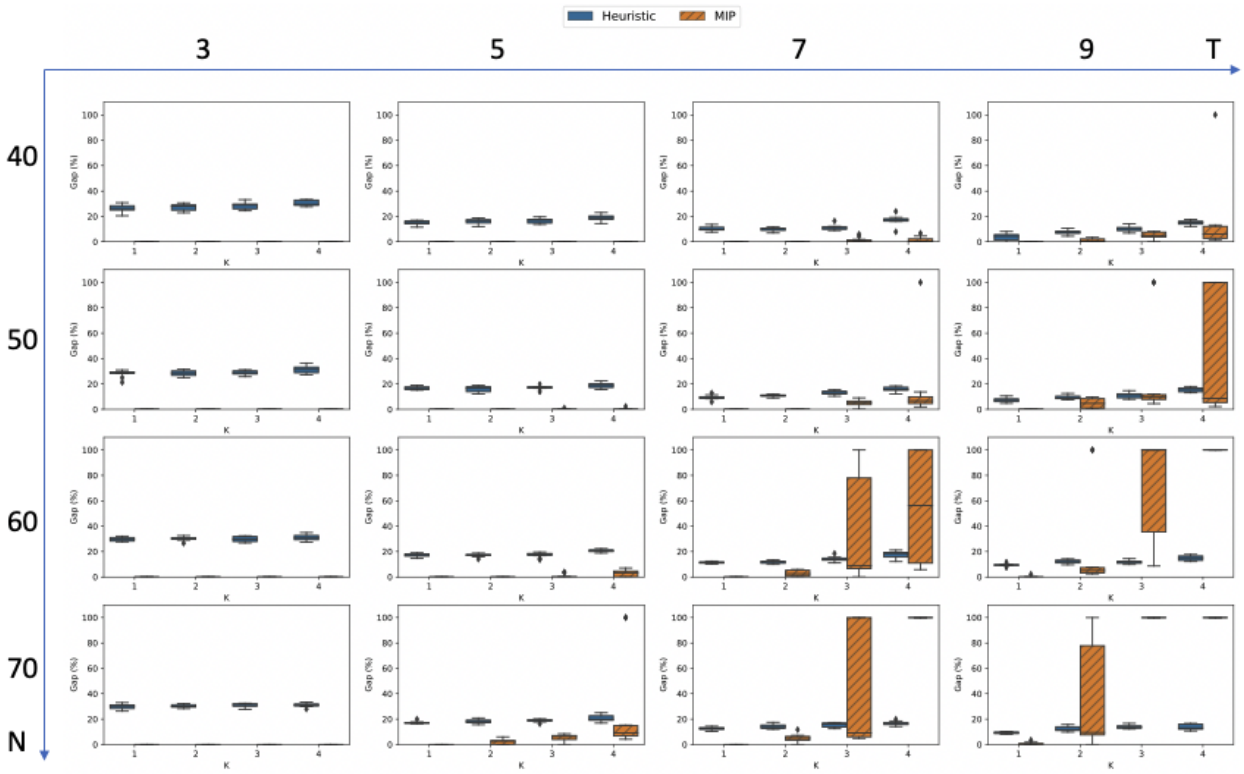


Figure 3.6: Comparison of methods for the capacitated case. A 4×4 grid of the gap comparison when varying N, T and K. Starting from the top-left corner, N increases from 40 to 70 at an interval of 10 along the vertical axis and T increases from 3 to 9 at an interval of 2 along the horizontal axis. K varies from 1 to 4 within each cell. Diagonal stripe pattern indicates MIP optimality gap.

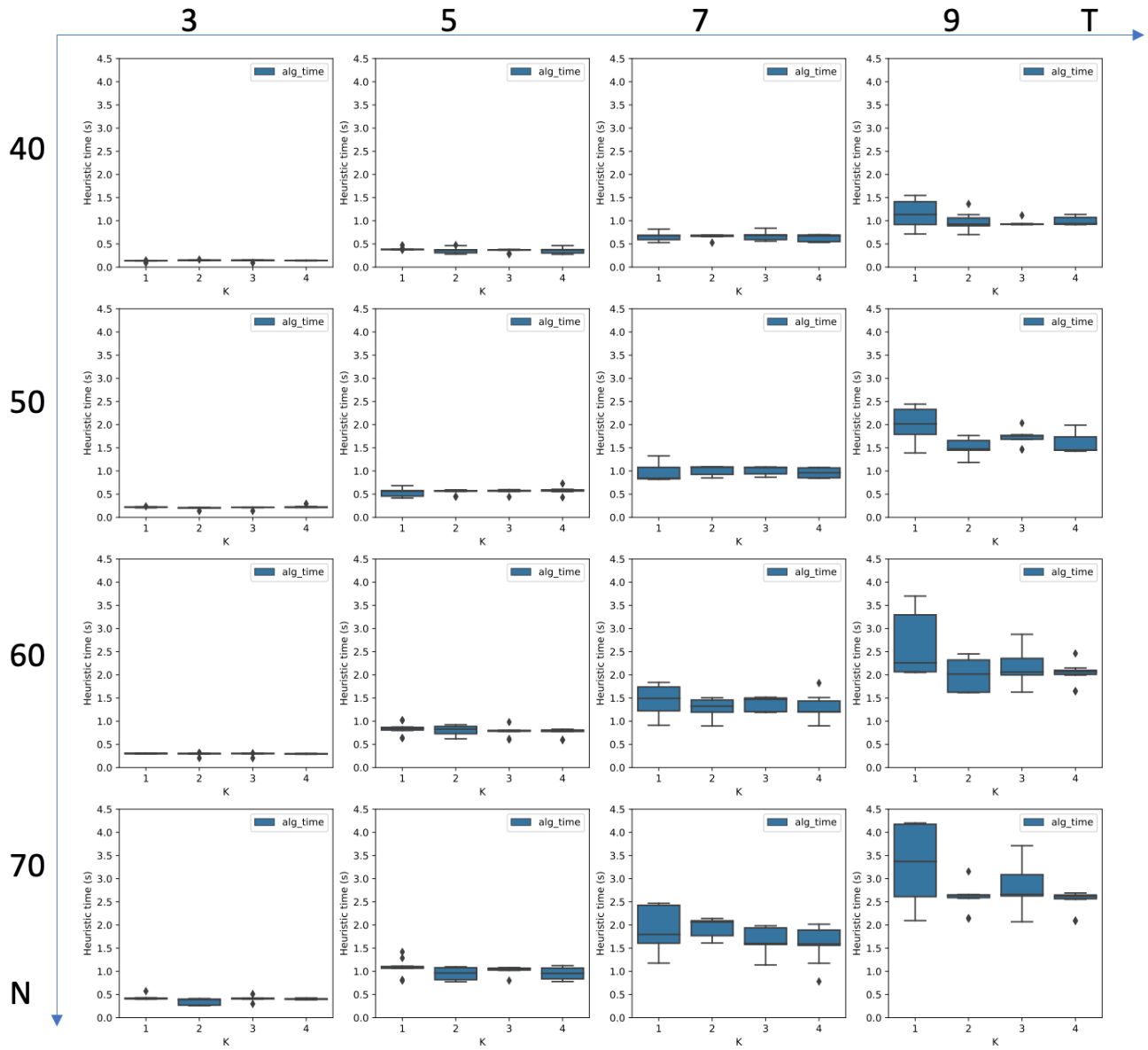


Figure 3.7: Runtime of the heuristic for the capacitated case. A 4×4 grid of the runtime of the heuristic when varying N and T. Starting from the top-left corner, N increases from 40 to 70 at an interval of 10 along the vertical axis and T increases from 3 to 9 at an interval of 2 along the horizontal axis. K varies from 1 to 4 within each cell.

3.6.4 True Optimality Gap when MIP is solved exactly

We report the Gap $(UB - LB)/UB$ of our heuristic for some data sets where an optimal solution can be found within reasonable time. Table 3.2 reports this gap for four small data sets. It shows that the optimality

gap generally decreases as we increase T.

N	T	Min Gap (%)	Median Gap (%)	Max Gap (%)
30	3	19.67	25.93	31.47
30	5	8.50	13.11	17.75
40	3	23.24	27.42	32.31
40	5	10.45	15.09	17.15

Table 3.2: Gap $(UB - LB)/UB$ of the capacitated Prioritized Local Search heuristic for capacitated instances solved to optimality. N is the number of clients and T is the number of days and the number of trucks is one.

3.7 Conclusion

In this chapter, we studied the deterministic inventory routing problem over a discrete finite time horizon. Given clients on a metric, each with daily demands that must be delivered from a depot and holding costs over the planning horizon, an optimal solution selects a set of daily tours through a subset of clients to deliver all demands before they are due and minimizes the total holding and tour routing costs over the horizon. For the capacitated case, a limited number of vehicles are available, where each vehicle makes at most one trip per day. Each trip from the depot is allowed to carry a limited amount of supply to deliver.

Motivated by an approximation algorithm proposed by Fukunaga et al. (2014) which relies on a reduction to a related problem called the prize-collecting Steiner tree (PCST) problem, we develop local search heuristics for uncapacitated IRPs. In particular, we define a very large neighborhood search step and reduce it to a PCST problem. Extending the local search ideas to the capacitated case, we refine the algorithms by penalizing any infeasibility until reaching a solution satisfying the capacities. Computational experiments show that proposed methods can find near optimal solutions for both cases and substantially reduce the computation time compared to a MIP-based approach.

Chapter 4

A Study on the Traveling Salesman Problem with Drone

4.1 Introduction

Vehicle routing problems have become increasingly important with the evolution of online shopping and fulfillment and a variety of delivery services. The use of unmanned aerial vehicles, or drones, for this purpose is actively explored by industry (Otto et al., 2018). A common model is to equip a delivery truck with one or more drones to deliver packages in parallel to the truck (UPS, 2017). Unlike the traditional setting where a fleet of vehicles have little operational constraints to each other, the drone operation is highly constrained to the truck operation because it needs to pick up packages for delivery from a truck. As a result the completion time also depends on the waiting time incurred due to the synchronization between the truck and the drone.

In this paper we study the design of optimal joint truck and drone routes under this scenario. We consider the elementary case where only one truck and one drone is available. Given a set of customers to be served either by a truck or a drone, our objective is to minimize the completion time of the entire delivery task, *i.e.* the total time it takes to serve all customers. For operational simplicity, we assume the drone can only be dispatched at a customer location and the service time at each location is instant. In a feasible solution, the truck route forms a tour which starts from and returns to the depot with a subset of customers served along the tour. Each remaining customer is served by the drone which is dispatched from a customer location and returns to a (possibly different) location on the truck tour. We follow Agatz et al. (2018) and call this problem the *traveling salesman problem with drone* (TSP-D).

Contributions. Our first contribution is a proof that the TSP-D is strongly NP-hard even in the case when a

truck route is given and we need to optimally integrate the remaining drone visits. Our second contribution is a new constraint programming (CP) formulation that relies on representing the TSP-D as a scheduling problem. We show experimentally that at the time of publication, our CP approach outperforms the best exact method from the literature, and is competitive with a state-of-the art heuristic method in terms of solution quality.

Finally in Section 4.6 we discuss our attempt of using logic-based Bender decomposition.

4.2 Related Work

In this section, we provide a literature review on TSP-D, which can be viewed as a subfield of a more general research area called *drone-assisted routing*. For other variants of drone-assisted routing problems, we refer the interested reader to two recent comprehensive surveys by Macrina et al. (2020) and Chung et al. (2020).

The hybrid truck and drone model was first studied by Murray and Chu (2015). In this work the authors present two new variants of the traditional TSP called the *flying sidekick traveling salesman problem* (FSTSP) and the *parallel drone scheduling TSP* (PDTSP), respectively. In FSTSP, the truck cooperates with the drone during the routing process while in PDTSP, different vehicles operate as independent work units. The authors present an integer programming model which is solved by Gurobi. Since it takes several hours to solve instances of 10 customers, they propose a heuristic which starts by finding a solution of the classic TSP, and then attempts to insert the drone and remove some customers from truck route by evaluating the achievable savings. Agatz et al. (2018) slightly relaxes the assumption in Murray and Chu (2015) to allow the truck to wait at the same location while the drone makes its delivery. They term this problem TSP-D. The authors propose an exponential-sized integer programming model. Faced with the similar scalability issues, they propose two route-first, cluster-second heuristics based on local search and dynamic programming and show that substantial savings are possible with this hybrid logistics model compared to truck-only delivery. Ponza (2016) proposes a simulated annealing heuristic for TSP-D and tested his method on a set of instances with up to 200 customers. Poikonen et al. (2019) propose a specialized branch-and-bound procedure, which includes boosted lower bound heuristics to further speed up the solving process. They analyze the trade-off between objective value and computation time, as well as the effect of drone battery duration and drone speed. Other (meta)heuristic algorithms for FSTSP or TSP-D have also been proposed in the literature, see e.g., Carlsson and Song (2018); de Freitas and Penna (2020); Yurek and Ozmutlu (2018).

Variants of TSP-D have also been studied in the literature. Macrina et al. (2020) further relax problem assumptions to allow the drone to be launched and connect to a truck either at a node or along a route arc. They present a greedy randomized adaptive search procedure (GRASP). Energy consumption of drones is considered in Dorling et al. (2017); Ferrandez et al. (2016). In addition to energy consumption, Jeon et al.

(2019) considers other practical limitations of drones such as 'no fly zones'. Ha et al. (2018) consider a TSP-D variant where the objective is to minimize operational costs including total transportation cost and the cost incurred by vehicles' waiting time. They present two heuristic algorithms for solving the min-cost TSP-D. Ha et al. (2020) extends their previous work (Ha et al., 2018) by considering two objective functions: the first one minimizes the total operational cost while the second one minimizes the completion time. The authors propose a hybrid genetic algorithms which combines genetic search and local search. Salama and Srinivas (2020) present mathematical programming models to jointly optimize customer clustering and routing and propose a machine learning warm-start procedure to accelerate the MILP solution.

Compared with the popularity of heuristic methods, only a limited number of research papers have been published on exact algorithms for TSP-D. Yurek and Ozmutlu (2018) develop a decomposition-based iterative algorithm that solves instances optimally with up to 12 customers. A three-phase dynamic programming approach is proposed by Bouman et al. (2018), which can optimally solve instances with up to 15 customers. Vásquez et al. (2021) proposes a Benders decomposition algorithm with additional valid inequalities and improved optimality cuts. They are able to solve instances with up to 24 customers. The current state-of-the-art is achieved by Roberti and Ruthmair (2019) via a branch-and-cut-and-price (BCP) algorithm with ng-route relaxation. The proposed algorithm can solve optimally instances with up to 40 customers within one hour of computation time.

The first theoretical study is by Wang et al. (2017), who consider the more general vehicle routing problem with multiple trucks and drones. They study the maximum savings that can be obtained from using drones compared to truck-only deliveries (*i.e.* TSP cost) and derive several tight theoretical bounds under different truck and drone configurations. Poikonen et al. (2017) extend Wang et al. (2017) to different cases by incorporating cost, limited battery life and different metrics respectively.

4.3 Problem Definition

The Traveling Salesman Problem with Drone (TSP-D) can be formally defined as follows. We are given a complete directed graph $G = (V, A)$. The vertex set $V = \{0, 0'\} \cup N$ where both 0 and $0'$ represent a single depot and N represents a set of customers. The arc set A is defined as $A = \{(0, j) : j \in N\} \cup \{(i, j) : i, j \in N, i \neq j\} \cup \{(j, 0'), j \in N\}$. Each customer demands one parcel. A single truck, equipped with a single drone is used to complete the overall delivery task. They start together from the depot, visit each customer by either vehicle and finally return to the depot. During the process, the drone may travel separately from the truck for parcel deliveries before reconnecting with the truck at some point, therefore potentially increasing efficiency via parallelization. The time for the truck and the drone to traverse arc $(i, j) \in A$ is denoted by t^T_{ij} and t^D_{ij} respectively. Furthermore, we make the following assumptions about the behavior of the truck and the drone.

1. The truck can dispatch and pick up a drone only at the depot or at a customer location. The truck may continue serving customers after a drone is dispatched and reconnect with the drone at a possibly different node;
2. Due to capacity and safety considerations, the drone can only deliver one parcel at a time;
3. The vehicle (truck or drone) which first arrives at the reconnection node has to wait for the other one;
4. Once the drone returns to the truck, the time required to prepare the drone for another launch is negligible.

Our objective is to minimize the completion time, i.e. from the time the truck is dispatched from the depot with the drone to the time when the truck and the drone return to the depot.

Notation. Let $n := |N|$ be the number of customers. A *truck node* is a node visited by the truck alone. Similarly, a *drone node* is a node visited by the drone alone. A *combined node* is a node visited by both the truck and the drone. In a feasible solution to TSP-D, denote V_d as the set of drone nodes, and $V_n := V \backslash V_d$ as the set of non-drone nodes (including the depot) visited by the truck either with or without the drone atop the truck. For $i \in V_d$, let $p(i)$ be the dispatch node where the drone is dispatched right before visiting i, $q(i)$ be the pick up node where the drone returns immediately after visiting i. Let E_t be the set of edges in the truck tour. For $i, j \in V_n$, let T_{ij} denote the path induced by E_t and $w(T_{ij}) = \sum_{a \in T_{ij}} t^T_{ij}$. Consider a partial drone schedule where the drone is dispatched from the truck at node j, visits node i and meets up with the truck at node k (we allow $j = k$). We call this partial drone schedule a *drone activity* and use a shorthand notation $j \to i \to k$ to represent this activity.

4.4 Theoretical Analysis

In this section, we perform theoretical analysis on TSP-D. In particular, we prove the NP-hardness of a restricted problem and develop an approximation algorithm for a special case.

4.4.1 Computational Complexity

Solving the TSP-D to proven optimality is computationally challenging. While the TSP-D is known to be NP-hard due to a reduction from TSP, we aim to provide more insight in the computational difficulty by considering a restricted version, which we call the drone routing problem (DRS). Below We prove strong NP-hardness result for this restricted version.

We associate the drone activity $j \to i \to k$ with a cost c_{ijk} defined as

$$c_{ijk} = \max\{0, t^D_{ji} + t^D_{ik} - w(T_{jk})\} \tag{4.1}$$

This is the marginal time a drone activity adds to the truck tour. Given V_n, V_d and E_t, we define a set of drone activities to be *feasible* if (1) each drone node in V_d appears in exactly one drone activity and (2) any pair of activities do not overlap in time.

We now define the drone routing problem as follows:

Definition 4.4.1 (Drone routing problem)**.** Given V_n, V_d, E_t. The drone routing problem is to find a feasible set of drone activities with minimum total drone activity cost.

We show that DRS is strongly NP-hard by a reduction from 3-partition.

Definition 4.4.2 (3-Partition)**.** Given positive integers m, B and $3m$ positive integers $x_1, \ldots, x_{3m}$ satisfying $\sum_{q=1}^{3m} x_q = mB$ and $\frac{B}{4} < x_q < \frac{B}{2}$ for $q = 1, \ldots, 3m$. Does there exist a partition of the set $Y = \{1, \ldots, 3m\}$ into m disjoint subsets $Y_1, \ldots, Y_m$ such that $\sum_{q \in Y_i} x_q = B$ for $i = 1, \ldots, m$?

Theorem 4.4.1. *The drone routing problem is strongly NP-hard.*

Proof. We prove the theorem when the costs are symmetric and truck and drone traverse at the same speed, i.e. $t^T_a = t^D_a, \forall a \in A$. We give a pseudo-polynomial time reduction from 3-partition. Given an instance of 3-partition as in Definition 4.4.2, we construct a graph with $m(B+1)$ non-drone nodes and $4m$ drone nodes. The truck route connects m paths P_i, each having $B+1$ nodes and B unit edges. Edges that connect two paths are assigned $\epsilon = \frac{1}{2m}$. Direct all edges in the cycle counterclockwise. The tail of a directed path P_i is defined as the tail of the first arc in P_i. We similarly define the head of P_i. The drone nodes are partitioned into two disjoint sets A and B. A has $3m$ nodes $v_1, \ldots, v_{3m}$. For $i = 1, \ldots, 3m$, v_i is connected to each node on the cycle via an edge of weight $\frac{x_i}{2}$. B contains m dummy nodes $u_1, \ldots, u_m$. For $i = 1, \ldots, m$, each u_i is connected to the head of P_i and tail of P_{i+1} via two edges of weight $\frac{\epsilon}{2}$ (we assume $P_{m+1} = P_1$). Other edges connected to u_i are assigned a unit weight so metric inequality holds. Below we show Lemma 4.4.2, from which the theorem follows. □

Lemma 4.4.2. *There exists a 3-partition if and only if there exists a feasible solution to the above DRS instance of zero total cost.*

Proof. 'if': connect each dummy node u_i to the head of P_i and tail of P_{i+1}. Without loss of generality assume the feasible partition is $\{x_{3k+1}, x_{3k+2}, x_{3k+3}\}$ for $k = 0, \ldots, m-1$. Then $v_{3k+1}, v_{3k+2}, v_{3k+3}$ are connected to path P_k in the following way: v_{3k+1} is connected to the first node and $(x_{3k+1}+1)$-th node on the path, v_{3k+2} is connected to $(x_{3k+1}+1)$-th and $(x_{3k+1}+x_{3k+2}+1)$-th nodes, v_{3k+3} is connected to $(x_{3k+1}+x_{3k+2}+1)$-th and x_{B+1}-th nodes. It is easy to check that the total cost is zero.

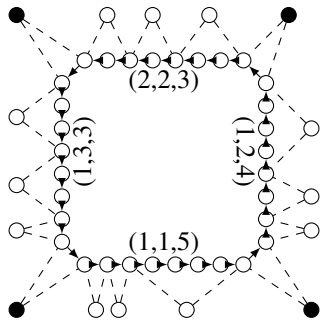


Figure 4.1: An example of the reduction, with $m = 4$, $B = 7$ and feasible partitions $(1, 1, 5)$, $(1, 2, 4)$, $(1, 3, 3)$, $(2, 2, 3)$. Drone activities are shown as dashed lines and dummy nodes are marked as solid. While each drone node has the same distance to any node on the truck cycle, we put the drone nodes outside the cycle for visualization purposes.

'only if': we claim each dummy node u_i in any solution with zero total cost must be connected to the head of P_i and tail of P_{i+1}: suppose not, note for any $t \neq i$, u_i cannot be connected to the head of P_t and tail of P_{t+1} since otherwise such a drone activity has non-zero cost. As a result any drone activity which visits u_i covers at least a unit-length edge on the cycle. Therefore after visiting u_i, remaining edges on the cycle have at most $mB - 1 + m\epsilon = mB - \frac{1}{2} < mB$ length to use for visiting the remaining drone nodes. Notice each visit of a node $v_l \in A$ must cover a path at least x_l to make the drone activity cost zero and each visit must cover non-overlapping path on the cycle, which is a contradiction to the fact that the remaining edge length on the cycle is less than mB. Therefore we've shown the claim. The result follows by reversing the steps in the 'if' part to partition drone nodes into m sets where each set contains 3 nodes that are visited by using edges in the same path. □

4.4.2 Approximation Algorithm for a Special Case

A common simplification in the literature of TSP-D is the following: assume the graph is undirected, and the truck is ρ times slower than the drone, i.e. for each edge e, $t_e^T = \rho t_e^D (\rho \geq 1)$. In the remaining of this section, we take the above assumptions. Below we present a constant-factor approximation algorithm for this special case. Our algorithm applies a simple reduction to the *ring-star network design* (RSND) problem defined below,

Definition 4.4.3 (Ring-Star Network Design Problem)**.** We are given an undirected graph $G = (V, E)$ with non-negative edge cost c_e for $e \in E$. The problem is to pick a tour (ring backbone) on a subset of the nodes and connect the remaining nodes to the tour such that the sum of the *tour cost* and the *access cost* is minimized. The cost of connecting a non-tour node i to a tour node j is c_{ij}. An edge used to connect non-tour node to a tour node is called an *access edge*. The access cost includes the cost of all access edges. On the other hand, the cost of including an edge e in the tour is ρc_e where $\rho \geq 1$ is a given non-negative

constant which reflects the ratio between the cost per unit length of a tour edge and that of an access edge.

Ravi and Salman (1999) obtained a constant-factor approximation algorithm for the RSND by LP rounding with filtering. The performance of their algorithm is summarized in the following lemma (Corollary 2 of Ravi and Salman (1999)):

Lemma 4.4.3. *There exists a polynomial-time* $(3 + 2\sqrt{2})$*-approximation algorithm for RSND.*

To reduce TSP-D to RSND, we construct an instance of TSP-D and an instance of RSND on the same graph $G = (V, E)$ with the same edge cost c_e and truck-drone speed ratio ρ. Let v_{tspd}, v_{rsnd} be respectively the optimal solutions to TSP-D and RSND. The following lemma relates the two quantities.

Lemma 4.4.4. $v_{tspd} \geq \frac{2}{3} v_{rsnd}$

Proof. Due to the synchronization requirement in section 4.3, the vehicle (truck or drone) that arrives at the pick-up node first has to wait for the other one. Therefore the total completion time of TSP-D is at least the travel time of the truck tour and that of the access connection traveled by the drone, i.e.,

$$v_{tspd} \geq \rho \sum_{e \in E_t} c_e \tag{4.2}$$

$$v_{tspd} \geq \sum_{i \in V_d} c_{p(i),i} + c_{i,q(i)} \geq 2 \sum_{i \in V_d} c_{i,N(i)} \tag{4.3}$$

where the second inequality (4.3) holds since $N(i)$ is i's nearest non-drone node. Therefore we have

$$\begin{aligned} 3v_{tspd} &\geq 2\rho \sum_{e \in E_t} c_e + 2 \sum_{i \in V_d} c_{i,N(i)} \\ &\geq 2v_{rsnd} \end{aligned}$$

where the first inequality follows by using the two lower bounds obtained above. □

Combining lemma 4.4.3 and 4.4.4 and noticing that a feasible solution to RSND is also feasible to TSP-D, we have

Theorem 4.4.5. *There exists a polynomial-time* $(\frac{9}{2} + 3\sqrt{2})$*-approximation algorithm for TSP-D.*

Finally we remark that inequality (4.2) and (4.3) form a non-trivial lower bound on the optimal value of TSP-D. We will use this fact in Section 4.6 to obtain a valid subproblem relaxation.

4.5 Constraint Programming Formulation

An essential feature of a truck-drone schedule is the synchronization between truck and drone operations. This poses a significant challenge to construct a MIP model with a tight linear relaxation. Below we explain how to construct a compact CP with $O(n^2)$ variables and constraints, using the constraint-based scheduling formalism introduced in IBM ILOG CPLEX (2017); Laborie (2009); Laborie et al. (2018). The CP solver IBM ILOG CP Optimizer Laborie et al. (2018) provides an expressive modeling language based on the notion of interval variables representing the execution of an activity. Its domain encodes the presence status (Boolean) (true if the activity is executed). When a is present, it is represented by variables $s(a)$ for its start time, $e(a)$ for its end time, and $d(a)$ for its duration, obeying the relationship $d(a) = e(a) - s(a)$. On the other hand, an absent interval variable is not considered by any constraint or expression on interval variables in which it is involved. An activity can be forced to present or declared 'optional', *i.e.* its presence status can be either true or false to be decided by the solver. Below we assume all interval variables are optional unless stated otherwise.

