Submitted to *Transportation Science*
manuscript TS-2021-0170.R2

Dual Bounds from Decision Diagram-Based Route Relaxations: An Application to Truck-Drone Routing

Ziye Tang
Tepper School of Business, Carnegie Mellon University, z.orion.tang@gmail.com,

Willem-Jan van Hoeve*
Tepper School of Business, Carnegie Mellon University, vanhoeve@andrew.cmu.edu,

For vehicle routing problems, strong dual bounds on the optimal value are needed to develop scalable exact algorithms, as well as to evaluate the performance of heuristics. In this work, we propose an iterative algorithm to compute dual bounds motivated by connections between decision diagrams and dynamic programming models used for pricing in branch-and-cut-and-price algorithms. We apply techniques from the decision diagram literature to generate and strengthen novel route relaxations for obtaining dual bounds, without using column generation. Our approach is generic and can be applied to various vehicle routing problems where corresponding dynamic programming models are available. We apply our framework to the traveling salesman with drone problem, and show that it produces dual bounds competitive to those from the state of the art. Applied to larger problem instances where the state-of-the-art approach does not scale, our method outperforms other bounding techniques from the literature.

Key words: truck-drone routing, route relaxation, decision diagrams, network flow, Lagrangian relaxation
History:

1. Introduction

The problem of supplying customers using vehicles based at a central depot is generally known as a *vehicle routing problem* (VRP) (Toth and Vigo 2014). VRPs have become increasingly important with the evolution of online shopping and fulfillment and a variety of delivery services. Alongside the growing commercial value of VRPs, tremendous amount of research has been carried out on mathematically 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*

* Partially supported by Office of Naval Research Grant No. N00014-21-1-2240 and National Science Foundation Award #1918102.

(BCP) algorithms (Costa, Contardo, and Desaulniers 2019). In general, a BCP algorithm is based on a mathematical programming formulation with a huge number of variables: one for each possible route. It starts, however, with a small number of variables representing a restricted *master problem* and iteratively generates new promising variables. The problem of finding new promising variables is referred to as the *pricing* problem. As variables can also be viewed as columns in the model, this iterative process is referred to as *column generation* (CG). 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 (BCP).

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 dual (or lower) bound on the optimal value. Such a bound is usually computed by solving the linear relaxation of the SP. 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 VRP 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 Christofides, Mingozzi, and Toth (1981). The resulting pricing problem is typically solved by *dynamic programming* (DP).

A drawback of using column generation for solving the set partition formulation is that no dual bound can be reported until an optimal basis to the LP relaxation of the master problem has been found. [1] As we will see in our experimental evaluation, when many iterations are required this can be prohibitive to computing a dual bound in reasonable time. Rather than working with a restricted model (using a subset of variables), we therefore investigate the use of a *relaxed* model that over-approximates the set of variables, using *decision diagrams* (DDs). Such a relaxed model always returns a valid dual bound to the original VRP, and it can be strengthened using iterative refinement techniques from the decision diagram literature. This can be particularly useful when

[1] We note that it is possible to terminate the column generation process prematurely and derive an approximate bound based on reduced cost information, as discussed in (Barnhart et al. 1998).

applied to larger instances of difficult VRPs where solving the master problem takes up a significant portion of the entire solution time.

Specifically, we propose novel route relaxations based on DDs, which are motivated