Recall the number of nodes including the starting and ending depot is $n + 2$. Denote both node 0 and $n + 1$ as the depot (leaving and returning). For each node i define three sets of interval variables: `dVisit[i]` and `tVisit[i]` and `visit[i]`. For each customer i, `tVisit[i]` represents the time interval between the truck arriving at i and leaving from i. Each `dVisit[i]` represents the time interval (if i is a drone customer) between the drone leaving to serve i and the drone rejoining the truck. The variable `visit[i]` represents `tVisit[i]` if i is not a drone customer and `dVisit[i]` otherwise. For each node i, we create two interval variables `dVisit_before[i]` and `dVisit_after[i]` which represent splitting `dVisit[i]` (if present) into the interval before i is served and after i is served. For each pair (i, j), we define two interval variables `tdVisit[i][j]` and `dtVisit[i][j]`, where the former represents the drone leaving from j to serve i and the latter represents the drone leaving from drone customer i to rejoin the truck at j. As an example, an activity $i \to j \to k$ is composed of `tdVisit[i][j]` and `dtVisit[i][k]`, which are constrained to be equivalent to `dVisit_before[i]` and `dVisit_after[i]` respectively. The complete model is presented in pseudocode in Figure 4.2.

In the CP model (Figure 4.2), the objective `endOf(visit[n+1])` is the end time of `visist[n+1]`, i.e. the time that both vehicles return to the depot. Constraints can be divided into three categories: (a) set basic properties (presence, duration, etc.) of interval variables (line 4-16) (b) set correct logical relationships or representations between interval variables (line 17-28) and (c) set precedence relations between interval variables (line 29-40). Below we provide a detailed description of these three categories.

Line 5 requires each customer to be visited either by truck, drone or together. Line 6-7 forbid the drone to deliver packages to the depot. Line 8-9 require that the tour starts from the depot and ends at the depot. Line 10-12 constrain the drone fly range to be shorter than R. Line 13-16 state that, if drone flies from j to i, then travel time is at least t^D_{ij}.

```
1   minimize
2     endOf(visit[n+1])
3   subject to {
4     // basic properties of variables
5     forall (i in 0...n+1) setPresent(visit[i])
6     setPresent(tVisit[0])
7     setPresent(tVisit[n+1])
8     first(tVisit, tVisit[0])
9     last(tVisit, tVisit[n+1])
10    forall(i in 0...n+1) {
11      dVisit[i].setUB(R)
12    }
13    forall(i, j in 0...n+1) {
14      tdVisit[i][j].setLB(t^D_ji)
15      dtVisit[i][j].setLB(t^D_ij)
16    }
17    // representations and logical constraints
18    forall(i in 0...n+1) {
19      alternative(visit[i], [tVisit[i], dVisit[i]])
20      alternative(dVisit_before[i], [all (j in 0...n+1) tdVisit[i][j]])
21      alternative(dVisit_after[i], [all (j in 0...n+1) dtVisit[i][j]])
22      span(dVisit[i], [dVisit_before[i], dVisit_after[i]])
23      presenceOf(dVisit[i]) == presenceOf(dVisit_before[i]) && presenceOf(dVisit_after[i])
24    }
25    forall(i, j in 0...n+1) {
26      ifThen(presence_of(tdVisit[i][j]), presence_of(tVisit[j]))
27      ifThen(presence_of(dtVisit[i][j]), presence_of(tVisit[j]))
28    }
29    // temporal constraints
30    noOverlap(tVisit, t^T)
31    noOverlap(dVisit)
32    forall(i in 0...n+1) {
33      endBeforeStart(dVisit_before[i], dVisit_after[i])
34    }
35    forall(i, j in 0...n+1) {
36      startBeforeStart(tVisit[j], tdVisit[i][j])
37      startBeforeEnd(tdVisit[i][j], tVisit[j])
38      endBeforeEnd(dtVisit[i][j], tVisit[j])
39      startBeforeEnd(tVisit[j], dtVisit[i][j])
40    }
```

Figure 4.2: A compact constraint programming formulation for TSP-D.

Constraints of the form `alternative(`a, A`)` where a is an interval variable and A is an array of interval variables, states that when a is present, its start time (end time, resp.) will be equal to the start time (end time, resp.) of one of the intervals in A. Therefore line 19 states that each customer i is either visited by the drone or not. Line 20 states that for each customer i, `dVisit_before[i]` (if present, i.e. if i is a drone customer) denotes the interval between the drone leaving j for some j and arriving at i. Similarly, line 21 states that for each customer i, `dVisit_after[i]` (if present) denotes the interval between the drone leaving i and arriving at j for some j. Constraints for the form `span(`a, A`)` states that, if a is present, the interval starts together with the first present interval in A and ends together with the last present interval in A. Therefore line 22 states that each `dVisit[i]` (if present) can be divided into `dVisit_before[i]` and `dVisit_after[i]`. Line 23 states that, for each customer i, the presence of `dVisit[i]` is equivalent

to the presence of `dVisit_before[i]` and `dVisit_end[i]`. Similar logical constraints are imposed from line 25-28.

Constraints of the form `noOverlap(tVisit,` t^T`)` state that whenever both `tVisit[i]` and `tVisit[j]` are present, `tVisit[i]` is constrained to end before the start of `tVisit[j]` by at least t^T_{ij} amount of time. Similarly `noOverlap(dVisit)` states that intervals denoting drone deliveries cannot overlap. Constraints for the form `endBeforeStart(`v_1, v_2`)` imposes a temporal constraint between interval v_1 and v_2: it requires that v_1 cannot end before the start of v_2. Other temporal constraints such as `startBefore, startBeforeEnd, endBeforeEnd, startBeforeEnd` are all similarly defined. It can be verified that line 35-40 implements the synchronization constraint between the truck and the drone.

Computational Experiments

We implemented and solved our CP model with CP Optimizer version 12.8.0, using the Python interface DOcplex. Our experiments are run on a 2.2GHz Intel Core i7 quad-core machine with 16GB RAM. We compared our approach to the two best approaches from the literature: the exact dynamic programming (DP) algorithm in Bouman et al. (2018), and the branch-and-bound algorithm from Poikonen et al. (2019). The implementation of the latter relies on the assumption that the drone has a finite range, for which the method is not guaranteed to provide optimal solutions.

Runtime w.r.t. the number of locations. We first present the results on the exact comparison. Since benchmark instances used in Bouman et al. (2018) are not publicly available, we follow their approach to generate 10 uniform instances of each size and set the truck-drone speed ratio two. Table 4.1 reports the average runtime (in seconds) of our approach (CP) and the reported runtime from Bouman et al. (2018) (DP). While DP can solve the smaller problems faster than CP, our approach scales more gracefully.

Size	10	11	12	13	14	15	16	17	18
CP	6.79	5.71	16.66	15.66	50.83	120.59	216.46	375.49	564.22
DP	1.00	4.00	12.00	56.00	306.00	1568.00	9508.00	–	–

Table 4.1: Runtime comparision (s) of CP and DP (exact)

Runtime w.r.t the truck-drone speed ratio. Computational experience from the literature suggests that the problem becomes easier when the truck-drone speed ratio is large. To understand how sensitive our CP approach is w.r.t. different ratios, we report in Figure 4.3 the average runtime of instances solved above with the number of locations equal to 15. It confirms the finding from the literature that the problem is easier to solve when the ratio becomes larger.

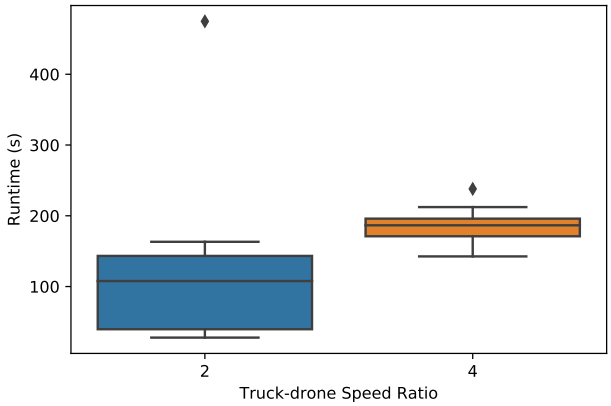


Figure 4.3: Runtime comparision (s) of CP varying the truck-drone speed ratio with 15 locations

Incumbent value for large instances. Finally we investigate the quality of incumbents found by CP Optimizer for large instances. Table 4.5 presents the comparison with the branch-and-bound method (BAB) of Poikonen et al. (2019) in terms of solution quality. For this experiment, we apply a time limit of 10 minutes for each instance. As a benchmark, we use the same dataset as Poikonen et al. (2019) (25 instances of each size). The speed ratio is set to two and the drone range is limited to 20 as in Poikonen et al. (2019). The table reports the mean objective value for each problem size. The results for BAB are the best solutions found among all branch-and-bound heuristic variants. We note that the runtime of the BAB approach is typically less than one minute. In comparison, we terminate our CP model at a time limit of 10 minutes. These results show that the time-limited CP approach can produce competitive solutions for instances of up to 60 locations but that the dedicated heuristic branch-and-bound outperforms CP on larger instances.

Size	10	20	30	40	50	60	70	80	90	100	200
CP	116.60	136.64	160.12	198.88	237.4	276.96	316.20	407.36	515.80	679.64	–
BAB	149.53	171.64	200.95	226.15	241.36	267.54	283.30	299.09	322.37	337.91	465.63

Table 4.2: Solution value comparison of CP and BAB (heuristic)

4.6 Logic-based Benders Decomposition

Inspired by the theoretical analysis in Section 4.4, we design a logic-based Benders decomposition (LBBD) algorithm. Computational experiments and its limitations are discussed in Section 4.6.4.

On a high level, LBBD divides a complex decision-making process into hierarchical levels. For simplicity, we assume a two-level decomposition below. The upper and lower level form a feedback loop, i.e. the upper level tries to fix a subset of decisions, the lower level solves the resulting subproblem and returns to the upper level useful information to improve the current decision. The process continues until we reach optimality or

prove infeasibility. For a formal presentation of LBBD, we refer the interested reader to Hooker (2019b). We remark that a traditional Benders decomposition approach is proposed by Vásquez et al. (2021) and can solve instances of about 20 customers.

The routing process of TSP-D naturally suggests the following decomposition: first we determine the partition of customers into drone and non-drone nodes, and then we construct truck and drone routes. Formally, we can decompose the problem into a partition master problem (PMP) and a truck-drone scheduling subproblem (TDSS). The PMP is concerned with finding a partition for all customer nodes into truck and drone nodes and the TDSS aims to find the optimal schedule for the truck and the drone to coordinate with each other and visit each customer. We model our master problem as an integer program (IP) and our subproblem as a constraint program (CP).

4.6.1 The Partition Master Problem

The master problem partitions each customer into either the truck node or the drone node. In order to find a high-quality partition, it is important, as suggested in Hooker (2007) to include a relaxation of the subproblem within the master problem. Our subproblem relaxation is inspired by the lower bound used in the design of our approximation algorithm in section 4.4.2. Recall V_n, V_d are the set of non-drone and drone nodes respectively. For each $i \in V_d$, $p(i)$ is the truck node from which the drone is dispatched right before visiting i, $q(i)$ is the truck node to which the drone returns immediately after visiting i and $N(i)$ is nearest truck node to drone node i. E_t is the set of edges in the truck tour. Let v be the optimal solution to SVRP-D. The proof of lemma 4.4.4 also shows that $v \geq \max\{\rho \sum_{e \in E_t} c_e, 2 \sum_{i \in V_t} c_{i,N(i)}\}$. We adapt an IP model from Ravi and Salman (1999) to represent this lower bound. Define

$$x_i = \begin{cases} 1 & \text{if i is a truck node} \\ 0 & \text{otherwise} \end{cases}$$

$$z_{ij} = \begin{cases} 1 & \text{if non-tour node i is assigned to tour node j} \\ 0 & \text{otherwise} \end{cases}$$

$$y_e = \begin{cases} 1 & \text{if edge e is chosen in the truck tour} \\ 0 & \text{otherwise} \end{cases}$$

Note in the subproblem relaxation any drone node i is assigned to its nearest truck node $N(i)$. These decisions are captured by the set of z variables defined above. We allow self-assignment, i.e., $z_{ii} = 1$ when i is a truck node. Recall 0 is the depot and N is the set of customer nodes. For any set $S \subset N$, $\delta(S)$ denotes the cut around set S. We use $y(\delta(S))$ as a shorthand expression for $\sum_{e \in \delta(S)} y_e$. The adapted IP

model (SPR) is formulated as follows:

$$\begin{aligned}
\min \quad & v && \text{(SPR)}\\
\text{s.t.} \quad & \sum_j z_{ij} = 1, \quad \forall i \in V && (4.4)\\
& z_{ij} \le x_j, \quad \forall j \in V && (4.5)\\
& y_{ij} \le x_i, y_{ij} \le x_j, \quad \forall i, j \in V && (4.6)\\
& \sum_{j \notin S} z_{ij} + \frac{1}{2} y(\delta(S)) \ge 1, \quad \forall i \in V, S \subset N && (4.7)\\
& v \ge \rho \sum_{e \in E} c_e y_e && (4.8)\\
& v \ge \sum_{i,j \in V} 2 c_{ij} z_{ij} && (4.9)\\
& x_i, y_{ij}, z_{ij} \in \{0, 1\}, \quad \forall i, j \in N_0 && (4.10)
\end{aligned}$$

Constraint (4.4) requires each node to be assigned to exactly one node (self-assignment is allowed when the node itself is a truck node). Constraint (4.5) prevents invalid assignment to a drone node. Constraint (4.7) is intended to capture the requirement of crossing certain cuts in the graph by edges in the subgraph that connect the truck nodes. Consider a set of nodes S not including the depot r and a particular node i, either i is assigned to some truck node outside S or S must contain at least one truck node. In the latter case, there must be at least two edges crossing $\delta(S)$ due to the tour requirement on truck nodes. This disjunction is expressed by constraint (4.7). Due to constraint (4.8) and (4.9), the objective function v is lower bounded by $\rho \sum_{e \in E} c_e$ and $2 \sum_{i \in V_d} c_{i,N(i)}$, corresponding to the truck tour cost and drone connection cost respectively.

Although (SPR) has exponential number of constraints, efficient row generation can be implemented via a separation oracle based on a minimum cut procedure Ravi and Salman (1999). To separate a given solution (x, y, z) over constraint (4.7) for a particular node i, we set up a capacitated undirected graph as follows: For every edge (i, j) of the complete graph on the entire node set V, we assign an edge-capacity of $y_{ij}/2$. We add a new node u_i and assign the capacity of the undirected edge between u_i and a node $j \in V$ to be z_{ij}. A polynomial-time procedure to determine the minimum cut separating r and p_i can now be used to test violation of all constraints of type (4.7) for node i. Repeating this for every node provides a polynomial-time separation oracle for constraints (4.7).

4.6.2 The Truck-Drone Scheduling Subproblem

Given the partition of truck and drone nodes, the goal of the TDSS is to optimally schedule and coordinate truck and drone routes so that each customer is visited. To solve the TDSS, wo formulate it as a scheduling problem and use it by the CP Optimizer (Laborie et al., 2018). Although the CP model detailed in Section 4.5 models the original TSP-D (instead of the TDSS), it can be easily adapted to model the subproblem in the following way. Recall V_n, V_d are the set of non-drone and drone nodes respectively. The change to make to the CP model in section 4.5 is: for all $k \in V_t$, force `tVisit[k]` to be present and for all $i \in V_d$, force `dVisit[i]` to be present.

4.6.3 Benders Cuts

Recall n is the number of customers and v is the objective function in the PMP. At iteration h, let V_t^h, V_d^h be the set of truck and drone nodes, x^h be indicator variables for truck nodes. Let v_{tdss}^h be the optimal solution to the TDSS given partition x^h. We are only able to generate naive benders cuts of the following form

$$v \geq v_{tdss}^h(\sum_{i \in V_t} x_i^h + \sum_{i \in V_d}(1 - x_i^h) - n)$$

Recall $|V_t \cup V_d| = n + 1$, therefore such a benders cut attains the value of v_{tdss}^h when $x = x^h$ and has negative values otherwise. Based on the result shown by Hooker and Ottosson (2003), the addition of the above benders cuts guarantees optimality as well as the finite convergence of the iterative process. We discuss the challenge of generating stronger benders cuts in section 4.6.4.

4.6.4 Preliminary Results and Limitations

Preliminary tests on benchmark instances from Poikonen et al. (2019) show that the LBBD is able to solve instances of $n = 10$ customers within 100 seconds, with a median of 40 seconds. However it fails to converge when $n \geq 15$. This is due to the following two reasons:

- The LBBD algorithm requires the subproblem to be solved exactly. In our decomposition, the subproblem is a non-trivial scheduling problem that is unscalable as the problem size increases;
- We are only able to generate nogood Benders optimality cuts which are ineffective to close the optimality gap within a reasonable amount of time.

To overcome these issues, one should modify the decomposition in such a way that subproblems can be completely decoupled from each other and solving each subproblem to optimality becomes easier. Moreover, stronger Benders cuts can be derived. Unfortunately, our alternative attempts along this direction so

far have been unsuccessful.

4.7 Conclusion

In this chapter, we studied a new routing problem called the traveling salesman problem with drone (TSP-D) that involves the collaboration between traditional trucks and modern drones. The drone can pick up packages from the truck and deliver them by air while the truck is serving other customers. The operational challenge combines the allocation of customers to either the truck or the drone, and the coordinated routing of the truck and the drone. In this work, we consider the scenario of a single truck and one drone, with the objective to minimize the completion time (or makespan). Since TSP-D generalizes the well-known traveling salesman problem (TSP), it is theoretically hard to solve to optimality. However this theoretical result is not sufficient to explain the computational findings in the literature that TSP-D is significantly harder to solve than TSP. To shed light on this question: we proved that this problem is strongly NP-hard, even in the restricted case when drone deliveries need to be optimally integrated in a given truck route. We then presented a constraint programming formulation that compactly represents the operational constraints between the truck and the drone. Our computational experiments showed that solving the CP model to optimality is significantly faster than the state-of-the-art exact algorithm at the time of publication. For larger instances up to 60 customers, our CP-based heuristic algorithm is competitive with a state-of-the-art heuristic method in terms of the solution quality. Finally, we described our attempts of a decomposition approach based on the idea of logic-based Benders decomposition. We discussed its limitations and possible alternatives.

Chapter 5

Truck-Drone Routing with Decision Diagrams

5.1 Introduction

The problem of supplying customers using vehicles based at a central depot is generally known as a *vehicle routing problem* (VRP). VRPs have become increasingly important with the evolution of online shopping and fulfillment and a variety of delivery services. Alongide the growing commercial value of VRPs, tremendous amount of research has been carried out on formulating and solving such problems. In this work, we focus on exact algorithms for solving VRPs. Currently, the most effective exact algorithms are *branch-and-cut-and-price* (BCP) algorithms. In general, a BCP algorithm is based on a mathematical programming formulation with a huge number of variables. It starts with a small number of variables and iteratively generates new promising variables. As variables can also be viewed as columns in the model, this iterative process is referred to as *column generation* (CG). The problem of finding new promising variables is referred to as the *pricing* problem. Apart from CG, additional constraints may need to be identified and added as well, which is referred to as *cutting*, and hence the name branch-and-cut-and-price.

In the context of VRPs, a BCP algorithm is typically based on a *set partitioning* (SP) formulation, where an element represents a customer and a set represents a route. An element is contained in a set if and only if the corresponding customer is visited in the corresponding route. Assuming the goal is to minimize a certain objective function such as total travel time, an important building block of a BCP algorithm is to obtain a tight lower bound on the optimal value. Such a bound is usually computed by solving the linear relaxation of the SP, which is referred to as the *master problem*. The quality of this lower bound crucially depends on the set of routes under consideration. At a first glance, one may only consider feasible routes and exclude infeasible ones when solving the master problem, which indeed provides a tight lower

bound. However, this approach typically leads to a pricing problem as complicated as the original problem itself. As a compromise, the set of routes under consideration is relaxed to also include infeasible ones in such a way that it reduces the complexity of solving pricing problems without the quality of lower bounds worsening excessively. Extensive research effort has been invested in designing effective route relaxations. Generally speaking, researchers in the literature have focused on route relaxations that allow infeasible routes excluding certain structures such as k-cycles. The resulting pricing problem is typically solved by *dynamic programming* (DP).

In this chapter, we propose novel route relaxations based on *decision diagrams* (DDs). Our route relaxations are motivated by close connections between DDs and DP models used for solving pricing problems. We adapt techniques from the DD literature to tighten our route relaxations which in turn improves the lower bounds. Furthermore, we propose two general approaches for deriving lower bounds from route relaxations without solving the master problem by CG. Our approaches are based on a flow model with side constraints and Lagrangian relaxation, respectively. Interestingly, these two approaches are shown to be equivalent to CG in the sense that they all compute the same lower bound under the same route relaxation. Based on these, we propose iterative frameworks which dynamically adjust route relaxations in order to compute strong lower bounds. We numerically evaluate the performance of our iterative frameworks on a new and challenging VRP variant called the *Traveling Salesman Problem with Drone* (TSP-D). Computational experiments show that our approaches are able to generate lower bounds whose values are competitive to those from the state-of-the-art BCP algorithm. Applied to larger problem instances where the BCP algorithm fails to output a lower bound within an hour, our approaches are shown to outperform CP or other MIP-based lower bounds.

The rest of this chapter is organized as follows. In Section 5.2 we review route relaxations used by BCP algorithms and relevant background on DDs. In Section 5.3 we provide a formal definition of the TSP-D. Section 5.4 provides preliminary information that connects DDs with DP models, as well as the set partitioning formulation which forms the basis of our approaches. Section 5.5 describes our DD-based route relaxations as well as techniques to refine those relaxations. Section 5.6 describes alternative approaches for solving the master problem and Section 5.7 describes our iterative frameworks in detail. Key implementation details are discussed in Section 5.8. Experimental evaluations are reported in Section 5.9.

5.2 Related Work

A BCP algorithm for a VRP uses CG to solve the master problem defined w.r.t a route relaxation. There is typically a trade-off between the efficiency for solving the pricing problem and the quality of the route relaxation: the higher the quality, the harder it is to solve the corresponding pricing problem. Extensive research efforts have contributed to striking a balance between those two aspects. Below we review route relaxations used by state-of-the-art BCP algorithms. We refer the interested reader to Chapter 3 of Toth and

Vigo (2014) and Costa et al. (2019) for other important components in BCP algorithms, such as cutting, branching, etc.. For the literature review on TSP-D, we refer the reader to Section 4.2.

Pricing and Route Relaxation. For most VRPs, the pricing problem can be modeled as the *elementary shortest path problem with resource constraints* (ESPPRC). Resources are quantities (e.g., time, load) used to assess the feasibility or cost of a route. Dror (1994) showed that this problem is NP-hard in the strong sense. For this reason, some authors have relaxed the ESPPRC to the *shortest path problem with resource constraints* (SPPRC), where repeated visits to the same customer (i.e. cycles) are allowed. The SPPRC is easier to solve than the ESPPRC as shown by Desrochers et al. (1992) who devised a pseudo-polynomial-time algorithm. However, completely relaxing the elementarity of routes may significantly reduce the lower bound quality obtained from the set partitioning formulation. As a result, several techniques have been devised to seek a better compromise between bound quality and computational efficiency.

- *k-Cycle Elimination.* Christofides et al. (1981) introduced this technique that has been largely employed to avoid cycles. It consists of forbidding cycles of length k or less while solving SPPRC. The use of 2-cycle elimination has been largely applied in the literature, as well as state-of-the-art techniques as it yields stronger bounds without changing the complexity of the labeling algorithm used for solving SPPRC (see e.g., Christofides et al. (1981)). To the best of our knowledge, k-cycle elimination with $k \geq 3$ has only been tested by Irnich and Villeneuve (2006) and Fukasawa et al. (2006).

- *Partial Elementarity.* Desaulniers et al. (2008) introduced another route relaxation called partially ESPPRC. It requires elementarity only for a subset of customers, whose maximal cardinality is determined a priori. This set is built dynamically from scratch by including customers that are visited more than once in a route of the optimal solution to the current set partitioning formulation. Note that if the maximal cardinality is less than the number of customers, there is no guarantee that elementary routes will be obtained at the end of the algorithm.

- *ng-Route Relaxation.* Currently, state-of-the-art BCP algorithms use the ng-route relaxation in their pricing procedures proposed by Baldacci et al. (2011). The ng-route concept relies on the definition of a neighborhood N_i for each customer i, which is usually defined as k customers nearest to i, for some parameter k chosen a priori. An ng-route is not necessarily elementary: it can contain a cycle starting and ending at a customer i if and only if there exists another customer j such that $j \notin N_i$. An important parameter in this relaxation is k, the cardinality of the neighborhood. On the one hand, the larger the value of k, the closer to the ESPPRC this relaxation becomes. On the other hand, the algorithm complexity increases exponentially with the value of k.

In contrast with the above route relaxations which forbid repeated visits based on the past visiting history, we propose novel relaxations based on DDs. Below we provide relevant background on DDs.

Decision Diagram. Decision diagrams are compact representations of Boolean functions, originally introduced for applications in circuit design by Lee (1959) and widely studied and applied in computer science. They have been recently used to represent the feasible set of discrete optimization problems, as demonstrated in Becker et al. (2005) and Bergman et al. (2011, 2012). This is done by perceiving constraints of a problem as a Boolean function representing whether a solution is feasible. Nonetheless, such DDs can grow exponentially large which makes any practical computation prohibitive in general.

To circumvent this issue, Andersen et al. (2007) introduced the concept of a *relaxed DD*, which is a diagram of limited size that represents instead an over-approximation of the feasible solution set of a problem. Relaxed DDs have shown to be particularly useful as a discrete relaxation of the feasible set of optimization problems. In particular, they can be embedded within a complete search procedure such as branch-and-bound for integer programming (Tjandraatmadja and van Hoeve, 2019, 2020), backtracking search for constraint programming (Cire and Van Hoeve, 2013; Kinable et al., 2017), or a stand-alone exact solver for combinatorial optimization problems (Bergman et al., 2014a, 2016b; Castro et al., 2020). On the other hand, the concept of a *restricted* DD was introduced by Bergman et al. (2014b) as a heuristic method for optimization problems. A restricted DD represents an under-approximation of the set of feasible solutions. O'Neil and Hoffman (2019) used restricted DDs as a primal heuristic for solving the traveling salesman problem with pickup and delivery.

For the TSP-D, we make use of relaxed DDs to generate novel route relaxations. Unlike BCP algorithms which solve the master problem via CG, we rely on a flow model with side constraints and Lagrangian relaxation to compute lower bounds. Our constrained flow model is similar to the one in van Hoeve (2020), which is used to computer lower bounds and refine relaxed DDs for graph coloring problems. Bergman et al. (2015) first applied Lagrangian relaxation to DDs to obtain improved bounds and showed its effectiveness on the traveling salesman problem with time window. This approach was used by Hooker (2019a) and Castro et al. (2020) to obtain improved bounds for specific applications. In terms of general integer programming models, Tjandraatmadja and van Hoeve (2020) explored a substructure amenable to DD compilations and obtained improved bounds by Lagrangian relaxation and constraint propagation.

We enhance our route relaxations by iteratively refining relaxed DDs. Our refinement procedures are directly inspired by constraint separation in van Hoeve (2020).

5.3 Problem Definition

The Traveling Salesman Problem with Drone (TSP-D) can be formally defined as follows. We are given a complete directed graph $G = (V, A)$. The vertex set $V = \{0, 0'\} \cup N$ where both 0 and $0'$ represent a single depot and N represents the set of customers. For technical reasons, we differentiate between 0 and $0'$: we call 0 the *starting depot* and $0'$ the *ending depot*. The arc set A is defined as $A = \{(0, j) : j \in$

$N\} \cup \{(i,j) : i,j \in N, i \neq j\} \cup \{(j,0'), j \in N\}$. Each customer demands one parcel. A single truck, equipped with a single drone is used to complete the overall delivery task. They start together from the depot, visit each customer by either vehicle and finally return to the depot. During the process, the drone may travel separately from the truck for parcel deliveries before reconnecting with the truck at some point, therefore potentially increasing efficiency via parallelization. The time for the truck and the drone to traverse arc $(i,j) \in A$ is denoted by t_{ij}^T and t_{ij}^D respectively. In general, the time to traverse an arc with the drone is not greater than the time to traverse the same arc with the truck, i.e. $t_{ij}^D \leq t_{ij}^T, \forall (i,j) \in A$. However we do not need this assumption for our solution methods. Furthermore, we make the following assumptions about the behavior of the truck and the drone.

1. The truck can dispatch and pick up a drone only at the depot or at a customer location. The truck may continue serving customers after a drone is dispatched and reconnect with the drone at a possibly different customer;

2. Due to capacity and safety considerations, the drone can only deliver one parcel at a time;

3. The vehicle (truck or drone) which first arrives at the reconnection customer has to wait for the other one;

4. Once the drone returns to the truck, the time required to prepare the drone for another launch is negligible.

Our objective is to minimize the completion time, i.e. from the time the truck is dispatched from the depot with the drone to the time when the truck and the drone return to the depot.

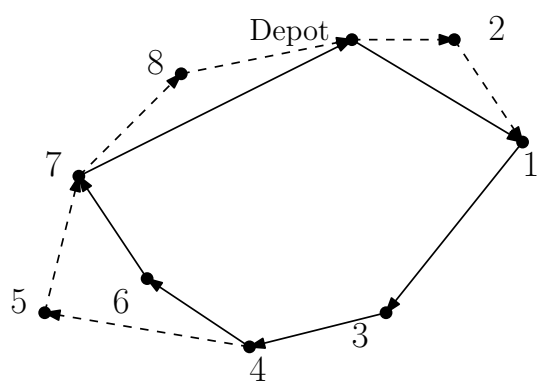


Figure 5.1: A feasible solution to a TSP-D instance with eight customers

Example 1. Figure 5.1 illustrates a feasible solution to a TSP-D instance with eight customers and the depot, i.e. customers 0 and $0'$. The truck leaves the depot to visit customer 1 while the drone is dispatched to visit customer 2. At customer 1 the truck and the drone have to be synchronized, and depending on the travel time, either the truck waits for the drone (when $t_{01}^T > t_{02}^D + t_{21}^D$), or the drone waits for the truck (when $t_{01}^T < t_{02}^D + t_{21}^D$). After they rejoin, the truck and the drone travel together to visit customer 3 and 4

consecutively. At customer 4, the drone is dispatched to visit customer 5 while the truck visits customer 6 and 7 consecutively. They rejoin at customer 7. The drone is dispatched to visit customer 8 before returning to the depot while the truck returns directly to the depot. The total completion time of this solution is computed as:

$$t = \max\{t^T_{01}, t^D_{02} + t^D_{21}\} + t^T_{13} + t^T_{34} + \max\{t^T_{46} + t^T_{67}, t^D_{45} + t^D_{57}\} + \max\{t^T_{70'}, t^D_{78} + t^D_{80'}\} \tag{5.1}$$

In the remainder of this chapter, we adopt the same notations used by Roberti and Ruthmair (2019). Let $n := |N|$ be the number of customers. A *truck customer* is a customer visited by the truck alone. Similarly, a *drone customer* is a customer visited by the drone alone. A *combined customer* is a customer visited by both the truck and the drone. As an example, in Figure 5.1, customer 6 is a truck customer, customers $2, 5, 8$ are drone customers and the rest are combined customers. A *truck arc* (*drone arc*, respectively) is an arc traversed by the truck (drone, respectively) alone. A *combined arc* is an arc traversed by the truck and the drone together. The solution in Figure 5.1 consists of four truck arcs $((0, 1), (4, 6), (6, 7), (7, 0'))$, six drone arcs $((0, 2), (2, 1), (4, 5), (5, 7), (7, 8), (8, 0'))$ and two combined arcs $((1, 3), (3, 4))$. A *truck leg* is a concatenation of truck arcs traversed by the truck alone in between two consecutive combined customers. A *drone leg* is a sequence of exactly two consecutive drone arcs traversed by the drone alone in between two consecutive combined customers. A *combined leg* is a concatenation of combined arcs traversed by the truck and the drone together that consists of combined customers only.

Example 2. The solution in Figure 5.1 consists of three truck legs $(0 \to 1, 4 \to 6 \to 7, 7 \to 0')$, three drone legs $(0 \dashrightarrow 2 \dashrightarrow 1, 4 \dashrightarrow 5 \dashrightarrow 7, 7 \dashrightarrow 8 \dashrightarrow 0')$ and one combined leg $(1 \to 3 \to 4)$. An *operation* is a synchronized pair of a truck leg and a drone leg in between the same pair of combined customers. Figure 5.1 consists of three operations, which we represent as $[0 \to 1, 0 \dashrightarrow 2 \dashrightarrow 1]$, $[4 \to 6 \to 7, 4 \dashrightarrow 5 \dashrightarrow 7]$ and $[7 \to 0', 7 \dashrightarrow 8 \dashrightarrow 0']$ respectively. A TSP-D solution can be seen as a concatenation of operations and combined legs. The solution in Figure 5.1 can be represented as $([0 \to 1, 0 \dashrightarrow 2 \dashrightarrow 1], 1 \to 3 \to 4, [4 \to 6 \to 7, 4 \dashrightarrow 5 \dashrightarrow 7], [7 \to 0', 7 \dashrightarrow 8 \dashrightarrow 0'])$.

We remark that a TSP-D solution can consist of operations only or a single combined leg. Based on the above definitions, a route can be formally defined as follows:

Definition 5.3.1 (Route)**.** A route is an ordered sequence of operations and combined legs that start from the depot and end at the depot such that the final customer of each operation or combined leg coincides with the initial customer of the subsequent operation or combined leg.

A route is *feasible* for the TSP-D if it visits each customer exactly once.

5.4 Preliminaries

In this section, we describe a dynamic programming (DP) model for the TSP-D and give a formal definition of decision diagrams. These two concepts form the basis of our algorithms. We then connect them by showing how to compile a DD based on a DP model.

5.4.1 Dynamic Programming Model

We slightly modify the DP model used by Roberti and Ruthmair (2019) to cater to our framework. Although we will not use this exact DP to solve the TSP-D, elements in this DP model (state definition, state transition function, etc.) are used throughout this chapter.

Recall from Section 5.3 that a solution to the TSP-D can be broken down into a concatenation of operations and combined legs. By definition, each operation consists of a truck leg and a drone leg; each truck leg is a sequence of truck arcs, and each drone leg is a sequence of exactly two drone arcs. Each combined leg is a sequence of combined arcs. For example, the solution illustrated in Figure 5.1 can be represented as $([0 \to 1, 0 \dashrightarrow 2 \dashrightarrow 1], 1 \to 3 \to 4, [4 \to 6 \to 7, 4 \dashrightarrow 5 \dashrightarrow 7], [7 \to 0', 7 \dashrightarrow 8 \dashrightarrow 0'])$. This can be viewed as sequentially generating operation $[0 \to 1, 0 \dashrightarrow 2 \dashrightarrow 1]$, combined leg $1 \to 3 \to 4$, operation $[4 \to 6 \to 7, 4 \dashrightarrow 5 \dashrightarrow 7]$ and finally operation $[7 \to 0', 7 \dashrightarrow 8 \dashrightarrow 0']$. Each operation can be generated by first generating the truck leg (one truck arc at a time), and then adding the drone leg. The DP model we describe in this section is based on the idea that each TSP-D solution can be decomposed into a set of truck arcs, drone legs, and combined arcs. Therefore, a complete solution can be generated by sequentially adding a truck arc, a drone leg, or a combined arc at a time to a *partial* solution which is a concatenation of operations and combined legs possibly followed by a concatenation of truck arcs.

In the DP model, the state information we maintain is the tuple (S, i^C, i^T, τ) where S is the set of *forbidden customers*, i.e. those that cannot be visited when transitioning from this state, i^C is the last customer visited by the truck and the drone together, i^T is the last visited by the truck and τ is the time spent by the truck traveling alone since visiting i^C. We start with an initial state $(\{0\}, 0, 0, 0)$. A state transition takes place when a *control* is applied to a state. For notational ease, we use $A = (S, i^C, i^T, \tau)$ to denote a state and ω to denote a control. $\Gamma(A, \omega)$ is defined as the new state after transitioning from A by control ω. The corresponding transition cost is denoted as $\gamma(A, \omega)$. Controls can be divided into the following three categories:

1. *Add a truck arc.* This control is denoted as T_j where j is the customer or the depot to be visited by the truck alone.

   - If $|S| < n$, for each customer $j \in N \backslash S$, a truck arc can be added to visit j, i.e. the truck travels to j alone, and the current state transitions to $\Gamma(A, T_j) = (S \cup \{j\}, i^C, j, \tau + t^T_{i^T j})$ with cost

$\gamma(A, T_j) = t^T_{i^T j}$;

- Else if $|S| = n$, a truck arc can be added to visit the depot $0'$, i.e. the truck returns to the depot, and the current state transitions to $\Gamma(A, T_{0'}) = (S \cup \{0'\}, i^C, 0', \tau + t^T_{i^T 0'})$ with cost $\gamma(A, T_{0'}) = t^T_{i^T 0'}$;

- Else (consequently $|S| = n + 1$), no truck arc can be added – in this case, either both vehicles are on their way back to the depot together, in which case $0' \notin S$, or the truck is at the depot with the drone finishing the last visit separately, in which case $0' \in S$.

2. *Add a drone leg.* This control is denoted as D_j where j is the customer to be visited by the drone alone. For each customer $j \in N \backslash S$, a drone leg can be added to visit j, i.e. the drone is dispatched at i^C to visit j and rejoins the truck at i^T. As a result, the current state A transitions to $\Gamma(A, D_j) = (S \cup \{j\}, i^C, i^C, 0)$ with cost $\gamma(A, D_j) = \max\{0, t^D_{i^C j} + t^D_{j i^T} - \tau\}$.

3. *Add a combined arc.* This control is denoted by C_j where j is the customer or the depot to be visited by the truck and the drone together. Controls of this type are allowed only if $i^C = i^T$. When this condition holds, we further split into the following two cases:

   - If $|S| < n - 1$, for each customer $j \in N \backslash S$, a combined arc can be added to visit j, i.e. the truck and the drone travels to j together. As a result, the current state A transitions to $\Gamma(A, C_j) = (S \cup \{j\}, j, j, 0)$ with cost $\gamma(A, T_j) = t^T_{i^T j}$;

   - If $|S| = n + 1$, a combined arc can be added to visit $0'$, i.e. the truck and the drone returns to the depot together. As a result, the current state A transitions to $\Gamma(A, C_{0'}) = (S \cup \{0'\}, 0', 0', 0)$ with cost $\gamma(A, C_{0'}) = t^T_{i^T 0'}$.

Example 3. The solution in Figure 5.1 can be decomposed into the following ordered sequence of controls: $(T_1, D_2, C_3, C_4, T_6, T_7, D_5, T_{0'}, D_8)$, with the following respective transition costs: $t^T_{01}, \max\{0, t^D_{02} + t^D_{21} - t^D_{01}\}, t^T_{13}, t^T_{34}, t^T_{46}, t^T_{67}, \max\{0, t^D_{45} + t^D_{57} - (t^T_{46} + t^T_{67})\}, t^T_{70'}, \max\{0, t^D_{78} + t^D_{80'} - t^T_{70'}\}$. It can be verified that the sum of transition costs coincide with the duration of this route.

Remark 5.4.1. By definition, $|S| \leq n + 2$. In particular this holds for terminal states. In fact, we slightly modified the transition function so that there exists exactly one terminal state, namely $(N \cup \{0, 0'\}, 0', 0', 0)$, which makes it easier to compile a DD (Section 5.4.3). Another slight modification is on the transition cost. In particular, in our model, the transition cost of adding a truck arc (drone leg, respectively) to visit customer j is $t^T_{i^T j}$ ($\max\{\tau, t^D_{i^C j} + t^D_{j i^T} - \tau\}$, respectively). On the other hand, their corresponding truck arc and drone leg costs are 0 and $\max\{\tau, t^D_{i^C j} + t^D_{j i^T}\}$ respectively. In other words, their DP model 'delays' adding the accumulated time spent by the truck traveling alone until a drone leg is added. We choose to define the transition cost in such a way that it gives a more accurate estimate of the state quality, which is helpful when we merge states to create a relaxed DD in Section 5.5.

Given the set of partial TSP-D solutions, we define the function

$$f(S, i^C, i^T, \tau) \tag{5.2}$$

as the minimum duration of any partial TSP-D solution that starts from the depot and visits the set of customers S, where the last combined customer visited by the truck and the drone together is i^C, the last vertex visited by the truck is i^T and the time by the truck traveling alone since visiting i^C is τ. The function $f(\{0\}, 0, 0, 0)$ is initialized as 0. Based on this and the observation that $(N \cup \{0, 0'\}, 0', 0', 0)$ is the only terminal state (Remark 5.4.1), it is straightforward to show that:

Lemma 5.4.1. *Value $f(N \cup \{0, 0'\}, 0', 0', 0)$ is equal to the minimum completion time t^* of the TSP-D.*

5.4.2 Basic Definitions of Decision Diagrams

For our purposes, a *decision diagram* (DD) will represent the set of solutions to an optimization problem P defined on an ordered set of decision variables $X = (x_1, \ldots, x_m)$. The feasible set of P is denoted as Sol(P).

A decision diagram for P is a layered directed acyclic graph $D = (N_D, A_D)$ with node set N_D and arc set A_D. D has $m + 1$ layers that represent state-dependent decisions. The first layer is a single root node r and the last layer is a single terminal node t. Layer j is a collection of nodes associated with the variable x_j, for $j = 1, \ldots, m$. An arc $a \in A_D$ is directed from a node in layer j to a node in layer $j + 1$ and has an associated label $l(a)$ to denote the value of decision variable x_j. Each arc, and each node, must belong to a path from r to t. Each arc-specified r-t path $P = (a_1, a_2, \ldots, a_m)$ defines a variable assignment by letting $x_j = l(a_j)$ for $j = 1, \ldots, m$. We slightly abuse the notation and denote by Sol(D) the collection of variable assignments defined by all r-t paths in D.

Definition 5.4.1. A decision diagram D is *exact* for problem P if Sol(D) = Sol(P). A decision diagram D is *relaxed* for problem P if Sol(D) $\supset$ Sol(P).

The benefit of using decision diagrams for representing solutions is that *equivalent* nodes, i.e., nodes with the same set of completions, can be merged. A decision diagram is called *reduced* if no two nodes in a layer are equivalent. For most applications, however, even reduced exact decision diagrams may be exponentially large to represent all feasible solutions. In this case, heuristics can be devised to further merge states to reduce the size, resulting in a relaxed decision diagram.

5.4.3 DD Compilation based on DP

On a high level, the state transition graph of a DP corresponds to a DD. Recall $A = (S, i^C, i^T, \tau)$ is the state we maintain, ω denotes a control and $\Gamma(A, \omega)$ denotes the new state after transitioning from A by control ω with transition cost equal to $\gamma(A, \omega)$. Let $\theta(A)$ be the set of feasible controls, i.e. those that can be applied to A. The *state transition graph* of a DP is defined recursively. The initial state corresponds to a node of the graph, and for a state A that corresponds to a node of the graph and a control $\omega \in \theta(A)$, there is an arc from that node to a node corresponding to $\Gamma(A, \omega)$. The arc has length equal to the transition cost $\gamma(A, \omega)$. A shortest path from the initial state to the terminal state corresponds to an optimal solution to the TSP-D. This state transition graph can be regarded as a DD, in which each layer (except the terminal layer) corresponds to a stage of the recursion.

The above recursive definition can be converted into an algorithm. Let D be the DD which is being compiled. In the algorithm, we represent D as a vector of $n + 2$ 'layers', where each layer L_k is a data structure called unordered_map that stores {key, value} mappings based on hashing values of keys. Therefore Keys are required to be unique. An unordered_map has a field called `keys` that denote the set of keys from mappings that are stored. It has two methods called `insert` that takes a mapping as the input and inserts that mapping if its key does not exists already, and `find` that takes a key as the input and returns the mapping with that key if it exists and a null pointer if not. For the TSP-D, a key in each layer is a state in the DP model and a value is the graph node corresponding to that state. The main reason for using an unordered_map instead of other data structures such as a vector is that search based upon key values takes constant time, which is useful, e.g., when we need to check the existence of certain states during the top-down compilation. A node u in our DD is a class object which has a field `u.A` storing state information (S, i^C, i^T, τ). Node u also contains a field `u.arcs` that stores its outgoing arcs. The field `u.arcs` is an unordered_map where keys are arc labels and values are (arc length, arc head) pairs.

The compilation algorithm (Algorithm 6) begins by initializing a root layer and a root node (line 1- 3). Then we iteratively compile the DD as follows: for each node in layer k ($k = 1, \ldots, n + 1$), we apply a feasible control to its corresponding state (line 6). We check if the new state already exists among keys of the next layer. If not, we create a new node corresponding to this new state in the next layer, and add an arc between those two nodes (line 9- 12).

By construction an arc label represents a control in the DP model. Given a root-terminal path $P = (a_1, \ldots, a_{n+1})$ in a DD constructed by Algorithm 6, it can be verified that $(l(a_1), \ldots, l(a_{n+1}))$ corresponds to a feasible route for the TSP-D. The converse is also true by construction. Therefore we have:

Lemma 5.4.2. *There is a one-to-one correspondence between a root-terminal path in D and a feasible route for the TSP-D.*

Algorithm 6: Exact DD compilation

```
create root layer L_1 and root node u_r
u_r.A ← ({0}, 0, 0, 0)
L_1 ← {u_r.A, u_r}
for k = 1, ..., n + 1 do
    Initialize layer L_{k+1}
    for all u ∈ L_k and ω ∈ θ(u.A) do
        A' ← Γ(u.A, ω)
        if A' ∉ L_{k+1}.keys then
            create node u'
            u'.A ← A'
            L_{k+1}.insert({A', u'})
            u.arcs.insert({ω, (u', γ(A, ω))}
```

5.4.4 Lower Bound from Set Partitioning

Our framework is based on the *set partitioning* (SP) formulation, where an element represents a customer and a set represents a route (Definition 5.3.1). Let R be a valid route relaxation, i.e. a route set containing all feasible routes (and possibly infeasible ones). Given a route r, let c_r be the duration of route r. Let a_{ir} be the number of times customer i is visited in r. Define z_r as the indicator that equals 1 if r is chosen and 0 otherwise. The set partitioning formulation is defined as follows:

$$t^* := \min \sum_{r \in R} c_r z_r \tag{SP}$$

$$\text{s.t.} \sum_{r \in R} z_r = 1 \tag{5.3}$$

$$\sum_{r \in R} a_{ir} z_r = 1, \qquad \forall i \in N \tag{5.4}$$

$$z_r \in \{0, 1\}, \qquad \forall r \in R \tag{5.5}$$

where constraints (5.3) ensure only one route is chosen in the end, constraints (5.4) ensure that each customer is visited exactly once and constraints (5.5) are integrality constraints.

A feasible solution to (SP) corresponds to a feasible route. However solving (SP) (i.e. the set partitioning problem) is NP-hard in general. The state-of-the-art algorithm (Roberti and Ruthmair, 2019) relies on the linear relaxation of the SP, also referred to as the *master problem*, to obtain a lower bound and use it in a branch-and-bound framework. In this chapter, we also focus on this linear relaxation. For notational ease, we denote it by SPLP(R) to emphasize that this LP is defined w.r.t. a route relaxation R. As R contains

an exponential number of routes, solving the LP is a nontrivial task. Indeed, (Roberti and Ruthmair, 2019) uses column generation, where the pricing problem is to find routes in R with negative reduced costs. They observe that if the route set R only contains feasible routes, solving the pricing problem is as complicated as solving the TSP-D itself. On the other hand, the more infeasible routes R contains, the worse the lower bound value becomes. In order to obtain a better trade-off between the pricing problem complexity and the lower bound quality, they propose *ng-route relaxation*, and solve the resulting pricing problem via DP. The DP model for the ng-route relaxation allows infeasible routes of certain structure (see Section 5.5.1 for more details).

Our approach is inspired by connections between their DP model and decision diagrams. More specifically, recall Lemma 5.4.2 shows that solving a DP model corresponds to finding a shortest path from the root to the terminal on the corresponding DD. As a consequence, a relaxed DD for the TSP-D gives rise to a route relaxation. Indeed, the ng-route relaxation can be viewed as a relaxed DD. This simple observation motivates us to adapt generic techniques developed by the DD community to compile and refine relaxed DDs, which leads to novel route relaxations (Section 5.5). Furthermore, in Section 5.6 we present two algorithms that compute lower bounds given a relaxed DD, without using CG to solve the SPLP defined w.r.t. its corresponding route relaxation. Interestingly, it is shown that our approaches are equivalent to the CG approach in the sense that they all obtain the same lower bound under the same route relaxation. Finally in Section 5.7 we integrate all the techniques and present our iterative framework.

5.5 Route Relaxation

First we briefly describe the ng-route relaxation for the TSP-D by Roberti and Ruthmair (2019) and explain why this relaxation corresponds to a relaxed DD. Then we apply generic techniques from the DD literature to generate alternative route relaxations and discuss how to refine our DDs.

5.5.1 ng-Route Relaxation

Let us define for each customer $i \in N$, a set of customers $N_i \subset N$ that contains the *neighborhood* of i, i.e. the so-called *ng-set*. The ng-set usually contains the customers that are closest to i. An ng-route is a not necessarily elementary route where a customer $i \in N$ is visited more than once if and only if there exists at least one customer j visited in between the two consecutive visits to i such that $i \notin N_j$. The size of the sets N_i determines the subtours allowed in the ng-routes, the quality of the lower bound returned by solving the SPLP, and the difficulty of the pricing problem. Indeed, the larger the set N_i, the fewer subtours are allowed in the ng-routes, the better the lower bound, but also the more time-consuming the pricing problem to solve. In the extreme case where $N_i = N$ for each customer $i \in N$, ng-routes cannot contain any subtour, so the SPLP provides the best possible bound among all route relaxations, but the pricing problem would be as

complicated as solving the TSP-D itself. It is typically a trial-and-error process to choose a suitable size for N_i for a specific application.

There exists a DP model for finding the shortest ng-route (Roberti and Ruthmair, 2019). As a result, we can compile a DD in a similar fashion as Algorithm 6. Let us call it D_{ng}. Based on the definition of ng-routes, we can show that:

Proposition 5.5.1. *D_{ng} is a relaxed DD for the TSP-D.*

Proof. By definition, the set of feasible routes is contained in the set of ng-routes. Therefore D_{ng} is a relaxed DD by Definition 5.4.1. □

It is noted that BCP algorithms do not explicitly construct D_{ng} to solve the pricing problem. Instead they rely heavily on dominance rules to reduce the number of states to examine. More details on dominance rules can be found in Roberti and Ruthmair (2019).

5.5.2 DD-based Route Relaxation

Generic techniques for creating relaxed DDs exist in the DD literature. In light of Proposition 5.5.1, a natural question is: can we apply those techniques to produce a relaxed DD different from D_{ng}, and how good is the lower bound derived from this DD? Below we begin our investigation along these lines.

A relaxed DD can be compiled in either of the following two ways: one is by top-down compilation and the other is by separation (Bergman et al., 2016a). The former one compiles a DD recursively starting from the root based on a DP model, as described in Section 5.4.3. Additionally, one should also have a rule describing how to *merge* nodes and perhaps how to adjust transition costs in a layer to ensure that the output DD will (a) be a relaxed DD and (b) the length of each root-terminal path does not increase. The underlying goal of this rule is to create a relaxed DD with a manageable size that provides a tight lower bound. The latter one starts with a relaxed DD and applies constraint separations iteratively, i.e. changing the node and arc set of the DD to remove the infeasible solutions that violate a particular constraint of the problem. This iterative process can simply be stopped when the size of a layer grows too large and the output DD is still a relaxed DD. In this section, we create an initial relaxed DD by top-down compilation with a problem-specific merging rule. This rule consists of a *merging heuristic* and a *merging operation* which determine how to identify a set of nodes to merge and how to merge a set of nodes, respectively.

Merging Heuristic. Recall from the DP model in Section 5.4.1 that a state A for the TSP-D is represented as a tuple (S, i^C, i^T, τ) where S is the set of customers visited so far, i^C is the last visited combined customer, i^T is the last visited truck customer and τ is the time spent by the truck traveling alone since leaving

i^C. To motivate our merging heuristic, consider the following case: two nodes to be merged have states $A_1 = \{S_1, i_1^C, i_1^T, \tau_1\}$ and $A_2 = \{S_2, i_2^C, i_2^T, \tau_2\}$ respectively. To ensure no feasible solution is lost after merging, the merged state should keep track of (i_1^C, i_1^T) and (i_2^C, i_2^T) simultaneously, since the feasible set of controls of a state is related to its (i^C, i^T) pair (Section 5.4.1). As a result, we may end up using the set representation $\{i_1^C, i_2^C\}$ and $\{i_1^T, i_2^T\}$ for i^C and i^T in the merged state, which is yet to be defined. Although it is possible to define this representation, it leads to further issues such as a potentially loose relaxation. We avoid these complications by simply forbidding A_1 and A_2 from merging if $i_1^C \neq i_2^C$ or $i_1^T \neq i_2^T$. Given a set of nodes whose states agree on (i^C, i^T) values, we apply a simple greedy heuristic for merging. This merging heuristic is described below.

For a given layer, we first partition nodes by the value of (i^C, i^T) pair. A *bucket* B is defined as the set of states within one partition class. The *bucket size* $|B|$ is defined as the number of states in bucket B. Let M be a parameter chosen a priori which denotes the maximum bucket size allowed after merging. For a bucket B of size $|B|$, the greedy merging heuristic is defined as follows: if $|B| < M$, no merging needs to be done; Else we sort nodes in B in increasing order of the length of the shortest path from the root to those nodes. We then merge the last $|B| - M + 1$ nodes according to the merging operation defined below. By merging nodes with larger shortest path lengths, we keep those most 'promising' nodes at the current layer exact and relax those less promising ones which are less likely to participate in the set of optimal solutions.

Remark 5.5.1. Given M, the maximum bucket size allowed after merging, the number of nodes in a layer is at most $O(Mn^2)$ since there are at most $O(n^2)$ buckets. Therefore the overall size of the resulting relaxed DD is at most $O(Mn^3)$.

Merging Operation. Given a set of q states $A_1 = \{S_1, i^C, i^T, \tau_1\}, \ldots, A_q = \{S_q, i^C, i^T, \tau_q\}$ that agree on (i^C, i^T), we use the symbol $\oplus$ as our merging operator and denote the merged node as $A' := \oplus_{i=1}^q A_i$. In order not to exclude any feasible solution, the set of customers that cannot be visited when transitioning from A' is defined as the intersection among visited customer sets, i.e. $\cap_{i=1}^q S_i$. To define the state variable τ of A', notice that the only impact τ has is on the cost of taking a drone leg when transitioning from A' (recall the definition of the transition function in Section 5.4.1). In particular, the large τ becomes, the smaller the transition cost becomes. Therefore we define τ of state A' as $\max_{i=1,\ldots,q} \tau_i$. To sum up, the merging operation is defined as:

Definition 5.5.1 (Merging Operation)**.**

$$\oplus_{i=1}^q A_i := (\cap_{i=1}^q S_i, i^C, i^T, \max_{i=1,\ldots,q} \tau_i)$$

Given D, a relaxed DD and a root-terminal path $P = (a_1, \ldots, a_{n+1})$ in D, let $r(P)$ denote the corresponding route, i.e. $r(P) = (l(a_1), \ldots, l(a_{n+1}))$. Let $\gamma_D(P)$ denote the length of path P in D, i.e.

$\gamma(P) = \sum_{i=1}^{n+1} \gamma(u_i.A, l(a_i))$, where $(u_1, \ldots, u_{n+1})$ is the ordered sequence of nodes on P. Recall $c_{r(P)}$ denotes the duration of route $r(P)$. When D is exact, we have $\gamma_D(P) = c_{r(P)}$. When D is relaxed, as we shall see in Example 4, it may happen that $\gamma_D(P) < c_{r(P)}$.

Example 4. Consider an instance of 4 customers and the following two routes: $r_1 = (T_1, T_2, D_4, C_3, C_{0'})$ and $r_2 = (T_3, T_2, D_4, C_1, C_{0'})$. For all $a \in A$, $t_a^T = t_a^D$. $t_{01}^T = t_{12}^T = t_{23}^T = 1, t_{04}^T = t_{42}^T = 2, t_{03}^T = 3$. Remaining arc lengths are omitted in this example. After taking the first two controls, we arrive at $A_1 = (\{0, 1, 2\}, 0, 2, 2)$ and $A_2 = (\{0, 2, 3\}, 0, 2, 4)$ respectively. Suppose we are to merge A_1 and A_2. According to the definition, the merged state $A' = (\{0, 2\}, 0, 2, 4)$. No other states are further merged. Call the resulting diagram D. Let P_1 and P_2 be root-terminal paths in D corresponding to r_1 and r_2 respectively. Notice both of them passes through A'. It can be verified that $\gamma_D(P_1) = 1 + 1 + (4 - 4) + 1 + 3 = 6$ whereas $c_{r_1} = 1 + 1 + (4 - 2) + 1 + 3 = 8$. The discrepancy is due to merging A_1 and A_2. Also notice that from A', customer 1 or 3 may be visited again, which creates non-elementary routes.

Based on the above analysis, we have:

Lemma 5.5.2. *Applications of the merging operation produce a relaxed decision diagram.*

For a TSP-D instance, Lemma 5.5.2 guarantees that we output a relaxed DD by applying the above merging rule. The set of root-terminal paths of this relaxed DD is thus a valid route relaxation that can be used to compute a lower bound from the SPLP. This observation can also be applied to a general vehicle routing problem, i.e., if the DP model for that problem is available, and a valid merging rule can be defined which leads to a relaxed DD, then the resulting relaxed DD is a route relaxation for that problem. As we shall see in Section 5.9.1, however, that this initial bound is typically not competitive to the state-of-the-art one from the ng-route relaxation. To bridge this performance gap, we further refine the DD as described below in Section 5.5.3.

5.5.3 Conflict Refinement

We describe two refinement techniques used to strengthen a relaxed DD. Our techniques are applied to a *prescribed* path associated with certain *conflicts* which we define below. Throughout this section, we assume we are given D, a relaxed DD compiled according to the merging rule in Section 5.5.2, and P, a root-terminal path upon which we wish to refine (if any conflicts). We defer the discussion on how to find such paths to Section 5.6.

Our techniques are very similar to constraint separation in van Hoeve (2020). Generally speaking, constraint separation refers to the process of changing the node and arc set of a relaxed DD to remove the infeasible solutions that violate a particular constraint of the problem. In the case of a relaxed DD for the TSP-D, the only constraint on a path is that it should be elementary, i.e. it visits each customer exactly once. Given a root-terminal path P, we say P is associated with a *repetition conflict* if it is non-elementary. Moreover,

there is another conflict that can be associated with P due to the merging operation: recall from Example 4 that if there is a relaxed state along P, it may happen that $\gamma_D(P) < c(r_P)$. We refer to this conflict as *inexact distance conflict* since it is a consequence of relaxing the state variable τ for some state along path P. Notice that inexact distance conflict can happen to a path with no repetition conflict. We identify a repetition conflict with a pair (j, k) where the j-th arc label l_j and the k-th arc label l_k along this path represents visiting the same customer (either by a truck, drone or both). To refine this conflict, we further require that there exists no repetition conflict (j', k') such that $j \leq j' < k' < k$. We identify an inexact distance conflict with a single index j where the state variable τ of the *next* state u_{j+1} is relaxed due to merging states. Next we describe two algorithms – refineRep and refineInexactDist that change the node and arc set in order to fix repetition and inexact distance conflicts respectively.

Recall from Section 5.4.3 that a DD is a vector of layers where each layer L_k ($k = 1, \ldots, n + 2$) is an unordered_map that stores (state, node) pairs. Each node u has two fields `u.A` that stores state information and `u.arcs` that stores outgoing arcs. `u.arcs` is an unordered_map where keys are controls and values are (arc length, arc head) pairs. An unordered_map has `keys` as its field, and `insert` and `find` as its methods. For a mapping M, its key is denoted as M.key.

Function refineRep (Algorithm 7) considers each node along the path in sequence and splits off the next node in the path. This is done by first creating a temporary node v by applying the transition function with control l_j (line 3). If there already exists a node v' in L_{i+1} with an equivalent state, we reassign v to represent v' (line 6-7). Otherwise, we complete the definition of v by copying the outgoing arcs of u_{i+1} whenever the arc does not introduce an additional repetition conflict (line 9-11). We add v to layer L_{i+1} (line 12). Function `redirectArc`(u_i, l_i, v) (line 13) redirects the arc with label l_i going out of u_i by changing the

arc head from u_{i+1} to v. We set u_{i+1} to be v and iterate.

Algorithm 7: Function refineRep(D, P, j, k): refining repetition conflict (j, k) in decision diagram D

Input: decision diagram D, a path P with repetition conflict (j, k) (it is assumed that the path contains no edge conflicts (j', k') such that $j \leq j' < k' < k$)

Output: decision diagram in which the repetition conflict (j, k) along the path has been eliminated

```
1  for i = j, ..., k − 1 do
2      create state v
3      v.A ← Γ(u_i, l_i)                                  // split the path towards node v
4      if v.A ∈ L_{i+1}.keys                              // check for equivalent states
5      then
6          v' ← L_{i+1}.find(v.A)                         // find node v' in L_{i+1} such that v'.A = v.A
7          v ← v'
8      else
           /* copy outgoing arcs from u_{i+1} which do not cause additional repetition
              conflicts                                                                  */
9          for all mapping M in u_{i+1}.arcs do
10             if M.key ∈ θ(u_{i+1}.A) then
11                 v.arcs.insert(M)
12         L_{i+1}.insert(v)
13         redirectArc(u_i, l_i, v)
14     u_{i+1} ← v
```

Function refineInexactDist splits off from u_j by creating a temporary node v by applying the transitioning function with control l_j. If there already exists a node v' with an equivalent state in L_{j+1}, we reassign v to represent v' (line 4-6). Otherwise we complete the definition of v by copying the outgoing arcs of u_{j+1} whenever the arc does not introduce an additional repetition conflict (line 8-10). Finally, we add v to layer

$j+1$ (line 11) and redirect arcs accordingly (line 12).

Algorithm 8: Function refineInexactDist(D, P, j): refining inexact distance conflict j in decision diagram D

Input: decision diagram D, a path P with inexact distance conflict j

Output: decision diagram in which the inexact distance conflict j along the path has been eliminated

```
1  create state v
2  v.A ← Γ(u_j, l_j)                                   // split the path towards node v
3  if v.A exists among keys of L_{i+1}                  // check for equivalent states
4  then
5      v' ← L_{i+1}.find(v.A)                          // find node v' in L_{i+1} such that v'.A = v.A
6      v ← v'
7  else
       /* copy outgoing arcs from u_{i+1} which do not cause additional repetition
          conflicts                                                                  */
8      for all mapping M in u_{i+1}.arcs do
9          if M.key ∈ θ(u_{i+1}.A) then
10             v.arcs.insert(M)
11     L_{i+1}.insert(v)
12     redirectArc(u_i, l_i, v)
```

5.6 Lower Bound Computation

In the remainder of this chapter, any route relaxation under consideration will always correspond to the set of root-terminal paths in a relaxed decision diagram D. For notational ease, we let R_D denote this route relaxation. In this section, we first describe two alternative approaches to derive lower bounds from R_D. We then show the equivalence between lower bounds derived from our approaches and the optimal value of SPLP(R_D). Throughout this section, we assume that we are given a precompiled relaxed DD $D = (V_D, A_D)$.

Constrained Flow Model

We construct a network flow model with side constraints where the network is D. Let r and t denote the root and terminal of D respectively. For an arc $a \in A_D$, let $\omega(a)$ be the arc label that represents the control which encodes which customer to visit and how to visit the customer. Let l_a be the length of arc a. For a vertex $u \in V_D$, let $\delta^+(u)$ and $\delta^-(u)$ denote the set of outgoing arcs from u and the set of incoming arcs into

u respectively. For a customer $i \in N$, let $\xi(i)$ denote the set of arcs whose label represent visiting customer i (either by truck, drone or both). For each $a \in A_D$, define an indicator variable y_a that equals one if a is chosen and 0 otherwise. The constrained flow model is defined as follows:

$$\min \quad \sum_{a \in A_D} \gamma_a y_a \tag{5.6}$$

$$\text{s.t.} \quad \sum_{a \in \delta^+(u)} y_a = \sum_{a \in \delta^-(u)} y_a, \quad \forall u \in V_D, u \neq r, t \tag{5.7}$$

$$\sum_{a \in \delta^+(r)} y_a = 1 \tag{5.8}$$

$$\sum_{a \in \delta^-(t)} y_a = 1 \tag{5.9}$$

$$\sum_{l(a) \in \xi(i)} y_a = 1, \quad \forall i \in N \tag{5.10}$$

$$y_a \in \{0, 1\}, \quad \forall a \in A_D \tag{5.11}$$

where constraints (5.7) (5.8) (5.9) ensure flow conservation on intermediate nodes, out-flow from the root and in-flow into the terminal respectively. Constraint (5.10) ensures that each customer is visited exactly once.

A feasible solution to the constrained flow model is a root-terminal path in D that visits each customer exactly once. Therefore an optimal solution to the constrained flow corresponds to an optimal route for the TSP-D. However, in general it is NP-hard to solve the constrained flow model. To derive a lower bound, we relax integrality constraints (5.11). We refer to this linear relaxation as CFLP(D). We can solve CFLP(D) by an off-the-shelf LP solver.

Lagrangian Relaxation

Another way of deriving lower bounds is to apply Lagrangian relaxation to the above constrained flow model by dualizing constraints (5.10). For each customer $i \in N$, define $\boldsymbol{\lambda}$ as the vector of *Lagrangian multipliers* where λ_i corresponds to constraint $\sum_{\omega(a) \in \xi(i)} y_a = 1$, which ensures that customer i is visited exactly once. The Lagrangian subproblem is defined as:

$$\psi_D(\boldsymbol{\lambda}) = \min \quad -\sum_{i \in N} \lambda_i + \sum_{a \in A_D} (l_a + \sum_{i \in N} \{\omega(a) \in \xi(i)\} \lambda_i) y_a$$

$$y \text{ subject to constraint (5.7) (5.8) (5.9)}$$

where $\{\omega(a) \in \xi(i)\}$ is an indicator function that equals 1 if $\omega(a) \in \xi(i)$ and 0 otherwise. We omit subscript D in $\psi_D(\boldsymbol{\lambda})$ when there is no ambiguity.

Observe that the Lagrangian subproblem can be seen as finding a shortest path on D whose arc lengths are modified in the following way: for each arc $a \in A_D$, let i be the customer that $\omega(a)$ visits, we increase arc length l_a by λ_i. Furthermore, for each arc a leaving the root, we decrease arc length l_a by the sum of Lagrangian multipliers: $\sum_{j \in N} \lambda_j$. The Lagrangian dual problem is defined as the following maximization problem:

$$\max_{\lambda} \psi(\boldsymbol{\lambda}) \tag{5.12}$$

We can solve the Lagrangian relaxation by the subgradient method, which is an iterative procedure that updates dual multipliers according to a subgradient direction and a step size. More specifically, let t be the iteration index of this method. Define $\boldsymbol{\lambda}^{(t)}$ as the vector of dual multipliers, $\boldsymbol{g}^{(t)}$ as the subgradient at $\boldsymbol{\lambda}^{(t)}$ for iteration t and α_t as the step size at iteration t. Then the method updates dual multipliers in the following way:

$$\boldsymbol{\lambda}^{(t+1)} = \boldsymbol{\lambda}^{(t)} + \alpha_t \boldsymbol{g}^{(t)}, \tag{5.13}$$

where $\boldsymbol{g}^{(t)} := (g_1^{(t)}, \ldots, g_n^{(t)})$ can be computed as follows. Let $P^{(t)}$ be an optimal path found by solving $\psi(\boldsymbol{\lambda}^{(t)})$. Let s_i be the number of times $P^{(t)}$ visits customer i, it can be verified that $g_i^{(t)}$ can be set as $s_i - 1$. We defer the discussion on our step size strategy to Section 5.8.

Equivalence of SPLP(R_D), CFLP(D) and LR(D)

We next show that optimal values of CFLP(D) and LR(D) are in fact equal to that of SPLP(R_D). For notational ease, let $v(P)$ denote the optimal value of an optimization problem P. We show that

Theorem 5.6.1. $v(\mathit{SPLP}(R_D)) = v(\mathit{CFLP}(D)) = v(\mathit{LR}(D))$.

Proof. For the first equality, it is sufficient to observe that SPLP(R_D) is the path formulation and CF(D) is the flow formulation for the same problem. More formally, a route $r \in R_D$ corresponds to a path from the root to the terminal in D. For an arc $a \in A_D$, let $\beta(a)$ be the set of paths (routes) that contain a. For any feasible solution $\{z_r : r \in R_D\}$ to SPLP(R_D), there is a feasible solution to CF(D) with the same objective value where $\forall a \in A_D, y_a = \sum_{r: r \in \beta(a)} z_r$. Conversely, for any feasible solution to CF(D), there exists a path decomposition of the flow, which corresponds to a feasible solution to SP(D) with the same objective value.

The second equality is a direct application of Theorem 1 in Geoffrion (1974) that establishes the equivalence between the value of the Lagrangian relaxation and that of a primal relaxation, and the fact that the flow

conservation constraints form an integral polyhedral. □

CFLP(D) produces a bound that can only be better than $\psi_D(\boldsymbol{\lambda})$. On the other hand, solving the CFLP can become computationally prohibitive as the instance size grows, as we shall see in Section 5.9.3. In this case, we can instead solve the Lagrangian subproblem that computes a lower bound for any multipliers $\boldsymbol{\lambda}$ by solving a simple shortest path problem.

Finally, we remark that an optimal solution to CFLP(D) can be decomposed into a set of paths while an optimal solution to the Lagrangian subproblem is a single path. In both cases, repetition or inexact distance conflicts may be detected and refined on those paths. This motivates us to iteratively perform lower bound computation and conflict refinements in Section 5.7.

5.7 Iterative Framework

In this section, we describe two iterative frameworks based on the constrained flow model and Lagrangian relaxation respectively. Throughout this section, we assume that a relaxed DD D is initially compiled according to the merging rule in Section 5.5.2 with a prescribed bucket size. Our goal is to iteratively refine D in order to tighten the corresponding route relaxation and consequently improve the lower bound for the TSP-D. Each iterative algorithm presented below can be terminated at the end of an arbitrary iteration and can produce a valid lower bound. In this section, we leave the termination criterion unspecified to make our frameworks general. A time limit is imposed when we perform computational experiments in Section 5.9.

Combine CFLP with Conflict Refinement

Conflict refinement can be combined with the constrained flow model to improve the lower bound, as shown in function flowRefine (Algorithm 9). Starting from an initial relaxed DD D, the algorithm solves CFLP(D) which returns an optimal LP solution y^* with optimal LP value v^* (line 2). It then computes a path decomposition on the support graph of y^* (line 3), i.e. a subgraph of D with vertex set V_D and set of arcs with nonzero y_a^* values. For each path contained in the decomposition, we check and refine its conflicts (line 5) This iterative procedure continues until some termination criterion is met.

Function pathDecomp (Algorithm 10) computes a path decomposition corresponding to an optimal CFLP solution x^* in a heuristic fashion. An arc is implemented as a class object and has a field named `head`, which is the head node of that arc. The algorithm decomposes the flow x^* in the following iterative way: while there is enough flow remaining, it starts from the root and recursively follows the arc going out of the

Algorithm 9: Function flowRefine(D): iterative refinement with CFLP

Input: relaxed decision diagram D
Output: refined decision diagram

```
while termination condition is not met do
    (v*, x*) ← solveCF(D)
    𝒫 ← pathDecomp(D, x*)
    for P ∈ 𝒫 do
        refinePath(D, P)
```

current node with the most residual flow until the terminal is reached (line 5 - line 10). We add such a new path to the set of paths $\mathcal{P}$ and then updates the residual flow on each arc in this path (line 11 - line 12).

Function refinePath (Algorithm 11) detects inexact distance and repetition conflicts and refine them in sequence (if any). Function findRep(p) finds the first repetition conflict (j, k) along path p for which there is no conflict (j', k') such that $j \leq j' < k' < k$. Similarly function findInexactDist(p) finds the first inexact distance conflict j along path p. More details on implementations are discussed in Section 5.8.

Algorithm 10: Function pathDecomp(D, y^*): a path decomposition based on an optimal CFLP solution

Input: relaxed decision diagram D, optimal solution to CFLP y^*
Output: set of paths $\mathcal{P}$

```
1  𝒫 ← ∅                                                  // 𝒫 stores the set of paths
2  while enough flow remains do
3      create path P                          // P stores a sequence of arcs (currently empty)
4      v ← D.root, f ← 1                                          // starting from the root
5      while v ≠ D.terminal do
6          a ← findArcWithMostFlow(v.arcs, x*)      // find arc a which carries the most flow
7          P.add(a)                                              // extend the path with arc a
8          v ← a.head
9          if y*_a < f then
10             f ← y*_a                                // update the flow value to subtract from
11     for a ∈ P do
12         y*_a ← y*_a − f                  // update the residual flow for each arc in the path
```

Combine Lagrangian Relaxation with Conflict Refinement

It is also possible to combine conflict refinement with Lagrangian relaxation to obtain improved bounds. Notice that,, however, the convergence of the subgradient method will not be guaranteed (in fact, it is not well-defined) once we start to refine paths found during this process, since modifying the route relaxation changes the Lagrangian dual. Therefore, it is better to think of this integration as an iterative process of

Algorithm 11: Function refinetPath(D, P): refine conflicts along a prescribed path

Input: relaxed decision diagram D, a prescribed path P
Output: refined decision diagram

```
i ← findInexactDist(P)
if i ≠ −1                                   // i = −1 means no such conflict exists
 then
    refineInexactDist(D, P, i)
(j, k) ← findRep(P)
if k ≠ −1                                   // k = −1 means no such conflict exists
 then
    refineRep(D, P, j, k)
```

searching for suitable Lagrangian multipliers that lead to strong lower bounds while refining conflicts on paths computed during the search process. As a result, we need to decide when to stop/restart the subgradient method and which paths found during the iterative process to refine upon. Below we describe two heuristic implementations.

Function lagAdapt (Algorithm 12) immediately refines the path found at each iteration of the subgradient method. Lagrangian multipliers are then updated according to the same strategy as if we are solving Lagrangian relaxation on a static DD. This process terminates until a prescribed *improvement criterion* is violated (line 5-11), at which point it fixes multipliers to those corresponding to the best lower bound so far. It then continues the iterative process of finding and refining the shortest path with arc lengths modified w.r.t. those fixed multipliers (line 12-13). Details on the improvement criterion are discussed in Section 5.8.

Function lagRestart (Algorithm 13) restarts the subgradient method periodically according to the *buffer size* H chosen a prior. The subgradient method is reset every h iterations. We refer to the interval between the start and end of the subgradient method as an *epoch*. During an epoch, It runs h iterations of the subgradient method, stores each path found during the epoch (line 10), refines all of them at the end of the epoch (line 12) and restarts from scratch, treating the refined DD as an initial DD.

Remark 5.7.1. In principle, we can integrate CG with conflict refinements as well. This can be useful when solving the CFLP(R_D) becomes computationally prohibitive.

5.8 Implementation Details

This section discusses parameter choices in our implementation.

Algorithm 12: Function lagAdapt(D): iterative refinement that adaptively terminates the subgradient method

Input: relaxed decision diagram D
Output: refined decision diagram

```
λ ← 0
bestBound ← 0
bestMultipliers ← λ
while termination criterion is not met do
    while improvement criterion holds do
        (P, lb) ← findShortestPath(D, λ)
        if bestBound < lb then
            bestBound ← lb
            bestMultipliers ← λ
        updateMultipliers(λ)
        refinePath(D, P)
    (P, lb) ← findShortestPath(D, bestMultipliers)
    refinePath(D, P)
```

5.8.1 Conflict Refinement

Recall that in Algorithm 11, when a path is processed for conflict refinement, we identify the first inexact distance and repetition conflict encountered from the root to the terminal, refine the inexact distance conflict first (if any) and then the repetition conflict (if any). Alternatively, one may try to identify multiple inexact distance or repetition conflicts, or change the order in which the conflicts are refined. Preliminary tests show that

- Refining multiple conflicts (either inexact distance or repetition conflicts) does not necessarily yield stronger lower bounds than refining one conflict of each type. In other words, the improvement of lower bounds do not outweigh the amount of time spent in identifying and refining those multiple conflicts. Therefore we choose to only identify the first inexact distance conflict and the first repetition conflict.

- Reversing the order in which the two types of conflicts are refined leads to worse lower bounds for all test instances. This is due to the following reason: let (i, j) be the pair of layers of the repetition conflict ($i < j$) and k be the layer of the inexact distance conflict. The order in which they are refined matters only when $k > j$, i.e. the inexact distance conflict is below the repetition pair. In this case, refining the repetition first eliminates the existence of the current path. As a result, the inexact distance conflict is left unrefined. Therefore we choose to refine the inexact distance conflict before the repetition conflict.

Algorithm 13: Function lagRestart(D, h): iterative refinement that periodically restarts the subgradient method

Input: relaxed decision diagram D, buffer size H
Output: refined decision diagram

```
1  λ ← 0
2  bestBound ← 0
3  𝒫 ← ∅                          // stores the set of paths that need to be refined upon
4  while termination criterion is not met do
5      while 𝒫.size < H do
6          (P, lb) ← findShortestPath(D, λ)
7          if bestBound < lb then
8              bestBound ← lb
9          updateMultipliers(λ)
10         𝒫.add(P)
11     for P ∈ 𝒫 do
12         refinePath(D, P)
13     𝒫.clear()
14     λ ← 0
```

5.8.2 Path Decomposition

Recall we decompose the flow into multiple paths in Algorithm 9. In practice, we need to set a tolerance value to stop the iterative path finding procedure when the outgoing residual flow from the root is below the tolerance. If this tolerance value is set too small, the number of paths to be further refined upon can be too large. Preliminary tests show that the value below 0.001 produces too may paths and the value above 0.05 is too conservative. We choose 0.01 to be the tolerance value.

5.8.3 Step Size for the Subgradient Method

Recall from Section 5.6 that t is the iteration index of of the subgradient method. $\boldsymbol{\lambda}^{(t)}$ is the vector of dual multipliers, $\boldsymbol{g}^{(t)}$ is the subgradient at $\boldsymbol{\lambda}^{(t)}$ and α_t is the step size at iteration t. The method updates dual multipliers in the following way:

$$\boldsymbol{\lambda}^{(t+1)} = \boldsymbol{\lambda}^{(t)} + \alpha_t \boldsymbol{g}^{(t)}, \tag{5.14}$$

We use an approximate version of Polyak's step length to update our multipliers Boyd et al. (2003). Let ψ^* be the optimal value of the Lagrangian dual problem. Polyak's step length is defined as:

$$\alpha_t = \frac{\psi^* - \psi(\boldsymbol{\lambda}^{(t)})}{\left\|\boldsymbol{g}^{(t)}\right\|_2^2} \tag{5.15}$$

When ψ^* is not known, we can estimate it by $\psi_{\text{best}}(1+\eta_t)$ where ψ_{best} is the best lower bound found so far. η_t satisfies that $\eta_t \to 0$ as $t \to +\infty$. In our implementation, η_t is set to be $0.05 \cdot \frac{100}{100+t}$.

5.8.4 Improvement Criterion for Function lagAdapt

The underlying logic of function lagAdapt (Algorithm 12) is that it switches to pure conflict refinement if updating multipliers via subgradient method is unlikely to further improve the lower bound. We define the improvement criterion as follows: let κ and μ be two parameters chosen a priori. We say the improvement criterion does not hold if in each of the latest κ iterations, the improvement percentage of the best lower bound is below μ. Preliminary tests are performed among $\kappa \in \{100, 200, 300\}$ and $\mu \in \{0.05\%, 0.1\%, 0.15\%, 0.2\%\}$. We find the parameter pair $\kappa = 200, \mu = 0.1\%$ achieves the best performance.

5.8.5 Sensitivity of Bucket and Buffer Size

Each iterative refinement algorithm takes as input an initial relaxed DD, which depends on the maximum bucket size M (Section 5.5.2). As a result, the performance of our algorithms depend on M. In addition, function lagRestart (Algorithm 13) also depends on the buffer size H. We perform preliminary tests to study the sensitivity of our algorithms w.r.t. these two parameters. In particular, they are chosen from $B \in \{2, 4, 6, 8\}$ and $H \in \{10, 20, 30\}$ (H is only used by lagRestart). We found that all our algorithms are not sensitive to B. We set $B = 4$ when compiling relaxed DDs for flowRefine (Algorithm 9) and lagAdapt (Algorithm 12) since the performance is slightly better than other settings. For lagRestart (Algorithm 13), we choose $B = 4, H = 20$ since the performance is both competitive and robust.

5.9 Computational Experiments

We implemented our iterative refinement algorithms in C++ and performed experimental evaluation on a wide range of problem instances. We use CPLEX 12.10 as integer and linear programming solver, using a single thread and the Barrier Method as LP algorithm. All reported experiments were run on a Macbook Pro laptop with 2.2 GHz Quad-Core Intel Core i7 and 16GB memory. For notational ease, let $m = n+1$ denote the number of locations, i.e. the number of customers plus the depot. Each problem instance is generated as follows. Given m, the number of locations, and α, the speed ratio between the drone and the truck, we sample m points uniformly from a 1000×1000 Euclidean plane. We regard the first generated point as the depot and the rest as customers. We take the Euclidean distance between a pair of points as its truck distance and divide it by the speed ratio as the drone distance. We generate 10 instances for each (m, α) pair where $n \in \{15, 20, 25, 30, 40, 50\}$ and $\alpha \in \{2, 4\}$.

We use *optimality gap* as the performance metric. Formally, given an upper bound UB, we measure the quality of a lower bound LB by the optimality gap defined as $\frac{UB-LB}{UB}$. In our experiments, UB is the incumbent value when solving the CP model for the TSP-D (Section 4.5) by CP Optimizer whose time limit is set to one hour. Best lower bounds in the literature are computed by the ng-route-based BCP algorithm (Roberti and Ruthmair, 2019). More precisely, they showed their root LP gaps are very tight which means root LP bounds from the ng-route relaxation are substantially better than other existing approaches. Since their code is not publicly available, we re-implemented their column generation approach based on the ng-route relaxation to solve the LP relaxation SPLP(R). In our implementation, column generation is terminated when computation time exceeds one hour. In this section, the termination criterion of all our algorithms is a prescribed time limit, which varies for different sets of experiments as we describe below.

5.9.1 Size and Lower Bound from Initial Route Relaxation

The first set of experiments studies the quality of lower bounds derived from our initial route relaxations, i.e. SPLP(R_D) where D is a relaxed DD compiled with maximum bucket size M according to the merging rule in Section 5.5.2. Below we report the comparison of the sizes and optimality gaps between our initial route relaxation and the ng-route relaxation. As Roberti and Ruthmair (2019), we set the neighborhood size $|N_i| = 5$ for all $i \in N$ in our ng-route relaxation. The lower bound from our route relaxation is computed by solving CFLP(R_D). The lower bound from the ng-route relaxation is computed by CG. In our implementation, column generation exceeds the time limit for all instances with $m \geq 30$. We report our findings for $m \in \{15, 20, 25\}, \alpha \in \{2, 4\}$ and $M \in \{2, 4, 6, 8\}$.

Number of locations	15	20	25
$M = 2$	4974	12734	26043
$M = 4$	9736	25086	51485
$M = 6$	14498	37438	76928
$M = 8$	19260	49790	102370
ng-route	24114	73554	174725

Table 5.1: Average size of route relaxations. For DD-based route relaxations, the size is the number of nodes in the DD; for ng-route, the size is the number of labels after dominance rules are applied.

Table 5.1 reports the size of our route relaxations as well as the ng-route relaxation. For DD-based route relaxations, the size is the number of nodes in the DD; for ng-route, the size is the number of labels after dominance rules are applied. Recall M is the bucket size after merging. The table shows that the size of our relaxed DDs increases roughly linearly w.r.t. the bucket size M, and is typically smaller than that of the ng-route relaxation. Next we report the optimality gap of our relaxed DDs.

In Figure 5.2, the horizontal axis represents the number of locations and the vertical axis represents the optimality gap (%). Boxplots are divided into three groups based on the number of locations. Within each

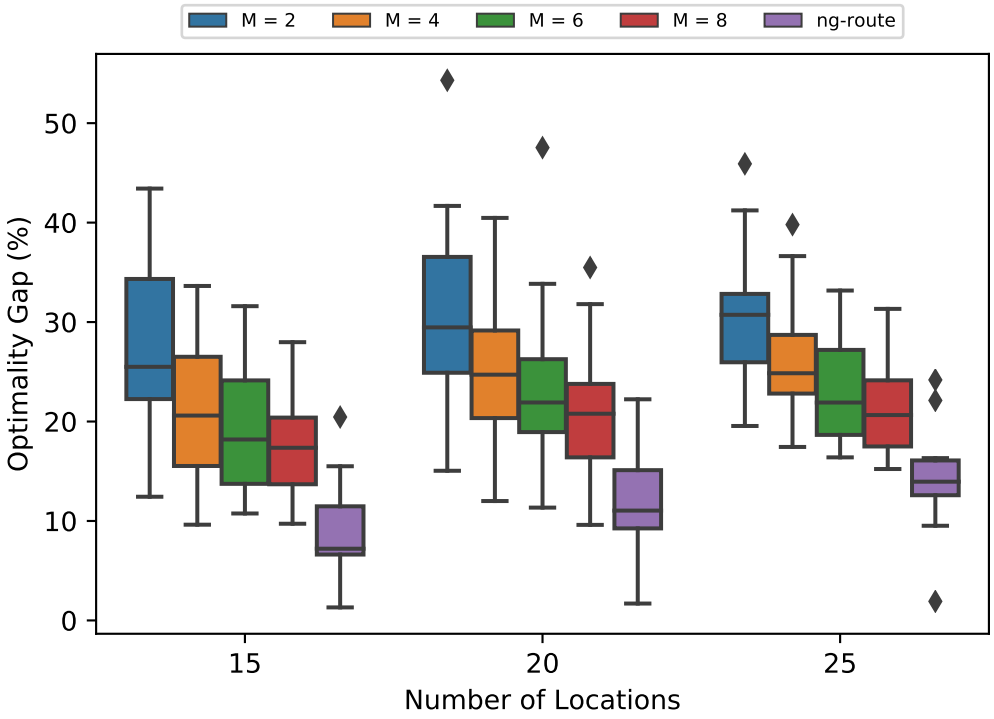


Figure 5.2: Optimality gap (%) of our initial route relaxation and the ng-route relaxation. M denotes the maximum bucket size used in the merging process. CG denotes the column generation approach that solves the ng-route relaxation. We follow the same parameter choice as Roberti and Ruthmair (2019) which set $|N_i| = 5$ for all $i \in N$.

group, we vary bucket size M and compare with the ng-route relaxation. Figure 5.2 shows that the optimality gap from the initial route relaxation typically worsens as the number of locations increases. Furthermore, the optimality gap from the ng-route relaxation is better than that from all our initial route relaxations. This is not surprising because the size of the DD corresponding to the ng-route relaxation is typically much larger than that of our initial relaxations. In other words, the ng-route relaxation, viewed as a relaxed DD, captures more structural information than ours do. Figure 5.2 also shows that, as the maximum bucket size M increases, the quality of lower bounds improves but the marginal improvement decreases. Therefore, we cannot hope to significantly improve lower bound quality by increasing the value of M when compiling our initial route relaxations. Fortunately we can resort to our iterative frameworks. Below we study how our frameworks can improve the initial relaxations.

5.9.2 Lower Bound Improvement

Recall that all of our iterative algorithms in Section 5.5.3 are able to output valid lower bounds at the end of each iteration. We measure bound improvement as follows. For each instance, we take T_{CG}, the

computation time of CG, as a baseline. During the process of an iterative refinement algorithm, we record the best lower bound computed so far every $10\% \cdot T_{\text{CG}}$ seconds. Experiments are run on instances with 15 and 20 locations.

Figure 5.3 shows bound improvement over time for flowRefine, lagAdapt and lagRestart. In the figure, x-axis represents the runtime of each algorithm measured in percentages w.r.t T_{CG} and y-axis represents the average optimality gap (%). As we go from left to right along x-axis, the runtime of each algorithm increases from 0 second to $150\% \cdot T_{\text{CG}}$ seconds, i.e., we overextend the runtime by $50\% \cdot T_{CG}$ seconds to examine the decrease of optimality gap. The gap achieved by ng-route relaxation is shown in a solid line starting from $100\% \cdot T_{\text{CG}}$.

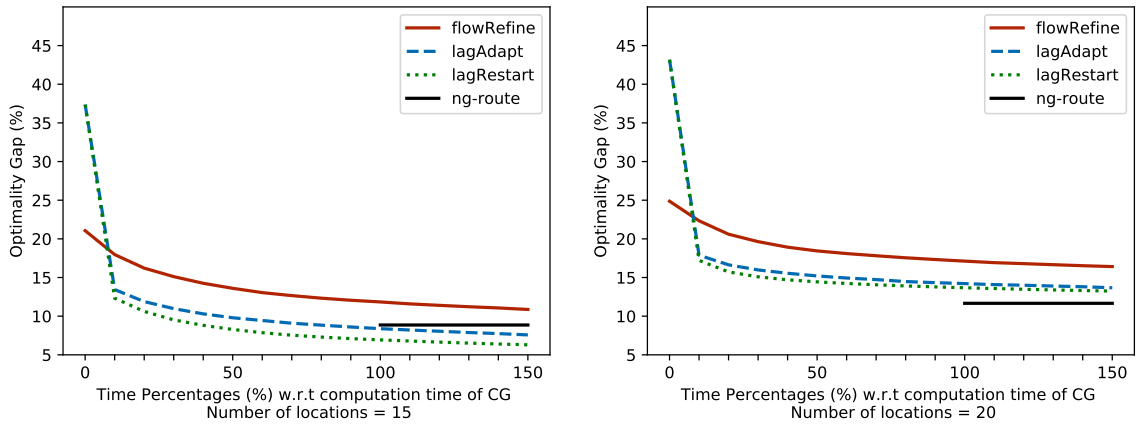


Figure 5.3: Optimality gap (%) over time for flowRefine, lagAdapt and lagRestart. Each interval on the x-axis represent 10% of CG computation time. The optimality gap from the ng-route relaxation appears after 100%.

Figure 5.3 shows that our algorithms are able improve the optimality gap over time. Compared among each other, function lagAdapt and lagRestart are able to outperform function flowRefine with very similar performance for both $m = 15$ and 20. It should be noted that a significant improvement in the optimality gap (the gap decreases by at least 50%) occurs for lagAdapt and lagRestart at $10\% \cdot T_{\text{CG}}$. The initial gap is large because we initialized our multipliers to be all zeros at the start of Lagrangian-based frameworks. Therefore the first lower bound is simply the shortest path length in the initial relaxed DD, which creates a large optimality gap. The sharp decrease in the gap suggests that lagAdapt and lagRestart are very effective in improving the lower bound in the beginning. Indeed, both of them start to outperform flowRefine after $10\% \cdot T_{\text{CG}}$, although the latter starts with a relatively small gap (compared to lagAdapt and lagRefine). It also shows the tapering effect of our iterative algorithms, i.e. the marginal improvement of the optimality gap diminishes over time.

Compared against the ng-route relaxation, when $m = 15$, function lagAdapt achieves better gaps than CG does given 80% of the computation time of CG and function lagRefine achieve better gaps than CG does

given only 40% of the computation time of CG. Function flowRefine gradually improves the gap but is unable to outperform the ng-route relaxation bound given 150% of CG computation time. When $m = 20$, function lagAdapt and lagRefine gradually improve the bound to very close to CG but are not yet able to outperform the ng-route relaxation given 150% of the computation time of CG. Function flowRefine is worse than the other two approaches in this case.

5.9.3 Scalability of Iterative Refinement Algorithms

Next, we study the performance of iterative refinement algorithms when m, the number of locations increases. Below we report our computational results for $m \in \{15, 20, 25, 30, 40, 50\}$ and speed ratio $\alpha \in \{2, 4\}$. Since our implementation of CG does not scale to 30 and the lower bound value found by CP Optimizer is typically not competitive to our algorithms, we need better lower bounds when $m >= 30$. The only other source of lower bounds available in the literature is the MIP model proposed by Roberti and Ruthmair (2019). Their original MIP model uses big-M constraint which results in loose linear relaxations. We replace these constraints with indicator constraints that are adaptively relaxed in a modern MIP solver such as Gurobi (Gurobi Optimization, 2021). This typically leads to a stronger linear relaxation. Detailed on this model can be found in Appendix B. We solve the modified model by Gurobi with parameter MIPFocus set to 3 to force the solver to focus on improving the lower bound. The time limit for this set of experiments is set as follows. When $m \leq 25$, CG can terminate within one hour so we set the time limit of all other algorithm as the computation time of CG. Otherwise, when $m \geq 30$, the computation time for all algorithms is set to one hour.

In Figure 5.4, x-axis indicates the number of customers and y-axis indicates the optimality gap $(\%)$. 'CP' denotes the optimality gap from CP Optimizer, 'ng-route' denotes the optimality gap by solving the ng-route relaxation via column generation and 'MP' denotes the optimality gap from the modified MIP model. Figure 5.4 shows that lagAdapt and lagRestart are competitive to CG when $m \leq 25$. flowRefine is slightly worse. When $m \geq 30$, CG cannot terminate within one hour due to the complexity of the pricing problem. All our algorithms continue to output valid lower bounds. Although the quality deteriorates gradually, lagAdapt and lagRestart still outperform CP and MIP-based lower bounds for all tested instances. Function flowRefine cannot solve the initial CFLP within one hour when $m = 50$ and is thus not shown for this case.

Remark 5.9.1. For $m = 40$, it is observed that only a limited number of iterations (typically fewer than 5) were run by flowRefine due to the large size of the CFLP(D). Indeed, recall its size is proportional to the number of edges and nodes. Although the number of nodes is bounded by $O(Mn^3)$, the number of edges is much larger. Nevertheless, its optimality gap is still competitive to lagAdapt and lagRestart. Therefore, we can obtain stronger bounds if the CFLP can be solved more efficiently. In light of Remark 5.7.1, we may integrate CG with conflict refinements in our future work.

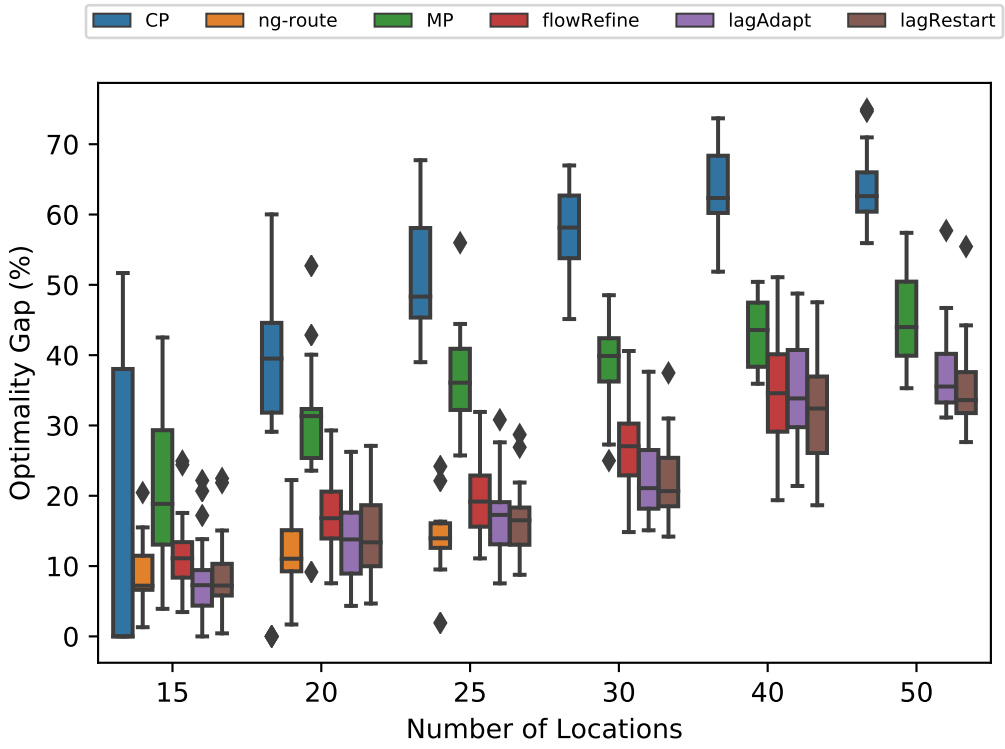


Figure 5.4: Optimality gap (%) from different algorithms, where labels CP, ng-route, MP denote the optimality gap achieved by the constraint programming model, the column generation approach for the ng-route relaxation and the modified MIP model, respectively.

Table 5.2 reports the average size of the ng-route relaxation, and our DD-based route relaxations before and after iterative refinements. For DD-based route relaxations, the size is the number of nodes in the DD; for ng-route, the size is the number of labels after dominance rules are applied. The second row shows the size of initial DDs, which can also be found in Table 5.1. The table shows that our DD size increases after applying iterative algorithms, but is much smaller compared to that of the ng-route relaxation. It is noted that algorithm lagRefine typically produces larger DDs than flowRefine does because solving Lagrangian subproblems is significantly faster than solving the CFLP. As a result, lagRefine typically refines more paths than flowRefine does given the same amount of computation time, and thus producing larger DDs.

Number of locations	15	20	25
initial DD	9736	25086	51485
flowRefine	10725	26538	51967
lagRestart	15567	37197	59204
ng-route	24114	73554	174725

Table 5.2: Average size of route relaxations before and after iterative refinements. For DD-based route relaxations, the size is the number of nodes in the DD; for ng-route, the size is the number of labels after dominance rules are applied. Recall we set the bucket size $M = 4$ for both flowRefine and lagRestart.

5.9.4 Effect of Drone-Truck Speed Ratio

Finally, we study the effect of drone-truck speed ratio on the performance of our algorithms. Computational experience from the literature suggest that the problem becomes easier to solve when the drone-truck speed ratio becomes higher. Figure 5.5, 5.6 and 5.7 shows the optimality gap for flowRefine, lagAdapt and lagRestart respectively, where x-axis represents the number of locations ranging between 15 and 40 and y-axis represents the optimality gap (%). The speed ratio is 2 and 4.

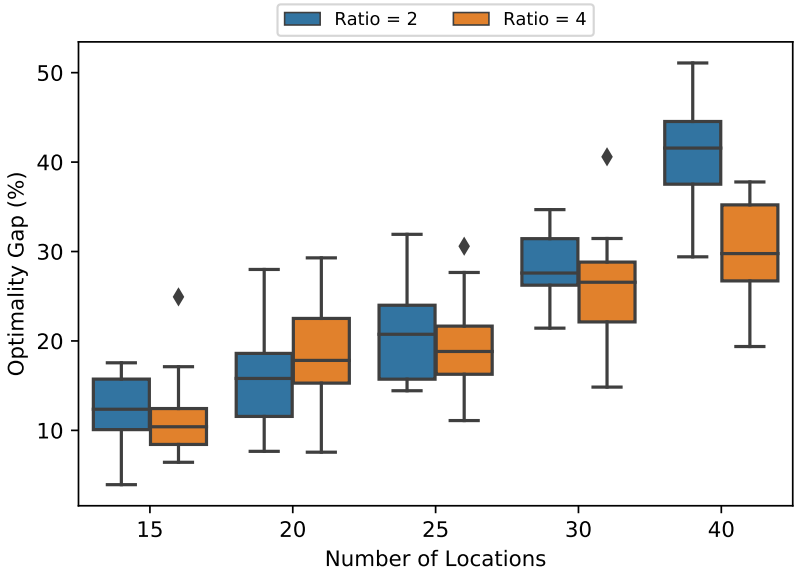


Figure 5.5: Optimality gap from flowRefine w.r.t. the number of locations and drone-truck speed ratio.

Figure 5.5 shows that function flowRefine achieves better gaps when the speed ratio is large for all cases except when $m = 20$. In that case, flowRefine is able to achieve a slightly better gap when the ratio is smaller.

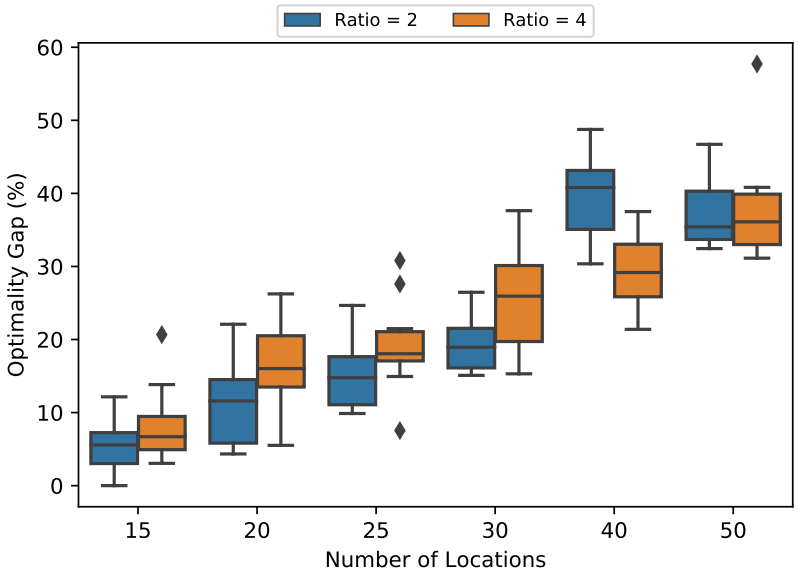


Figure 5.6: Optimality gap from lagAdapt w.r.t. the number of customers and drone-truck speed ratio.

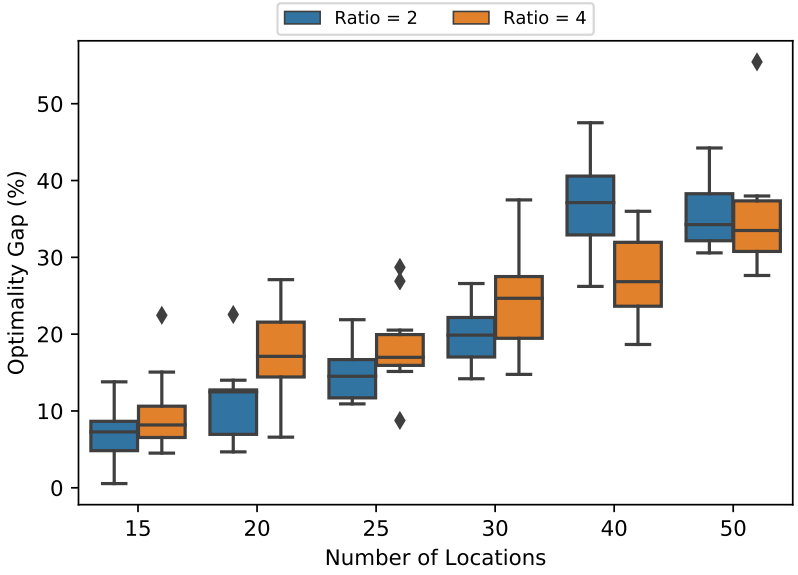


Figure 5.7: Optimality gap from lagRestart w.r.t. the number of customers and truck-drone speed ratio.

Figure 5.6 and 5.7 show that lagAdapt and lagRestart typically perform better when the ratio is smaller except when $m = 40$, which contradicts the computational experience from the literature. For $m \leq 25$, this phenomenon can be partially explained by the setup of our experiments. i.e., for $m \leq 25$, our iterative algorithms have a time limit equal to the computation time of CG, which is typically longer when the speed

ratio is smaller (this is due to the fact that dominance rules for the DP model become less effective when the ratio becomes smaller). As a consequence, our algorithms are run for a longer period of time for cases that are perceived difficult by the DP (and therefore the CG) approach. For $m \geq 30$, however, further investigations are needed.

5.10 Conclusion

In this chapter, we proposed novel route relaxations for the VRPs. Our relaxations are motivated by close connections between DDs and DP models used for pricing in a BCP algorithm. More precisely, we showed there is a one-to-one correspondence between a relaxed DD and a route relaxation. In contrast with relaxations proposed in the literature that forbid repeated visits based on structural information, we construct our initial route relaxations by merging states in a relaxed DD. We further propose two approaches that compute lower bounds from a given relaxed DD without using CG to solve the master problem defined w.r.t. the corresponding route relaxation. We then proved all three approaches are in fact equivalent in the sense that they produce the same lower bound under the same route relaxation. These approaches are integrated with refinement techniques adapted from the DD literature into our iterative frameworks to obtain improved lower bounds. We tested the proposed approaches on the TSP-D, a new and challenging VRP variant. Computational experiments show that, although lower bound values from our initial route relaxations are not competitive to those from the state-of-the-art route relaxation, the proposed iterative frameworks are very effective in improving these bounds to make them competitive or outperform the state-of-the-art approach. When applied to larger instances where the state-of-the-art approach does not scale, our methods are able to generate lower bounds whose values outperform all other existing lower bounding techniques.

We conclude this chapter by outlining future research directions. On the methodological side, we did not implement the integration of conflict refinement with CG, which can be beneficial when solving the CFLP becomes computationally prohibitive, as seen in Section 5.9.1. Better yet, this integration naturally gives a set of paths to refine upon, i.e. those that have non-zero values in an optimal solution to the SPLP. In terms of the integration with Lagrangian relaxation, our algorithms are developed in a rather heuristic fashion by updating Lagrangian multipliers via the subgradient method. Generally speaking, a fundamental question is: how to choose the set of paths to refine upon and how do those refinements and the update of Lagrangian multipliers affect each other. A more systematic approach is needed to improve the current state. Furthermore, our methods can be incorporated into a branch-and-bound framework to compute exact solutions in the following two ways: (a) we can perform branching directly on DDs as described in Bergman et al. (2016b) and (b) we can use any branching scheme based on the set partitioning formulation by either solving the SPLP by CG, or solving the CFLP (Theorem 5.6.1 shows solutions to the two models can be converted to each other). In the second case, the major difference between our methods and traditional BCP algorithms is that we keep and dynamically modify a DD in memory whereas they do not because their route

relaxations are typically much larger. A benefit for keeping the DD in memory is that, at each new node of the branching tree, a lower bound is readily available without solving the resulting master by CG. Indeed, notice that optimal dual variables obtained by solving the SPLP or the CFLP in any node in the branching tree can be used to compute a Lagrangian bound. This bound may be sufficient to perform pruning, which may in turn speed up the entire process.

On the application side, it is interesting to apply our approaches to other well studied VRP variants, such as the capacitated VRP, the VRP with time windows, etc.. DP models for those problems are typically readily available so one of the challenges is to develop merging rules that limit the size of relaxed DDs without the initial bound worsening excessively. Furthermore, one may need to develop problem-specific refinement procedures for different problem classes.

Chapter 6

Conclusions

Supply chain management frequently involves strategic decision-making. This dissertation studied the following three optimization problems arising in the management process: hub network design, inventory routing and traveling salesman problem with drone. Our focus was on combining ideas and techniques developed in computer science and operations research, with the final goal of accommodating large-size real-world instances.

We contributed to the area of network design by introducing a novel logistics network model that captures the trade-off between the cost of hub installation and management, and the reduced cost of routing aggregated flows due to the economy of scales. We developed approximation algorithms for this novel problem as well as its variant, and showed that they are optimal up to constant factors. Using realistic data, we demonstrated our approximation techniques provide very good starting solutions. We further designed an efficient population-based matheuristic that produces solutions with an optimality gap of less than 3% within a reasonable amount of time.

We contributed to the inventory routing problem by designing fast heuristics by solving a family of prize-collecting Steiner tree instances. We showed that our heuristics can find near optimal solutions for instances with or without vehicle capacity in a substantially less amount of time than a mixed-integer programming based approach.

We contributed to the traveling salesman problem with drone by proving a restricted version is still NP-hard, and by formulating it as a compact constraint program model. Computational experiments showed that solving the model by an off-the-shelf solver to optimality is significantly faster than the state-of-the-art exact algorithm at the time of publication of this work. For larger instances up to 60 customers, our CP-based heuristic algorithm is competitive with a state-of-the-art heuristic method in terms of the solution quality.

As our final contribution, we propose several iterative algorithms to compute lower bounds for VRPs motivated by connections between decision diagrams (DDs) and dynamic programming (DP) models used for pricing in a branch-and-cut-and-price algorithm. Our approaches are general and can be applied to various vehicle routing problems where corresponding DP models are available. By adapting merging and refinement techniques from the DD literature, we are able to generate and strengthen novel route relaxations. We also propose alternative approaches to derive lower bounds from DD-based route relaxations which use a flow model with side constraints or Lagrangian relaxation in place of column generation. All the techniques are then integrated into iterative frameworks to obtain improved lower bounds. When applied to the TSP-D, our algorithms are able to produce lower bounds whose values are competitive to those from the state-of-the-art approach. Applied to larger problem instances where the state-of-the-art approach does not scale, our methods are shown to outperform all other existing lower bounding techniques. Finally, we discussed future research directions for solving vehicle routing problems with DD-based route relaxations.

Bibliography

Adulyasak, Y., Cordeau, J.-F., and Jans, R. Formulations and branch-and-cut algorithms for multivehicle production and inventory routing problems. *INFORMS Journal on Computing*, 26(1):103–120, 2014.

Agatz, N., Bouman, P., and Schmidt, M. Optimization approaches for the traveling salesman problem with drone. *Transportation Science*, 52(4):965–981, 2018.

Akers, S. B. Binary decision diagrams. *IEEE Computer Architecture Letters*, 27(06):509–516, 1978.

Aksen, D., Kaya, O., Salman, F. S., and Tüncel, O. An adaptive large neighborhood search algorithm for a selective and periodic inventory routing problem. *European Journal of Operational Research*, 239(2): 413–426, 2014.

Alumur, S. and Kara, B. Y. Network hub location problems: The state of the art. *European journal of operational research*, 190(1):1–21, 2008.

Andersen, H. R., Hadzic, T., Hooker, J. N., and Tiedemann, P. A constraint store based on multivalued decision diagrams. In *International Conference on Principles and Practice of Constraint Programming*, pages 118–132. Springer, 2007.

Andersson, H., Hoff, A., Christiansen, M., Hasle, G., and Løkketangen, A. Industrial aspects and literature survey: Combined inventory management and routing. *Computers & Operations Research*, 37(9):1515–1536, 2010.

Ando, R. and Matsui, T. Algorithm for single allocation problem on hub-and-spoke networks in 2-dimensional plane. *Algorithms and Computation*, pages 474–483, 2011.

Andrews, M. Hardness of buy-at-bulk network design. In *Foundations of Computer Science, 2004. Proceedings. 45th Annual IEEE Symposium on*, pages 115–124. IEEE, 2004.

Andrews, M. and Zhang, L. The access network design problem. In *Foundations of Computer Science, 1998. Proceedings. 39th Annual Symposium on*, pages 40–49. IEEE, 1998.

Archetti, C., Bertazzi, L., Laporte, G., and Speranza, M. G. A branch-and-cut algorithm for a vendor-managed inventory routing problem. *Transportation Science*, 41(3):382–391, 2007.

Archetti, C., Bertazzi, L., Hertz, A., and Speranza, M. G. A hybrid heuristic for an inventory routing problem. *INFORMS Journal on Computing*, 24(1):101–116, 2012.

Archetti, C., Boland, N., and Speranza, M. G. A matheuristic for the multivehicle inventory routing problem. *INFORMS Journal on Computing*, 29(3):377–387, 2017.

Archetti, C., Bianchessi, N., Irnich, S., and Speranza, M. G. Formulations for an inventory routing problem. *International Transactions in Operational Research*, 21(3):353–374, 2014.

Arkin, E., Joneja, D., and Roundy, R. Computational complexity of uncapacitated multi-echelon production planning problems. *Operations Research Letters*, 8:61–66, 1989.

Avella, P., Boccia, M., and Wolsey, L. A. Single-period cutting planes for inventory routing problems. *Transportation Science*, 52(3):497–508, 2018.

Awerbuch, B. and Azar, Y. Buy-at-bulk network design. In *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, pages 542–547. IEEE, 1997.

Balakrishnan, A., Magnanti, T. L., and Mirchandani, P. Modeling and heuristic worst-case performance analysis of the two-level network design problem. *Management Science*, 40(7):846–867, 1994.

Baldacci, R., Mingozzi, A., and Roberti, R. New route relaxation and pricing strategies for the vehicle routing problem. *Operations research*, 59(5):1269–1283, 2011.

Becker, B., Behle, M., Eisenbrand, F., and Wimmer, R. Bdds in a branch and cut framework. In *International Workshop on Experimental and Efficient Algorithms*, pages 452–463. Springer, 2005.

Bentert, M., van Bevern, R., Nichterlein, A., Niedermeier, R., and Smirnov, P. V. Parameterized algorithms for power-efficiently connecting sensor networks: Theory and experiments. *INFORMS Journal on Computing, to appear*, 2017.

Berger, B., Peng, J., and Singh, M. Computational solutions for omics data. *Nat Rev Genet.*, 14(5):333–346, 2013.

Bergman, D., Van Hoeve, W.-J., and Hooker, J. N. Manipulating mdd relaxations for combinatorial optimization. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pages 20–35. Springer, 2011.

Bergman, D., Cire, A. A., Van Hoeve, W.-J., and Hooker, J. N. Variable ordering for the application of bdds to the maximum independent set problem. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 34–49. Springer, 2012.

Bergman, D., Cire, A. A., Sabharwal, A., Samulowitz, H., Saraswat, V., and van Hoeve, W.-J. Parallel combinatorial optimization with decision diagrams. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pages 351–367. Springer, 2014a.

Bergman, D., Cire, A. A., van Hoeve, W.-J., and Yunes, T. Bdd-based heuristics for binary optimization. *Journal of Heuristics*, 20(2):211–234, 2014b.

Bergman, D., Cire, A. A., and van Hoeve, W.-J. Lagrangian bounds from decision diagrams. *Constraints*, 20(3):346–361, 2015.

Bergman, D., Cire, A. A., Van Hoeve, W.-J., and Hooker, J. *Decision diagrams for optimization*, volume 1. Springer, 2016a.

Bergman, D., Cire, A. A., Van Hoeve, W.-J., and Hooker, J. N. Discrete optimization with decision diagrams. *INFORMS Journal on Computing*, 28(1):47–66, 2016b.

Bienkowski, M., Byrka, J., Chrobak, M., Jeż, L., Nogneng, D., and Sgall, J. Better approximation bounds for the joint replenishment problem. In *Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 42–54, 2014.

Bienstock, D., Goemans, M. X., Simchi-Levi, D., and Williamson, D. A note on the prize collecting traveling salesman problem. *Mathematical programming*, 59(1-3):413–420, 1993.

Bouman, P., Agatz, N., and Schmidt, M. Dynamic programming approaches for the traveling salesman problem with drone. *Networks*, 72(4):528–542, 2018.

Boyd, S., Xiao, L., and Mutapcic, A. Subgradient methods. *lecture notes of EE392o, Stanford University, Autumn Quarter*, 2004:2004–2005, 2003.

Burns, L., Hall, R., Blumenfeld, D., and Daganzo, C. Distribution strategies that minimize transportation and inventory costs. *Operations Research*, 33(3):469–490, 1985.

Byrka, J., Grandoni, F., Rothvoß, T., and Sanità, L. Steiner tree approximation via iterative randomized rounding. *J. ACM*, 60(1):6:1–6:33, 2013.

Campbell, A. and Savelsbergh, M. *Inventory Routing in Practice*. SIAM Monographs on Discrete Mathematics and Applications, 2002.

Campbell, A., Savelsbergh, M., Clarke, L., and Kleywegt, A. *The Inventory Routing Problem*. Springer US, 1998.

Carlsson, J. G. and Song, S. Coordinated logistics with a truck and a drone. *Management Science*, 64(9): 4052–4069, 2018.

Carpenter, T. and Luss, H. Telecommunications access network design. In *Handbook of optimization in telecommunications*, pages 313–339. Springer, 2006.

Castro, M. P., Cire, A. A., and Beck, J. C. An mdd-based lagrangian approach to the multicommodity pickup-and-delivery tsp. *INFORMS Journal on Computing*, 32(2):263–278, 2020.

Chan, L., Fedgergruen, A., and Simchi-Levi, D. Probabilistic analyses and practical algorithms for inventory-routing models. *Operations Research*, 46(1):96–106, 1998.

Charikar, M. and Karagiozova, A. On non-uniform multicommodity buy-at-bulk network design. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 176–182. ACM, 2005.

Chechik, S. and Wulff-Nilsen, C. Near-optimal light spanners. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 883–892. Society for Industrial and Applied Mathematics, 2016.

Chekuri, C., Hajiaghayi, M. T., Kortsarz, G., and Salavatipour, M. R. Approximation algorithms for nonuniform buy-at-bulk network design. *SIAM Journal on Computing*, 39(5):1772–1798, 2010.

Chien, T. W., Balakrishnan, A., and Wong, R. T. An integrated inventory allocation and vehicle routing problem. *Transportation science*, 23(2):67–76, 1989.

Christofides, N., Mingozzi, A., and Toth, P. Exact algorithms for the vehicle routing problem, based on spanning tree and shortest path relaxations. *Mathematical programming*, 20(1):255–282, 1981.

Chung, S. H., Sah, B., and Lee, J. Optimization for drone and drone-truck combined operations: A review of the state of the art and future directions. *Computers & Operations Research*, page 105004, 2020.

Cire, A. A. and Van Hoeve, W.-J. Multivalued decision diagrams for sequencing problems. *Operations Research*, 61(6):1411–1428, 2013.

Coelho, L. C., Cordeau, J.-F., and Laporte, G. The inventory-routing problem with transshipment. *Comput. Oper. Res.*, 39(11):2537–2548, 2012a.

Coelho, L. C., Cordeau, J.-F., and Laporte, G. Consistency in multi-vehicle inventory-routing. *Transportation Res. Part C: Emerging Tech.*, 24(1):270–287, 2012b.

Contreras, I. and Fernández, E. General network design: A unified view of combined location and network design problems. *European Journal of Operational Research*, 219(3):680–697, 2012.

Cook, W. *Concorde TSP Solver*, 2015. http://www.math.uwaterloo.ca/tsp/concorde/index.html.

Costa, A. M. A survey on benders decomposition applied to fixed-charge network design problems. *Computers & operations research*, 32(6):1429–1450, 2005.

Costa, L., Contardo, C., and Desaulniers, G. Exact branch-price-and-cut algorithms for vehicle routing. *Transportation Science*, 53(4):946–985, 2019.

de Freitas, J. C. and Penna, P. H. V. A variable neighborhood search for flying sidekick traveling salesman problem. *International Transactions in Operational Research*, 27(1):267–290, 2020.

Desaulniers, G., Rakke, J., and Coelho, L. A branch-price-and-cut algorithm for the inventory-routing problem. *Transportation Science*, 50(3):1060–1076, 2016.

Desaulniers, G., Lessard, F., and Hadjar, A. Tabu search, partial elementarity, and generalized k-path inequalities for the vehicle routing problem with time windows. *Transportation Science*, 42(3):387–404, 2008.

Desrochers, M., Desrosiers, J., and Solomon, M. A new optimization algorithm for the vehicle routing problem with time windows. *Operations research*, 40(2):342–354, 1992.

Dinur, I. and Steurer, D. Analytical approach to parallel repetition. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 624–633. ACM, 2014.

Dorling, K., Heinrichs, J., Messier, G. G., and Magierowski, S. Vehicle routing problems for drone delivery. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 47(1):70–85, 2017.

Dror, M. Note on the complexity of the shortest path models for column generation in vrptw. *Operations Research*, 42(5):977–978, 1994.

Elkin, M., Neiman, O., and Solomon, S. Light spanners. *SIAM Journal on Discrete Mathematics*, 29(3): 1312–1321, 2015.

Erzin, A., Plotnikov, R., and Shamardin, Y. V. On some polynomially solvable cases and approximate algorithms in the optimal communication tree construction problem. *Journal of Applied and Industrial Mathematics*, 7(2):142–152, 2013.

Farahani, R. Z., Hekmatfar, M., Arabani, A. B., and Nikbakhsh, E. Hub location problems: A review of models, classification, solution techniques, and applications. *Computers & Industrial Engineering*, 64(4): 1096–1109, 2013.

Ferrandez, S. M., Harbison, T., Weber, T., Sturges, R., and Rich, R. Optimization of a truck-drone in tandem delivery network using k-means and genetic algorithm. *Journal of Industrial Engineering and Management (JIEM)*, 9(2):374–388, 2016.

Fischetti, M., Leitner, M., Ljubić, I., Luipersbeck, M., Monaci, M., Resch, M., Salvagnin, D., and Sinnl, M. Thinning out steiner trees: A node-based model for uniform edge costs. *Mathematical Programming Computation*, 9(2):203–229, 2017.

Fisher, M. L. The lagrangian relaxation method for solving integer programming problems. *Management science*, 27(1):1–18, 1981.

Fukasawa, R., Longo, H., Lysgaard, J., De Aragão, M. P., Reis, M., Uchoa, E., and Werneck, R. F. Robust branch-and-cut-and-price for the capacitated vehicle routing problem. *Mathematical programming*, 106 (3):491–511, 2006.

Fukunaga, T., Nikzad, A., and Ravi, R. Deliver or hold: Approxmation algorithms for the periodic inventory routing problem. In *Proceedings of the 17th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems*, pages 209–225, 2014.

Geoffrion, A. M. Lagrangean relaxation for integer programming. In *Approaches to integer programming*, pages 82–114. Springer, 1974.

Goel, V., Furman, K. C., Song, J., and El-Bakry, A. S. Large neighborhood search for lng inventory routing. *Journal of Heuristics*, 18(6):821–848, 2012.

Goemans, M. and Williamson, D. A general approximation technique for constrained forest problems. *SIAM Journal on Computing*, 24(2):296–317, 1995.

Goemans, M. X., Olver, N., Rothvoß, T., and Zenklusen, R. Matroids and integrality gaps for hypergraphic steiner tree relaxations. In Karloff, H. J. and Pitassi, T., editors, *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, May 19 - 22, 2012*, pages 1161–1176. ACM, 2012.

Guha, S., Meyerson, A., and Munagala, K. A constant factor approximation for the single sink edge installation problem. *SIAM Journal on Computing*, 38(6):2426–2442, 2009.

Gupta, A. and Könemann, J. Approximation algorithms for network design: A survey. *Surveys in Operations Research and Management Science*, 16(1):3–20, 2011.

Gupta, A., Kumar, A., and Roughgarden, T. Simpler and better approximation algorithms for network design. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 365–372. ACM, 2003.

Gurobi Optimization, L. Gurobi optimizer reference manual, 2021. URL http://www.gurobi.com.

Ha, Q. M., Deville, Y., Pham, Q. D., and Hà, M. H. On the min-cost traveling salesman problem with drone. *Transportation Research Part C: Emerging Technologies*, 86:597–621, 2018.

Ha, Q. M., Deville, Y., Pham, Q. D., and Hà, M. H. A hybrid genetic algorithm for the traveling salesman problem with drone. *Journal of Heuristics*, 26(2):219–247, 2020.

Hegde, C., Indyk, P., and Schmidt, L. A nearly-linear time framework for graph-structured sparsity. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, pages 4165–4169, 2016.

Held, M., Wolfe, P., and Crowder, H. P. Validation of subgradient optimization. *Mathematical programming*, 6(1):62–88, 1974.

Hooker, J. N. Planning and scheduling by logic-based benders decomposition. *Operations Research*, 55(3): 588–602, 2007.

Hooker, J. N. Improved job sequencing bounds from decision diagrams. In *International Conference on Principles and Practice of Constraint Programming*, pages 268–283. Springer, 2019a.

Hooker, J. N. Logic-based benders decomposition for large-scale optimization. In *Large Scale Optimization in Supply Chains and Smart Manufacturing*, pages 1–26. Springer, 2019b.

Hooker, J. N. and Ottosson, G. Logic-based benders decomposition. *Mathematical Programming*, 96(1): 33–60, 2003.

IBM ILOG CPLEX. CP Optimizer 12.7 User's Manual, 2017.

Irnich, S. and Villeneuve, D. The shortest-path problem with resource constraints and k-cycle elimination for k $\geq$ 3. *INFORMS Journal on Computing*, 18(3):391–406, 2006.

Iwasa, M., Saito, H., and Matsui, T. Approximation algorithms for the single allocation problem in hub-and-spoke networks and related metric labeling problems. *Discrete Applied Mathematics*, 157(9):2078–2088, 2009.

Jeon, I., Ham, S., Cheon, J., Klimkowska, A. M., Kim, H., Choi, K., and Lee, I. A real-time drone mapping platform for marine surveillance. *International Archives of the Photogrammetry, Remote Sensing & Spatial Information Sciences*, 2019.

Kawahara, Y., Nagano, K., Tsuda, K., and Bilmes, J. A. Submodularity cuts and applications. In *Advances in Neural Information Processing Systems*, pages 916–924, 2009.

Khurana, V., Peng, J., Chung, C. Y., Auluck, P. K., Fanning, S., Tardiff, D. F., Bartels, T., Koeva, M., Eichhorn, S. W., Benyamini, H., Lou, Y., er Upham, A. N., Baru, V., Freyzon, Y., Tuncbag, N., Costanzo, M., Luis, B.-J. S., Schöndorf, D. C., Barrasa, M. I., Ehsani, S., Sanjana, N. E., Zhong, Q., Gasser, T. P. D., Bartel, D. P., Vidal, M., Deleidi, M., Boone, C., Fraenkel, E., Berger, B., and Lindquist, S. L. Genome-scale networks link neurodegenerative disease genes to α-synuclein through specific molecular pathways. *Cell systems*, 4:157–170, 2017.

Kinable, J., Cire, A. A., and van Hoeve, W.-J. Hybrid optimization methods for time-dependent sequencing problems. *European Journal of Operational Research*, 259(3):887–897, 2017.

Klein, P. and Ravi, R. A nearly best-possible approximation algorithm for node-weighted steiner trees. *Journal of Algorithms*, 19(1):104 – 115, 1995a. ISSN 0196-6774.

Klein, P. N. and Ravi, R. A nearly best-possible approximation algorithm for node-weighted steiner trees. *J. Algorithms*, 19(1):104–114, 1995b.

Laborie, P. IBM ILOG CP Optimizer for Detailed Scheduling Illustrated on Three Problems. In *Proceedings of CPAIOR*, volume 5547 of *LNCS*, pages 148–162. Springer, 2009.

Laborie, P., Rogerie, J., Shaw, P., and Vilím, P. IBM ILOG CP optimizer for scheduling. *Constraints*, 23 (2):210–250, 2018.

Lee, C.-Y. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal*, 38(4):985–999, 1959.

Leitner, M., Ljubić, I., Luipersbeck, M., and Sinnl, M. A dual-ascent-based branch-and-bound framework for the prize-collecting steiner tree and related problems. 2016.

Leitner, M., Ljubić, I., Luipersbeck, M., and Sinnl, M. A dual-ascent-based branch-and-bound framework for the prize-collecting steiner tree and related problems. *INFORMS Journal on Computing*, 30(2):217–420, 2020.

Levi, R., Roundy, R., and Shmoys, D. Primal-dual algorithms for deterministic inventory problems. *Mathematics of Operations Research*, 31(2):267–284, 2006.

Levi, R., Roundy, R., Shmoys, D., and Sviridenko, M. First constant approximation algorithm for the one-warehouse multi-retailer problem. *Management Science*, 54(4):763–776, 2008.

Li, S. A 1.488 approximation algorithm for the uncapacitated facility location problem. *Automata, languages and programming*, pages 77–88, 2011.

Ljubić, I., Weiskircher, R., Pferschy, U., Klau, G. W., Mutzel, P., and Fischetti, M. An algorithmic framework for the exact solution of the prize-collecting steiner tree problem. *Mathematical Programming*, 105 (2-3):427–449, 2006.

Macrina, G., Pugliese, L. D. P., Guerriero, F., and Laporte, G. Drone-aided routing: A literature review. *Transportation Research Part C: Emerging Technologies*, 120:102762, 2020.

Mansour, Y. and Peleg, D. *An approximation algorithm for minimum-cost network design*. Weizmann Institute of Science. Department of Applied Mathematics and Computer Science, 1994.

Melo, M. T., Nickel, S., and Saldanha-Da-Gama, F. Facility location and supply chain management–a review. *European journal of operational research*, 196(2):401–412, 2009.

Mentzer, J. T., DeWitt, W., Keebler, J. S., Min, S., Nix, N. W., Smith, C. D., and Zacharia, Z. G. Defining supply chain management. *Journal of Business logistics*, 22(2):1–25, 2001.

Min, H. and Zhou, G. Supply chain modeling: past, present and future. *Computers & industrial engineering*, 43(1-2):231–249, 2002.

Min, S., Zacharia, Z. G., and Smith, C. D. Defining supply chain management: in the past, present, and future. *Journal of Business Logistics*, 40(1):44–55, 2019.

Mula, J., Peidro, D., Díaz-Madroñero, M., and Vicens, E. Mathematical programming models for supply chain production and transport planning. *European Journal of Operational Research*, 204(3):377–390, 2010.

Murray, C. C. and Chu, A. G. The flying sidekick traveling salesman problem: Optimization of drone-assisted parcel delivery. *Transportation Research Part C: Emerging Technologies*, 54:86–109, 2015.

Nagarajan, V. and Shi, C. Approximation algorithms for inventory problems with submodular or routing costs. *Mathematical Programming*, pages 1–20, 2016.

Nagy, G. and Salhi, S. Location-routing: Issues, models and methods. *European Journal of Operational Research*, 177(2):649–672, 2007.

Nonner, T. and Souza, A. Approximating the joint replenishment problem with deadlines. *Discrete Math., Alg. and Appl.*, 1(2):153–174, 2009.

Otto, A., Agatz, N., Campbell, J., Golden, B., and Pesch, E. Optimization approaches for civil applications of unmanned aerial vehicles (uavs) or aerial drones: A survey. *Networks*, 72(4):411–458, 2018.

O'Neil, R. J. and Hoffman, K. Decision diagrams for solving traveling salesman problems with pickup and delivery in real time. *Operations Research Letters*, 47(3):197–201, 2019.

Poikonen, S., Wang, X., and Golden, B. The vehicle routing problem with drones: Extended models and connections. *Networks*, 70(1):34–43, 2017.

Poikonen, S., Golden, B., and Wasil, E. A. A branch-and-bound approach to the traveling salesman problem with a drone. *INFORMS Journal on Computing*, 31(2):335–346, 2019.

Ponza, A. Optimization of drone-assisted parcel delivery. 2016.

Prodhon, C. and Prins, C. A survey of recent research on location-routing problems. *European Journal of Operational Research*, 238(1):1–17, 2014.

Raidl, G. R. and Puchinger, J. Combining (integer) linear programming techniques and metaheuristics for combinatorial optimization. *Hybrid metaheuristics*, 114:31–62, 2008.

Ravi, R. and Salman, F. S. Approximation algorithms for the traveling purchaser problem and its variants in network design. In *European Symposium on Algorithms*, pages 29–40. Springer, 1999.

Roberti, R. and Ruthmair, M. Exact methods for the traveling salesman problem with drone. *Optimization Online*, 2019.

Rothberg, E. An evolutionary algorithm for polishing mixed integer programming solutions. *INFORMS Journal on Computing*, 19(4):534–541, 2007.

Rothlauf, F. *Design of modern heuristics: principles and application*. Springer Science & Business Media, 2011.

Salama, M. and Srinivas, S. Joint optimization of customer location clustering and drone-based routing for last-mile deliveries. *Transportation Research Part C: Emerging Technologies*, 114:620–642, 2020.

Salman, F. S., Cheriyan, J., Ravi, R., and Subramanian, S. Approximating the single-sink link-installation problem in network design. *SIAM Journal on Optimization*, 11(3):595–610, 2001.

Schmidt, L., Hegde, C., Indyk, P., Lu, L., Chi, X., and Hohl, D. Seismic feature extraction using steiner tree methods. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1647–1651, 2015.

Shapiro, J. F. A survey of lagrangean techniques for discrete optimization. In *Annals of Discrete Mathematics*, volume 5, pages 113–138. Elsevier, 1979.

Shen, Z.-J. M. and Qi, L. Incorporating inventory and routing costs in strategic location models. *European journal of operational research*, 179(2):372–389, 2007.

Shirokikh, V. A. and Zakharov, V. V. Dynamic adaptive large neighborhood search for inventory routing problem. *Modelling, Computation and Optimization in Information Systems and Management Sciences*, 359:231–241, 2015.

Silver, E. A. Operations research in inventory management: A review and critique. *Operations Research*, 29(4):628–645, 1981.

Sohn, J. and Park, S. A linear program for the two-hub location problem. *European Journal of Operational Research*, 100(3):617–622, 1997.

Sohn, J. and Park, S. The single allocation problem in the interacting three-hub network. *Networks*, 35(1): 17–25, 2000.

Talbi, E.-G. *Metaheuristics: from design to implementation*, volume 74. John Wiley & Sons, 2009.

Tang, Z., Jiao, Y., and Ravi, R. Combinatorial heuristics for inventory routing problems. *INFORMS Journal on Computing*, to appear.

Tang, Z., van Hoeve, W.-J., and Shaw, P. A study on the traveling salesman problem with a drone. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 557–564. Springer, 2019.

Tjandraatmadja, C. and van Hoeve, W.-J. Target cuts from relaxed decision diagrams. *INFORMS Journal on Computing*, 31(2):285–301, 2019.

Tjandraatmadja, C. and van Hoeve, W.-J. Incorporating bounds from decision diagrams into integer programming. *Mathematical Programming Computation*, pages 1–32, 2020.

Toth, P. and Vigo, D. *Vehicle routing: problems, methods, and applications*. SIAM, 2014.

Tuncbag, N., Braunstein, A., Pagnani, A., Huang, S.-S. C., Chayes, J., Borgs, C., Zecchina, R., and Fraenkel, E. Simultaneous reconstruction of multiple signaling pathways via the prize-collecting steiner forest problem. *J Comput Biol.*, 20(2):124–136, 2013.

UPS. UPS Tests Residential Delivery Via Drone. Youtube, 2017. URL https://www.youtube.com/watch?v=xx9_6OyjJrQ.

van Hoeve, W.-J. Graph coloring lower bounds from decision diagrams. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 405–418. Springer, 2020.

Vásquez, S. A., Angulo, G., and Klapp, M. A. An exact solution method for the tsp with drone based on decomposition. *Computers & Operations Research*, 127:105127, 2021.

Vazirani, V. V. *Approximation algorithms*. Springer Science & Business Media, 2013.

Wang, X., Poikonen, S., and Golden, B. The vehicle routing problem with drones: Several worst-case results. *Optimization Letters*, 11(4):679–697, 2017.

Wegener, I. *Branching programs and binary decision diagrams: theory and applications*. SIAM, 2000.

Williamson, D. P. and Shmoys, D. B. *The design of approximation algorithms*. Cambridge university press, 2011.

Yu, Y., Chen, H., and Chu, F. A new model and hybrid approach for large scale inventory routing problems. *European Journal of Operational Research*, 189(3):1022–1040, 2008.

Yurek, E. E. and Ozmutlu, H. C. A decomposition-based iterative optimization algorithm for traveling salesman problem with drone. *Transportation Research Part C: Emerging Technologies*, 91:249–262, 2018.

Appendices

Appendix A

Appendix for Chapter 3

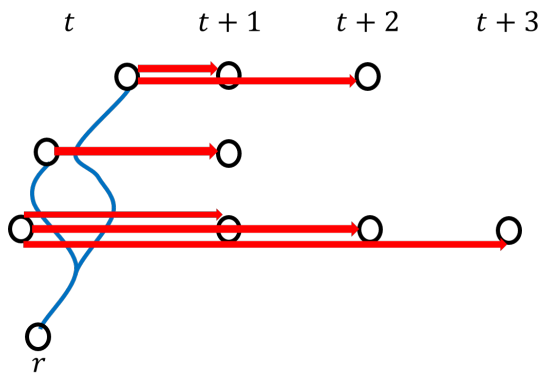


Figure A.1: To apply the greedy algorithm for set cover to IRP, we define a set in IRP to be a subset of demands. The way that a set is served is determined by three choices: a day t of service, a subset of stores to visit on day t which induces a minimum cost tree T spanning the subset, and a subset $D(T)$ of demands with deadlines no earlier than t. The routing cost of this set is cost of the blue tree. The holding cost of this set is the holding cost to serve $D(T)$ from day t, represented by the red segments.

A.1 Greedy Heuristic

In this section, we introduce a greedy heuristic for IRP. Section A.1.1 adapts the greedy framework of set cover to IRP, where a minimum density set is repeatedly chosen to cover some subset of demands. The search space for a minimum density set involves an exponential number of subsets of vertices. To simplify the choices needed to pick the set, we instead will show how to find a set whose density at most 3 times the minimum density value in Section A.1.2. We prove that picking the approximately minimum density as the greedy step achieves a logarithmic approximation factor for IRP. However, this greedy step is still computationally expensive (even though it is in polynomial time). So the implementation will modify the algorithm to repeatedly pick any set whose density is within a certain specified threshold and raise that threshold whenever no more such sets exists. The details of the implementation are described in A.1.3.

A.1.1 Greedy Framework

The greedy algorithm will attempt to cover the demands with routes choosing a route that minimizes the ratio of the coverage cost to the number of newly covered demands.

As before, T_t denotes the existing tree on day t. Let D be the set of uncovered demands. Let r be the routing cost function, h the holding cost function. For $D' \subset D$, define $d(D') = \sum_{(v,t)\in D'} d_t^v$. The *density* of a tree T and coverage set $D(T)$ of demands is $\rho(T, D(T)) := \frac{r(T)+h(D(T))}{d(D(T))}$.

The greedy algorithm is as follows.

Figure A.1 illustrates how sets of demands are chosen to be covered by visits. Instead of finding the exact

Algorithm 14: Greedy Framework

Initialize $T_t \leftarrow \varnothing \forall t \in [1, T]$ and $D \leftarrow D(V \times [T])$
while $|D| > 0$ **do**
  Find a day t, tree T on day t, and coverage set $D(T) \subset D$ minimizing $\rho(T, D(T))$
  $D \leftarrow D \setminus D(T)$
  $T_t \leftarrow T_t \cup T$

minimum density tree and coverage set, we will find those of density at most 3 times the minimum density by solving a PCST whose penalties will represent the best total value of new demands to cover.

A.1.2 Approximate Minimum Density Set

Formally, given a time t that we attempt to add client v to and target density value ρ, define the *coverage number* $\eta(v, t, \rho)$ to be the maximum number of consecutive (with respect to the timeframe) uncovered demand points D' at client v day within days $[t, T]$ such that the weighted average holding cost $\frac{h(D')}{d(D')}$ to serve all such demands is at most ρ. Let $A(v, t, \rho)$ and $h(v, t, \rho)$ be the total demand and total holding cost, respectively, corresponding to the $\eta(v, t, \rho)$ many uncovered demand points whose weighted average holding cost stays within ρ. We use the Algorithm 15 to approximate the minimum density tree and coverage set.

Algorithm 15: Approximately Minimum Density Set

Guess the best day t^* to add a minimum density tree and density value ρ^* of the best coverage set
Find the classical primal dual solution (Goemans and Williamson, 1995) to the PCST with edge costs and penalties

$$w(e) := \begin{cases} c(e) & \text{if } e \notin E(T_{t^*}) \\ 0 & \text{else} \end{cases} \tag{A.1}$$

$$\pi(v) := \begin{cases} A(v, t^*, \rho^*)\rho^* & \text{if } v \notin V(T_{t^*}) \\ 0 & \text{else.} \end{cases} \tag{A.2}$$

Return the PCST tree $PCST_{t^*}$ and coverage set $\cup_{v \in V(T_{PCST}) \setminus V(T_{t^*})} D_v$ where D_v is the set of $\eta(v, t^*, \rho^*)$ uncovered demands with demand time closest to t^* (starting from t^*)

Next, we show that the above procedure approximates the minimum density set within a factor of 3. For the analysis, we provide the dual LP used to construct the primal dual solution for PCST (Goemans and Williamson, 1995).

$$\min \sum_{e \in E} c_e x_e + \sum_{X \subset V \setminus \{r\}} \pi(X) z_X \tag{A.3}$$

$$\text{s.t.} \sum_{e \in \delta(S)} x_e + \sum_{X : X \supset S} z_X \geq 1 \qquad \forall S \subset V \setminus \{r\} \tag{A.4}$$

$$x_e \geq 0 \qquad \forall e \in E \tag{A.5}$$

$$z_X \geq 0 \qquad \forall X \subset V \setminus \{r\} \tag{A.6}$$

$$\max \sum_{S \subset V \setminus \{r\}} y_S$$

$$\text{s.t.} \sum_{S : e \in \delta(S), S \not\ni r} y_S \leq w_e \qquad \forall e \in E \tag{A.7}$$

$$\sum_{S : S \subset X} y_S \leq \pi(X) \qquad \forall X \subset V \setminus \{r\} \tag{A.8}$$

$$y_S \geq 0 \qquad \forall S \subset V \setminus \{r\} \tag{A.9}$$

We can bound the routing cost with respect to the dual values using the same analysis as (Goemans and Williamson, 1995), except that we bound the cost with respect to $\sum_{S \subset V(T)} y_S$ instead of their $\sum_{S \subset V \setminus \{r\}} y_S$. Let y denote the dual solution defined by the algorithm in selecting the tree T. The following lemma is implicit in (Goemans and Williamson, 1995).

Lemma A.1.1. $r(T) \leq 2 \sum_{S \subset V(T)} y_S$.

Lemma A.1.2. *Let T and $D(T)$ be the tree and coverage set returned by the above PCST algorithm. Then $\rho(T, D(T)) \leq 3\rho^*$.*

Proof. First, we compute the routing cost of T. We have

$$\begin{aligned} r(T) \leq & 2 \sum_{S \subset V(T)} y_S \text{ by Lemma A.1.1} \\ \leq & 2\pi(V(T)) \text{ by the dual constraints} \\ \leq & 2 \sum_{v \in V(T)} A(v, t^*, \rho^*)\rho^*. \end{aligned}$$

Second, the holding cost of $D(T)$ is

$$\begin{aligned} h(D(T)) &\leq \sum_{v \in V(T)} \sum_{(v,t) \in D_v} H^v_{t^*,t} \\ &\leq \sum_{v \in V(T)} A(v, t^*, \rho^*)\rho^*. \end{aligned}$$

We know $d(D(T)) = \sum_{v \in V(T)} A(v, t^*, \rho^*)$. So $\rho(T, D(T)) = \frac{r(T)+h(D(T))}{d(D(T))} \leq 3\rho^*$. □

Finally, we show that if all demand values are at least 1, then the greedy algorithm which uses a 3-approximate minimum density tree and coverage set each round still attains a logarithmic approximation for IRP. The derivation is a simple modification of the set cover analysis.

Theorem A.1.3. *If $d^v_t \geq 1 \forall (v,t) \in D$, then iteratively picking a 3-approximate minimum density tree and coverage set yields an $O(\log(d(D)))$-approximation for IRP.*

Proof. In each iteration, a set of density at most 3 times the minimum density was found. For each demand point, define its price to be the density of the set that covered it. Label the demand points in order of coverage from (v_1, t_1) to $(v_{|D|}, t_{|D|})$. Then the kth demand point covered has price at most $\frac{3OPT}{d(D \setminus \{(v_1,t_1),\ldots,(v_{k-1},t_{k-1})\})}$. So the final cost is at most $\sum_{k=1}^{|D|} \frac{3OPT}{d(D \setminus \{(v_1,t_1),\ldots,(v_{k-1},t_{k-1})\})} \leq 3H_{d(D)}OPT = O(\log(d(D)))OPT$, where the first inequality follows from $d^v_t \geq 1 \forall (v,t) \in D$. □

A.1.3 Implementation Detail

While the aforementioned greedy algorithm attains a provable bound on the cost, it is impractical to run on large instances. Just finding one coverage set per round involves a binary search for the best density value $\rho*$ that will induce an approximately minimum density set. On instances of 30 clients over 60 days, the version of greedy that searches for the approximately lowest density set took over an hour per instance. Another technique we tried to reduce the running time was to search for a single value of the lowest density such that there is a feasible set cover, instead of a different low density per round. However, the single density version of greedy still took over 45 minutes per 30 by 60 sized instance. Finally, we implemented a modification of this version that further limits the search space. First, given the existing trees T_t per day t and the uncovered set D, define the procedure $COVER(\rho)$ as in the following algorithm.

If ρ is too low, it is possible that $COVER(\rho)$ does not satisfy all demands. So whenever $COVER(\rho)$ stops serving new demands, we will relax the target density ρ by multiplying it a factor $\alpha > 1$ to continue serving demands until all are satisfied. Formally, given a relaxation factor $\alpha > 1$, we implement a heuristic called $GREEDY(\alpha)$ defined in Algorithm 17.

Algorithm 16: $COVER(\rho)$

while *there is a set of density* $\leq \rho$ **do**
  Find a day t, tree T on day t, and coverage set $D(T) \subset D$ with lowest density (minimized over $t \in T$), with PCST penalties induced by ρ and t
  $D \leftarrow D \setminus D(T)$
  $T_t \leftarrow T_t \cup T$

Algorithm 17: $GREEDY(\alpha)$

Initialize $T_t \leftarrow \varnothing \forall t \in [1, T]$
Find lowest value $\rho_{\min}$ such that $COVER(\rho_{\min})$ serves a nonempty set of demands and apply $COVER(\rho_{\min})$
while $|D| > 0$ **do**
  $\rho_{\min} \leftarrow \alpha * \rho_{\min}$
  Apply $COVER(\rho_{\min})$

To estimate the correct value for $\rho_{\min}$, we start with a small initial value for $\rho_{\min}$ and double it until $COVER(\rho_{\min})$ returns a nonempty set.

Since PCST already has fast near-optimal solvers, our implementation also differs from the stated algorithm by finding using the solver of Leitner et. al. (Leitner et al., 2020) to solve PCSTs rather than the primal dual algorithm of Goemans and Williamson (Goemans and Williamson, 1995).

Besides the pure Greedy heuristic, we also test how well Prioritized local search does if it is initialized with the solution from Greedy instead of from local search with ADDs. We refer to the combination of Greedy with Prioritized local search as *Pruned Greedy*.

In the next section, we show another application of the PCST ideas to design a primal-dual based heuristic for IRP.

A.2 Primal Dual Heuristic

In this section, we investigate a primal-dual approach similar to (Levi et al., 2006) for solving IRP. Inspired by the *waveform mechanism* introduced in (Levi et al., 2006) which was used for solving JRP, we generalize this idea and try to make it applicable for solving IRP. We will solve PCST instances where each vertex of the input represents a demand point of the IRP instance.

Section A.2.1 states the primal and dual LP relaxations for IRP. Using the LPs, the primal dual algorithm is presented in Section A.2.2. For simplicity of the algorithm, not all of the dual values are defined explicitly in the algorithm. In Section A.2.3, we prove that there is always a feasible setting of dual values corresponding

to the growth of moats in the algorithm. Finally, Section A.2.4 discusses a more efficient way that the primal dual algorithm can be implemented.

A.2.1 LP Formulation

To simplify the notation, we assume that each client v has a unique day t such that $d_t^v > 0$, otherwise we may add cost 0 edges to relabel multiple demand points at the same vertex as different vertices. Given a client v, the day t for which $d_t^v > 0$ is denoted by $t(v)$. For convenience, we use v to represent the demand point $(v, t(v))$. The variable y_s^e indicates whether edge e is used on day s. The variable $x_{s,t(v)}^v$ indicates whether demand $(v, t(v))$ is served on day s.

First, we state primal linear program and its dual:

$$\min \sum_{e \in E} \sum_{s=1}^{T} w_e y_s^e + \sum_{v \in D} \sum_{s=1}^{t(v)} H_{s,t(v)}^v x_{s,t(v)}^v$$

$$\text{s.t.} \sum_{s=1}^{t(v)} x_{s,t(v)}^v \geq 1 \qquad \forall (v, t(v)) \in D \tag{A.10}$$

$$\sum_{e \in \delta(X)} y_s^e \geq x_{s,t(v)}^v \qquad \forall (v, t(v)) \in D, 1 \leq s \leq t(v), X \subset V : X \ni v, X \not\ni r \tag{A.11}$$

$$x_{s,t(v)}^v \geq 0 \qquad \forall (v, t(v)) \in D, 1 \leq s \leq v(t) \tag{A.12}$$

$$y_s^e \geq 0 \qquad \forall e \in E, 1 \leq s \leq T \tag{A.13}$$

$$\max \sum_{(v,t(v)) \in D} b_{t(v)}^v$$

$$\text{s.t.} \; b_{t(v)}^v \leq H_{s,t(v)}^v + \sum_{X \ni v, X \not\ni r} \beta_{s,t(v)}^{v,X} \qquad \forall (v, t(v)) \in D, 1 \leq s \leq t(v) \tag{A.14}$$

$$\sum_{(v,t(v)) \in D, X \ni v, X \not\ni r, \delta(X) \ni e} \beta_{s,t(v)}^{v,X} \leq w_e \qquad \forall e \in E, 1 \leq s \leq T \tag{A.15}$$

$$b_{t(v)}^v \geq 0 \qquad \forall (v, t(v)) \in D \tag{A.16}$$

$$\beta_{s,t(v)}^{v,X} \geq 0 \qquad \forall (v, t(v)) \in D, X \subset V, 1 \leq s \leq T \tag{A.17}$$

In the primal LP, constraint A.10 ensures that every demand point is served on time. Constraint A.11 ensures that whenever a client v is served on day s, there is a path from the depot to v on day s.

A.2.2 A Primal-dual Approach

For the dual LP, variable $b^v_{t(v)}$ represents the budget amount that demand point $(v, t(v))$ has available to pay for visits to serve it. Variable $\beta^{v,X}_{s,t(v)}$ represents the amount that $(v, t(v))$ contributes towards building a tree crossing X to get served on day s. However, since the $\beta^{v,X}_{s,t(v)}$ variables are not part of objective function in the dual LP, we cannot directly use $\beta^{v,X}_{s,t(v)}$ to pay for visits. Instead, $\beta^{v,X}_{s,t(v)}$ represent copies of the total budget $b^v_{t(v)}$, one copy for each s. The general framework is to raise the budgets of demands as long as all constraints in the dual LP are able to hold. The final values of the budgets are determined by the tightening of dual constraints that they are involved in.

First, we describe the intuition of the algorithm. At the beginning of the algorithm, all budgets $b^v_{t(v)}$ and visit-specific payments $\beta^{v,X}_{s,t(v)}$ are to start at 0. We introduce a continuous parameter τ that slides through time from T to 1 at a constant rate. The position of τ within the time horizon will determine what value to raise the budgets and visit-specific payments. Whenever τ passes through an integral time t (i.e. $\tau < t$), it "wakes up" the budgets $b^v_{t(v)}$ of demands $(v, t(v))$ occurring on day $t(v) = t$. Those $b^v_{t(v)}$ shall increase at the same rate that $H^v_{\tau,t(v)}$ is increasing as τ is sliding towards 1, i.e., we keep $b^v_{t(v)}$ at exactly the same value as $H^v_{\tau,t(v)}$. The definition of $H^v_{\tau,t}$ for non-integral τ is interpolated linearly, i.e., define $H^v_{\tau,t} = (1 - \tau + \lfloor\tau\rfloor)H^v_{\lfloor\tau\rfloor,t} + (\tau - \lfloor\tau\rfloor)H^v_{\lceil\tau\rceil,t}$.

Observe that keeping $b^v_{t(v)} = H^v_{\tau,t(v)}$ ensures that each demand $(v, t(v))$ can at least pay for the holding cost from time τ to $t(v)$. To maintain feasibility to the dual constraints, we also raise $\beta^{v,X}_{s,t(v)}$ as needed to keep constraint A.14 satisfied. That means for each demand $(v, t(v))$ and each $s \in (\tau, t(v)]$, we raise the value of $\sum_{X \ni v, X \not\ni r} \beta^{v,X}_{s,t(v)}$ to at least $H^v_{\tau,t(v)} - H^v_{s,t(v)}$. For each value of τ and s, we create a PCST instance whose penalty at v is assigned to $H^v_{\tau,t(v)} - H^v_{s,t(v)}$ and solve it using the primal dual algorithm of Goemans and Williamson (Goemans and Williamson, 1995). The value to raise each $\beta^{v,X}_{s,t(v)}$ will be determined by the dual values of the PCST instance set by primal dual algorithm (Goemans and Williamson, 1995). We defer the details of the exact values to set them to the proof of feasibility for Theorem A.2.1.

Next, we give the necessary definitions to state the algorithm formally. Initially, all the dual variables are *unfrozen*. During the running of the algorithm, we set the value of the dual variables as τ goes to 1. By *freezing* a dual variable we mean that the value of that particular variable will not change from then on. A vertex $v \in D$ is a *frozen vertex* if and only if $b^v_{t(v)}$ is frozen. In the algorithm, we shall serve a vertex whenever it becomes frozen. Let $\mathcal{F}$ denote the set of the all frozen vertices since the beginning of the algorithm until the current moment, i.e. since when $\tau = T$ till when $\tau = t$ where t is the current location of the sweep line.

The algorithm assigns a service time $l(v)$ to each frozen vertex v; the details of the assignment to v will be explained later. This assignment would be in such a way that: $1 \le l(v) \le t(v) \le T$, and for any $v \in \mathcal{F}$, we have $b^v_{t(v)} = H^v_{l(v),t(v)}$.

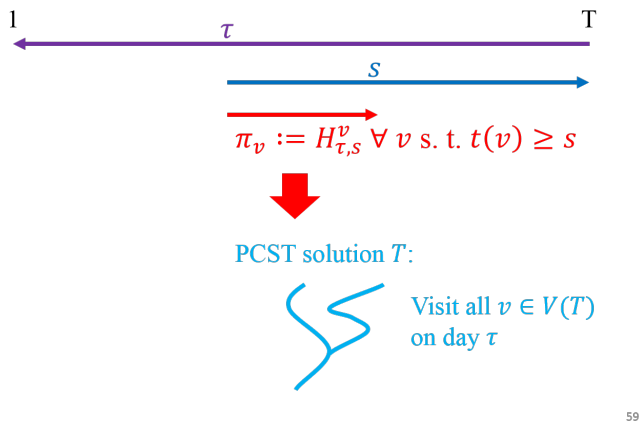


Figure A.2: At each value of τ and s, we define a PCST instance whose penalty at each client is the holding cost to store the product there from day τ to day s. A solution to the PCST instance determines the subset of clients to visit on day τ. After this procedure is repeated for every value of τ and s, we know exactly which clients are visited that day.

Finally, define the set of *active vertices at time* s to be $\mathcal{A}(s) = \{v : v \in D, s \leq t(v)\}$ for all $s \in [1, T]$.

Now, we are ready to give the algorithm formally in Algorithm 18. For the sake of intuition, we give a continuous description of the algorithm which can be easily modified to be a discrete and polynomial time algorithm. Figure A.2 provides a visualization of the algorithm.

Observe that at the end, all clients will have been frozen and served at some point.

A.2.3 Defining a Feasible Dual

Next, we show that there is a feasible dual solution $\boldsymbol{b}, \boldsymbol{\beta}$ satisfying the assignment of values for b^v_t from the algorithm. For the analysis, we shall refer to an particular iteration in the algorithm by the value of τ and s at that point.

Theorem A.2.1. *During any moment (τ, s) of the algorithm, for the setting $b^v_{t(v)} = H^v_{\tau,t(v)}$, there is an assignment of $\boldsymbol{\beta}$ so that $\boldsymbol{b}, \boldsymbol{\beta}$ is feasible to the dual.*

Proof. Assume that we are in iteration (τ, s) of the algorithm. Let y_S be the values of the dual variables corresponding to the primal dual solution for PCST in this iteration. Note that y_S depends on (τ, s), but we omit further subscripting by (τ, s) for simplicity of notation. We will distribute the dual value y_S among the client-specific dual variables $\beta^{v,S}_{s,t(v)}$ with the goal of satisfying constraint A.14.

Define the *potential* of client v to be $p(v) := \pi_v - \sum_{S \ni v, S \not\ni r} \beta^{v,S}_{s,t(v)}$. Initialize $\boldsymbol{\beta} = 0$. As y_S grows, assign

Algorithm 18: Primal Dual

```
Initialize $\mathcal{F} \leftarrow \varnothing$ and $\forall 1 \leq s \leq T$, $\mathcal{A}(s) \leftarrow \{v : v \in D, s \leq t(v)\}$
for $\tau \leftarrow T$ towards $1$ do
    for $s \leftarrow \lceil \tau \rceil$ to $T$ do
        Make an instance of the prize-collecting Steiner tree problem by assigning a penalty $\pi_v$ to each
          vertex $v \in \mathcal{A}(s)$ as follows for all $v \in \mathcal{A}(s)$ do
            if $v \notin \mathcal{F}$ then
                $\pi_v = H^v_{\tau,t(v)} - H^v_{s,t(v)}$
            else
                $\pi_v = 0$
        Solve the prize-collecting Steiner tree instance using the classical primal dual
          algorithm (Goemans and Williamson, 1995) and let $X$ be the subset of $\mathcal{A}(s)$ getting connected
          to the root $r$ in the solution if $X \not\subset \mathcal{F}$ then
            For all $v \in X \backslash \mathcal{F}$ let $l(v) = \tau$ and $b^v_{t(v)} = H^v_{l(v),t(v)}$, and visit $v$ at time $\lceil l(v) \rceil$ (the values to
              set $\beta^{v,X}_{s,t(v)}$ will be provided in the proof of Theorem A.2.1)
            $\mathcal{F} \leftarrow \mathcal{F} \cup X$
            Freeze the unfrozen vertices in $X$
For all $v \notin \mathcal{F}$ let $b^v_{t(v)} = H^v_{1,t(v)}$ and visit $V \setminus \mathcal{F}$ on day $1$
Output the IRP schedule specified by the service times for each demand point
Output the dual variables $b^v_{t(v)}$
```

$\beta^{v,S}_{s,t(v)} = \frac{y_S}{|\{v \in S : p(v) > 0\}|}$. Next, we show that this setting of $\boldsymbol{\beta}$ along with the setting $b^v_{t(v)} = H^v_{\tau,t(v)}$ of the algorithm constitutes a feasible dual solution to IRP.

First, we can easily verify constraint A.15. For a given $e \in E, s \leq T$, we have

$$\begin{aligned}
\sum_{v \in V, X \ni v, X \not\ni r, \delta(X) \ni e} \beta^{v,X}_{s,t(v)} &\leq \sum_{X : \delta(X) \ni e, X \not\ni r} \sum_{v \in X} \beta^{v,X}_{s,t(v)} \\
&\leq \sum_{X : \delta(X) \ni e, X \not\ni r} y_X \text{ by definition of } \boldsymbol{\beta} \\
&\leq w_e \text{ by the dual constraints for PCST.}
\end{aligned}$$

Second, we show that $\sum_{X \ni v, X \not\ni r} \beta^{v,X}_{s,t(v)} \geq H^v_{\tau,t(v)} - H^v_{s,t(v)}$ for all $v \in V$ and $s \leq t(v)$, which would imply constraint A.14. Fix v and s. Consider the moment just before $b^v_{t(v)}$ froze, which means the previous PCST solution did not span v. By the primal dual algorithm of Goemans and Williamson, v was in some set X such that $\pi(X) = \sum_{S : S \subset X} y_S$. Then

$$\begin{aligned}
\pi(X) &= \sum_{S : S \subset X} y_S \\
&= \sum_{S \in X} \sum_{v \in S} \beta^{v,S}_{s,t(v)} \\
&\leq \sum_{S \not\ni r} \sum_{v \in S \cap X} \beta^{v,S}_{s,t(v)} \\
&= \sum_{v \in X} \sum_{S \ni v, S \not\ni r} \beta^{v,S}_{s,t(v)} \\
&\leq \sum_{v \in X} \pi_v \text{ since only those } v \text{ whose potential are positive grow their } \beta^{v,S}_{s,t(v)} \\
&= \pi(X).
\end{aligned}$$

So all inequalities must be equalities, which means that $\sum_{X \ni v, X \not\ni r} \beta^{v,X}_{s,t(v)} = \pi_v = H^v_{\tau,t(v)} - H^v_{s,t(v)}$. Hence constraint A.15 holds.

□

A.2.4 Implementation

Here, we provide a simpler implementation of Algorithm 18, which does not require setting dual values and eliminates the loop over s from $\lceil \tau \rceil$ to T. In Algorithm 18, the purpose of the loop over s is to help determine feasible dual values to set for the variables $\beta^{v,X}_{s,t(v)}$ to prove Theorem A.2.1. However, for purposes of obtaining the same primal solution, we do not need to create and solve the PCST instance per s value.

For a fixed τ, no matter which value s takes, the vertices spanned by the PCST solution all become assigned to the service day τ. Also, for each demand day $t(v)$, there is some round when s takes value $t(v)$, so that the penalty assigned to v is at its highest possible value $H^v_{\tau,t(v)}$. If v gets assigned to be served on day τ by any s, it would certainly be part of the PCST solution to the instance having the highest penalty $H^v_{\tau,t(v)} - H^v_{t(v),t(v)}$. So instead of collecting the visits on day τ separately through different values of s, we could solve one PCST instance to determine the visit set for day τ by setting penalty $H^v_{\tau,t(v)}$ for v to collect all visits that could possibly have been induced by the largest s. Similarly, the raising of dual values $b^v_{t(v)}$ in Algorithm 18 was included to help prove Theorem A.2.1 and is not needed to determine the primal solution. One last detail we modify is the solution method for PCSTs. Algorithm 18 solved PCSTs using (Goemans and Williamson, 1995) so that the dual values of PCST from (Goemans and Williamson, 1995) could be used to determine the dual values to set $\beta^{v,X}_{s,t(v)}$, again to prove feasibility. However, for faster solving time, we solve PCSTs using (Leitner et al., 2020) instead since we only need to recover the primal solution at the end regardless of the dual values. Further, our implementation is a simplification of the original algorithm that discretizes τ to take only integer values from T to 1. This allows us to use the fast PCST solver of Leitner et. al. (Leitner et al., 2020) in a self-contained manner rather than having the breakpoints of τ depend on the dual solution for PCST. However, as noted above, the simplification only finds a primal solution for IRP. The dual values are no longer valid after restricting the breakpoints of τ to only integers. Algorithm 19 describes the aforementioned heuristic exactly as implemented.

Algorithm 19: Primal Only Implementation

```
Initialize F ← ∅ and ∀1 ≤ s ≤ T, A(s) ← {v : v ∈ D, s ≤ t(v)}
for τ ← T to 1 do
    Make an instance of the prize-collecting Steiner tree problem by assigning a penalty π_v to each
      vertex v ∈ A(τ) as follows for all v ∈ A(τ) do
        if v ∉ F then
            π_v = H^v_{τ,t(v)}
        else
            π_v = 0
    Solve the prize-collecting Steiner tree instance using the solver (Leitner et al., 2020) and let X be
      the subset of A(τ) getting connected to the root r in the solution
    if X ⊄ F then
        For all v ∈ X\F, visit v at time τ
        F ← F ∪ X
        Freeze the unfrozen vertices in X
For all v ∉ F, visit V \ F on day 1
Output the IRP schedule specified by the service times for each demand point
```

In addition to the pure Primal Dual heuristic, we test Prioritized local search initialized with the solution from the Primal Dual heuristic, which we call *Pruned Primal Dual*.

A.3 Uncapacitated IRP Results

In this section, we give two MIP formulations of uncapacitated IRP. Following that, we provide plots of all heuristics' gaps and running times against the parameters H, N, and T.

A.3.1 MIP Formulation for Uncapacitated IRP

First, we describe a compact MIP formulation of the IRP, that we use with modern solvers to establish the benchmark for comparing our solutions. Our exact MIP formulation for IRP is of size $O(N^2T) + O(NT^2)$. When the problem instances get larger, we are however only able to generate lower bounds for the value of a solution even using state-of-the-art solvers such as Gurobi Version 7.

As before, $x^v_{s,t}$ will be the variable indicating whether to serve (v,t) on day s. Define a related variable X^v_s indicating whether v is visited on day s. Let z^{uw}_s be the variable indicating whether to use an arc uw on day s. Let h^{uw}_s be the continuous variable representing the amount of total flow through arc uw on day s coming from the depot.

Intuitively, the purpose of h^{uw}_s is to enable expressing connectivity in a polynomial number of constraints, in contrast with using a non-compact set of exponentially many cut-covering constraints. In Figure A.3, we provide an example of how the values of h^{uw}_s are set in a feasible solution. Then IRP is modeled by the following MIP.

$$\min \sum_{u\in V}\sum_{w\neq u\in V}\sum_{s=1}^{T} c_{uw} z^{uw}_s + \sum_{(v,t)\in D}\sum_{s=1}^{t} H^v_{s,t} x^v_{s,t}$$

$$\text{s.t. } z^{uw}_s + z^{wu}_s \leq 1 \qquad \forall u \in V, w > u, s \leq T \tag{A.18}$$

$$\sum_{s=1}^{t} x^v_{s,t} = 1 \qquad \forall v \in V, t \leq T \tag{A.19}$$

$$X^v_s \geq x^v_{s,t} \qquad \forall v \in V, t \leq T, s \leq t \tag{A.20}$$

$$\sum_{w\neq v} z^{vw}_s = X^v_s \qquad \forall v \in V \setminus \{r\}, s \leq T \tag{A.21}$$

$$\sum_{w\neq v} z^{wv}_s = X^v_s \qquad \forall v \in V \setminus \{r\}, s \leq T \tag{A.22}$$

$$\sum_{w\neq r} z^{rw}_s \leq 1 \qquad \forall s \leq T \tag{A.23}$$

$$\sum_{w\neq r} z^{wr}_s \leq 1 \qquad \forall s \leq T \tag{A.24}$$

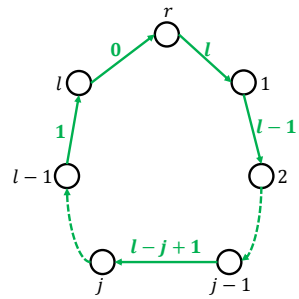


Figure A.3: For a fixed day s, suppose that nodes $1, \ldots, l$ are visited by a cycle in a feasible solution to IRP. To determine the appropriate values to set h_s^{uw} variables, note that each visited node contributes one unit of flow along the path from from r to itself. Then the flow through an arc uw would be the total number of all the paths between r and visited nodes that have uw in the path. The labels along the arcs indicate the values that h_s^{uw} would take per arc uw. Values of the remaining variables would be set in the obvious ways: $z_s^{uw} = 1$ if and only if arc uw is in the cycle, $X_s^v = 1$ if and only if $v \in \{1, \ldots, l\}$, $x_{st}^v = 1$ if and only if day s is the latest day before or on day t having a visit to v.

$$\sum_{w \neq u} h_s^{wu} - \sum_{w \neq u} h_s^{uw} = \begin{cases} X_s^u, u \neq r \\ \sum_{a \neq r} -X_s^a, u = r \end{cases} \qquad \forall u \in V, s \leq T \tag{A.25}$$

$$h_s^{uw} \leq (N-1) z_s^{uw} \qquad \forall u \in V, w \neq u, s \leq T \tag{A.26}$$

$$X_s^v, x_{s,t}^v, z_s^{uw} \in \{0, 1\} \qquad \forall v \in V, u \in V, w \neq u \in V, t \leq T, s \leq T \tag{A.27}$$

$$h_s^{uw} \geq 0 \qquad \forall u \in V, w \neq u, s \leq T \tag{A.28}$$

Constraint A.18 ensures that each edge is used at most once. Constraint A.19 guarantees that all demands are satisfied on time. Constraint A.20 ensures that whenever a demand is served on a specified day, there must be a visit to the client on that day. Constraints A.21 and A.22 guarantee that if a vertex is visited, then some in-arc and some out-arc incident to it must be traversed. Constraints A.23 and A.24 limit the number of cycles to 1.

We needed a separate case for the fractional degree at r because the depot could be served by itself on day s while not building any arcs on day s, which means that $\sum_{w \neq r} z_s^{rw}$ and $\sum_{w \neq r} z_s^{wr}$ could potentially be 0 even when $X_s^r = 1$. Constraint A.25 ensures that the net in-flow into any $u \neq r$ corresponds to whether u is visited on that day, and the net in-flow into r corresponds to the negative of the number of vertices visited (i.e., out-flow of one per node). Constraint A.26 requires that on each day, an arc must be built if there is flow through it from the depot, and the flow allowed is bounded by the maximum possible number of visited nodes.

Solving this MIP directly within MIPGap of 10% was not practical past instances of size 140 (nodes) by 6 (days) and 50 by 16. We use the lower bound found at 10% MIPGap to compare with the costs from our heuristics.

A.3.2 Additional Experimental Results on Uncapacitated IRP

First, we present results from varying the holding cost scaling parameter H. Then we provide more details of performance plots as we vary all three parameters N, T and H.

Varying H

In this test, N and T are fixed to 100 and 6, respectively. The holding cost scale H varies from 0.01 to 6.01 at increments of 0.5. Results that require lower bound from the MIP go up to only $H = 4.51$ due to high running times of the MIP and the large number of instances per parameter value. In summary, the heuristics in order of lowest to highest gaps are Prioritized Local Search, ADD, DELETE, Pruned Greedy, Pruned Primal Dual, Primal Dual, and Greedy. Detailed results are listed below.

- For ADD and Prioritized Local Search, the gap is at most 5% for all values of H. The values of H for which they have their highest gaps occur at 2.01 and 2.51. Since ADD and Prioritized Local Search are richer operations in their use of the PCST solutions, we expect them to reach close to optimality. The Prioritized Local Search and Local Search with ADDs have similar gaps because the solution from ADD is already nearly optimal and there is little room to improve the cost.
- For DELETE Local Search, the gap is largest at 1.1 for H at 2.51 and 3.01.
- Primal dual has its largest gap of 1.19 at $H = 2.51$.
- Pruned Primal Dual's gaps increase as H increases, starting from 1.01 at $H = 0.01$ to 1.11 at $H = 4.51$. It might be the case that the gap would eventually taper off at a larger value of H, but we do not have the lower bounds to compare with due to the long solution time of the MIP.

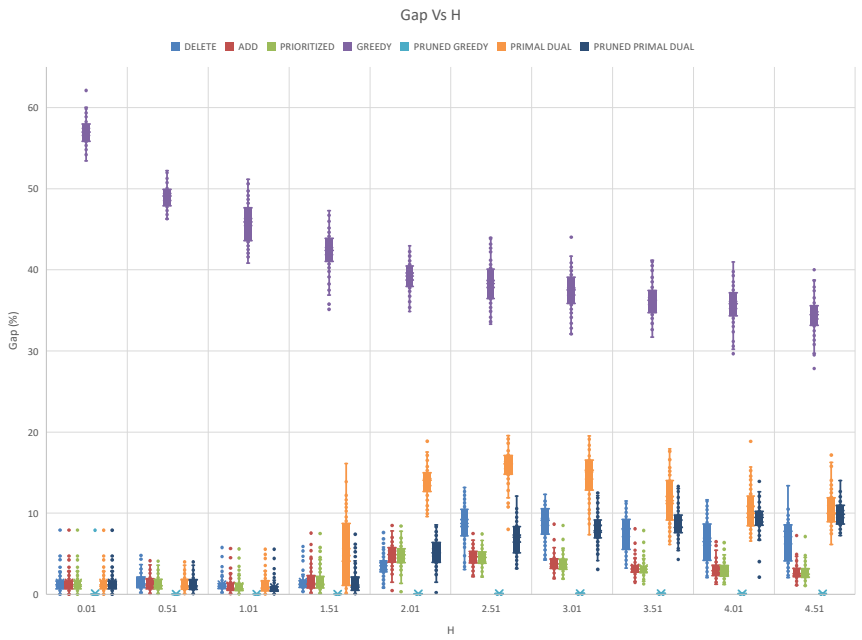


Figure A.4: Delete, Add, and Prioritized each correspond to the local search that DELETEs, ADDs, and all operations, respectively. The gaps for Delete, Add, Prioritized Local Search, Greedy, Pruned Greedy, Primal Dual, and Pruned Primal Dual are shown in blue, red, green, purple, light blue, orange, and dark blue, respectively.

Varying N

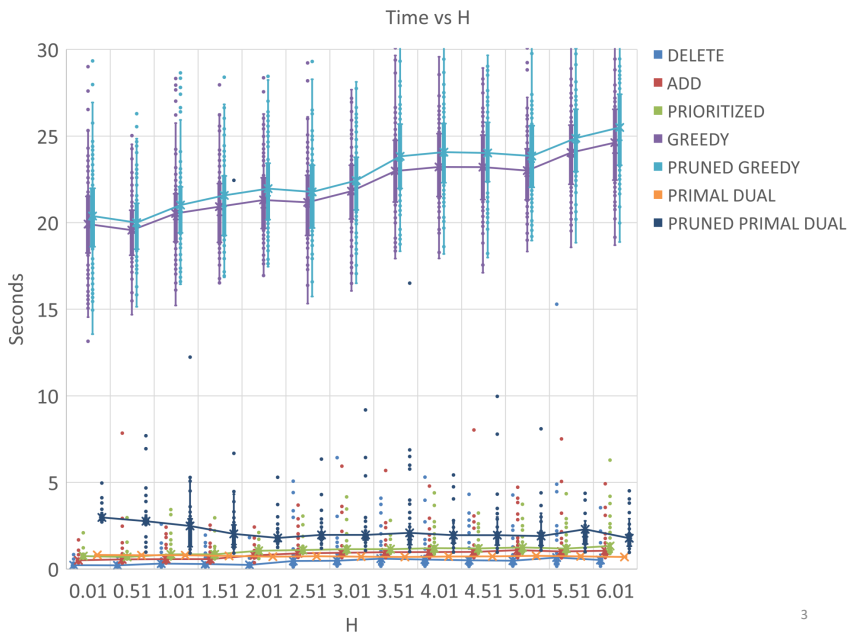


Figure A.5: The running times for Delete, Add, Prioritized Local Search, Greedy, Pruned Greedy, Primal Dual, and Pruned Primal Dual are shown in blue, red, green, purple, light blue, orange, and dark blue, respectively.

Varying T

A.4 MIP Formulation for Capacitated IRP

We adapt a MIP formulation for the multivehicle production and inventory routing problem due to Adulyasak et al. (2014) to our specific problem that we use with modern solvers to establish the benchmark for comparing our solutions. Specifically, we used the formulation in Secion 2.2.1 of their paper with a vehicle index for the trucks, oriented towards a policy that allows replenishment up to the maximum level of inventory at each destination (as opposed to the other formulations in the paper that avoid vehicle indices and use delivery amounts that obey order-up-to levels at destinations). We ignored constraints and variables related to depot and location capacities, as well as production capacity constraints at the depot since they were not addressed in our study. We describe the resulting formulation below.

We first define variables used in the MIP: I_{it} is the inventory at node i at the end of period t. z_{ikt} is equal to 1 if node i is visited by vehicle k in period t and 0 otherwise. x_{ijkt} is equal to 1 if vehicle k travels directly between node i and node j in period t and 0 otherwise. q_{ikt} is the quantity delivered to customer i with vehicle k in period t. h_i is the unit inventory holding cost at node i. c_{ij} is the transportation cost between node i and j. d_{it} is the demand at customer i in period t. Q is the vehicle capacity. Recall that V is the set of

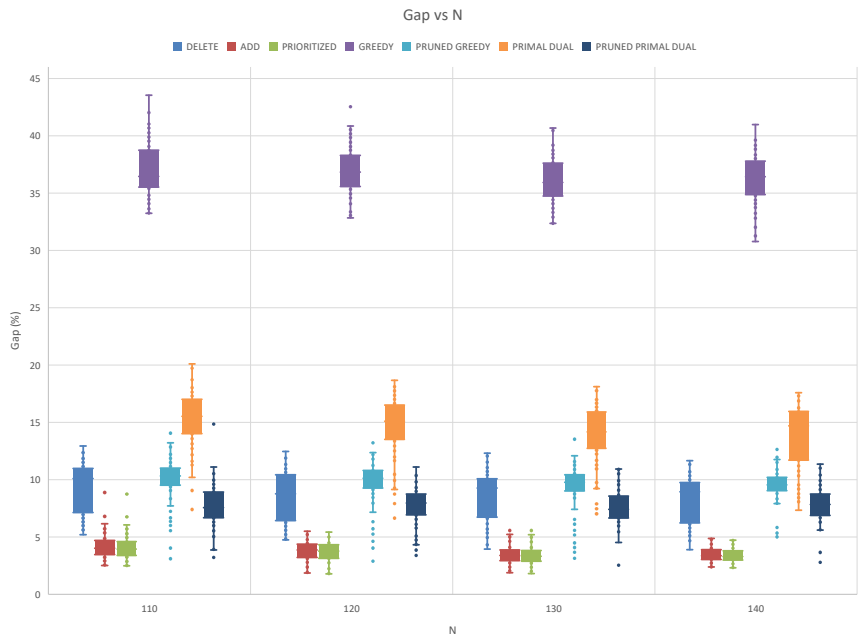


Figure A.6: The gaps for Delete, Add, Prioritized Local Search, Greedy, Pruned Greedy, Primal Dual, and Pruned Primal Dual are shown in blue, red, green, purple, light blue, orange, and dark blue, respectively.

clients, r is the depot, K is the number of trucks and we have T periods in the problem specification.

$$\min \quad \sum_{t \in T} \left(\sum_{i \in N} h_i I_{it} + \sum_{(i,j) \in E} \sum_{k \in K} c_{ij} x_{ijkt} \right) \tag{A.29}$$

$$\text{s.t.} \quad I_{i,t-1} + \sum_{k \in K} q_{ikt} = d_{it} + I_{it} \qquad \forall i \in V, t \in T, \tag{A.30}$$

$$\sum_{i \in V} q_{ikt} \leq Q z_{0kt} \qquad \forall k \in K, \forall t \in T, \tag{A.31}$$

$$\sum_{k \in K} z_{ikt} \leq 1 \qquad \forall i \in V, \forall t \in T, \tag{A.32}$$

$$q_{ikt} \leq \min\{Q, \sum_{j=t}^{l} d_{ij}\} z_{ikt} \qquad \forall i \in V, \forall k \in K, \forall t \in T, \tag{A.33}$$

$$\sum_{(j,j') \in \delta(i)} x_{jj'kt} = 2 z_{ikt} \qquad \forall i \in V \cup \{r\}, \forall k \in K, \forall t \in T, \tag{A.34}$$

$$\sum_{(i,j) \in E(S)} x_{ijkt} \leq \sum_{i \in S} z_{ikt} - z_{i'kt} \qquad \forall S \subset V, |S| \geq 2, \forall i' \in S, \forall k \in K, \forall t \in T, \tag{A.35}$$

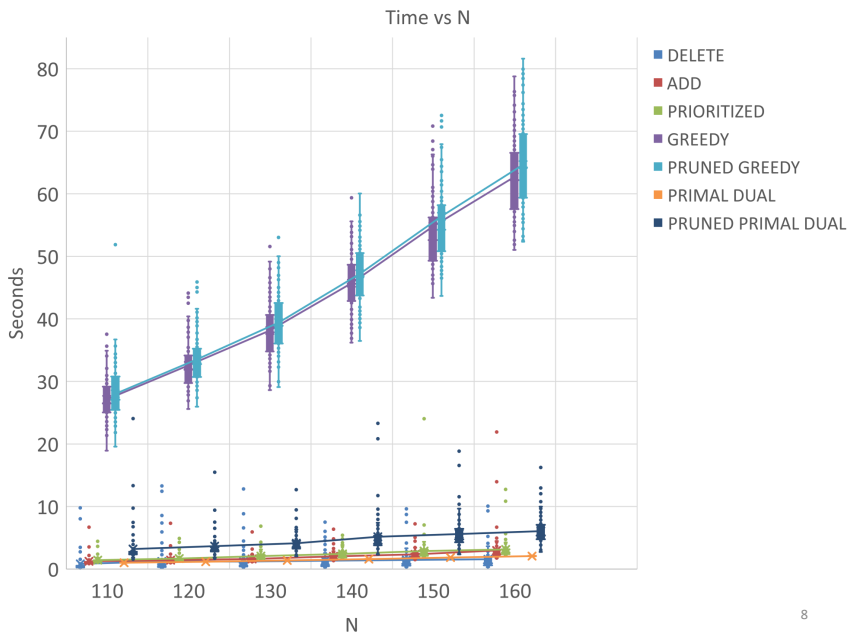


Figure A.7: The running times for Delete, Add, Prioritized Local Search, Greedy, Pruned Greedy, Primal Dual, and Pruned Primal Dual are shown in blue, red, green, purple, light blue, orange, and dark blue, respectively.

$$I_{i,t-s-1} \geq \left(\sum_{j=0}^{s} d_{i,t-j}\right)\left(1 - \sum_{k\in K}\sum_{j=0}^{s} z_{ik,t-j}\right) \qquad \forall i \in V, \forall t \in T, s = 0, 1, \ldots, t-1, \tag{A.36}$$

$$z_{ikt} \leq z_{rkt} \qquad \forall i \in V, \forall k \in K, \forall t \in T, \tag{A.37}$$

$$x_{ijkt} \leq z_{ikt}, x_{ijkt} \leq z_{jkt} \qquad \forall (i,j) \in E(V), \forall k \in K, \forall t \in T, \tag{A.38}$$

$$z_{rkt} \geq z_{r,k+1,t} \qquad \forall 1 \leq k \leq m-1, \forall t \in T, \tag{A.39}$$

$$I_{it}, q_{ikt} \geq 0 \qquad \forall i \in V \cup \{r\}, \forall k \in K, \forall t \in T, \tag{A.40}$$

$$z_{ikt} \in \{0,1\} \qquad \forall i \in V \cup \{r\}, \forall k \in K, \forall t \in T, \tag{A.41}$$

$$x_{ijkt} \in \{0,1\} \qquad \forall (i,j) \in E : i \neq r, \forall k \in K, \forall t \in T, \tag{A.42}$$

$$x_{rjkt} \in \{0,1,2\} \qquad \forall j \in V, k \in K, t \in T. \tag{A.43}$$

Constraint (A.30) ensures inventory balance at all nodes across time. Constraint (A.31) enforces the vehicle capacity constraints. Constraint (A.33) allows positive delivery from a truck at a location only if the truck visits that node in that time period. We have the subtour elimination constraints in (A.35). Constraints (A.36) ensure that sufficient inventory is at hand to satisfy future demand in light of no future vehicle visits.

The above MIP is solved by the branch-and-cut method where the subtour elimination constraints (A.35) is

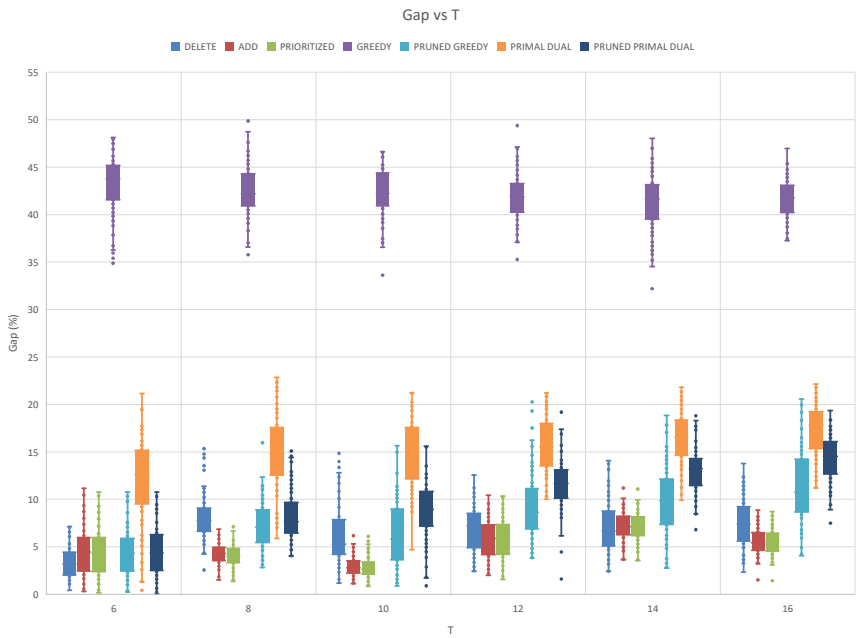


Figure A.8: The gaps for Delete, Add, Prioritized Local Search, Greedy, Pruned Greedy, Primal Dual, and Pruned Primal Dual are shown in blue, red, green, purple, light blue, orange, and dark blue, respectively.

separated via a generic min-cut subroutine. The branch-and-cut method is implemented in SCIP 6.0.2.

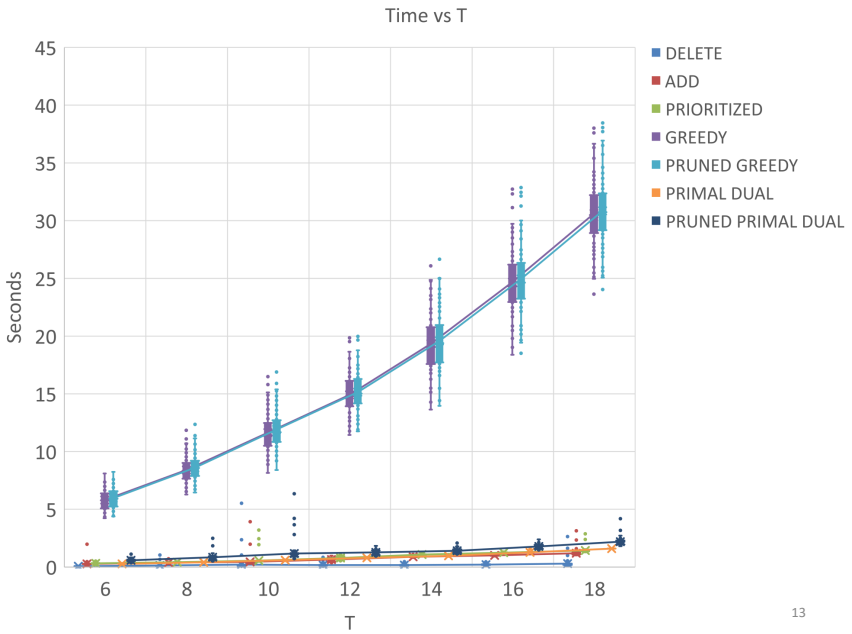


Figure A.9: The running times for Delete, Add, Prioritized Local Search, Greedy, Pruned Greedy, Primal Dual, and Pruned Primal Dual are shown in blue, red, green, purple, light blue, orange, and dark blue, respectively.

Appendix B

Appendix for Chapter 5

We describe a compact formulation for the TSP-D, which is a direct adoption of the MIP model from Roberti and Ruthmair (2019) with big-M constraints replaced with indicator constraint. Let A and B be two statements, we define $A \rightarrow B$ denote the constraint that if A is true, then B is true. Let $x_{ij}^T \in \{0,1\}$ be a binary variable equal to 1 if the truck traverses arc $(i,j) \in A$ (no matter if the drone is on-board or airborne), and let $x_{ij}^D \in \{0,1\}$ be a binary variable equal to 1 if the drone traverses arc $(i,j) \in A$ (no matter if it is on-board or airborne). Let $y_i^T \in \{0,1\}$ ($y_i^D \in \{0,1\}$, resp.) be a binary variable equal to 1 if $i \in N$ is a truck customer (drone customer, resp.). Moreover, let $y_i^C \in \{0,1\}$ be a binary variable equal to 1 if $i \in N$ is a combined customer. Finally, let $a_i \in \mathbb{R}_+$ be the arrival time of the truck or the drone (or both) at node $i \in V$. The TSP-D can be formulated as:

$$\min \quad a_{0'} \tag{B.1}$$

$$\text{s.t.} \quad \sum_{(i,j)\in A} x_{ij}^T = \sum_{(j,i)\in A} x_{ji}^T, \quad i \in N \tag{B.2}$$

$$\sum_{(i,j)\in A} x_{ij}^T = y_i^T + y_i^C, \quad i \in N \tag{B.3}$$

$$\sum_{(0,j)\in A} x_{0j}^T = \sum_{(i,0')\in A} x_{i0'}^T = 1 \tag{B.4}$$

$$\sum_{(i,j)\in A} x_{ij}^D = \sum_{(j,i)\in A} x_{ji}^D, \quad i \in N \tag{B.5}$$

$$\sum_{(i,j)\in A} x_{ij}^D = y_i^D + y_i^C, \quad i \in N \tag{B.6}$$

$$\sum_{(0,j)\in A} x_{0j}^D = \sum_{(i,0')\in A} x_{i0'}^D = 1 \tag{B.7}$$

$$y_i^T + y_i^D + y_i^C = 1, \quad i \in N \tag{B.8}$$

$$x_{ij}^T = 1 \rightarrow a_i + t_{ij}^T \leq a_j, \quad \forall (i,j) \in A \tag{B.9}$$

$$x_{ij}^D = 1 \rightarrow a_i + t_{ij}^D \leq a_j, \quad \forall (i,j) \in A \tag{B.10}$$

$$x_{ij}^D + x_{ji}^D \leq y_i^D + y_j^D, \quad \forall i,j \in N, i < j \tag{B.11}$$

$$x_{ij}^T, x_{ij}^D \in \{0,1\}, \quad \forall (i,j) \in A \tag{B.12}$$

$$y_i^T, y_i^D, y_i^C \in \{0,1\}, \quad \forall i \in N \tag{B.13}$$

$$a_i \in \mathbb{R}_+, \quad \forall i \in N \tag{B.14}$$

$$a_{0'} \geq \sum_{(i,j)\in A} t_{ij}^T x_{ij}^T, a_{0'} \geq \sum_{(i,j)\in A} t_{ij}^D x_{ij}^D \tag{B.15}$$

The objective function (B.1) aims at minimizing the total tour duration to serve all customers. Constraints (B.2) are flow conservation constraints for the truck. Constraints (B.3) links x_{ij}^T variables with y_i^T and y_i^C variables. Constraints (B.4) ensure that the truck leaves and returns to the depot exactly once. Constraints (B.6)-(B.7) correspond to constraints (B.2)-(B.4) but are defined for the drone. Constraints (B.8) ensure that each customer is visited at least once. The next two set of constraints act as subtour elimination constraints and set the arrival time at each node of the truck/drone. Constraints (B.11) ensure that the drone travels from $i \in N$ to $j \in N$ if and only if the truck visits at least one of the customers i and j, thus ensuring that each drone leg consists of a single drone customer. Constraints (B.15) state that the duration of the tour cannot be lower than the maximum of the sum of the travel time of the arcs traversed by the truck and that traversed by the drone.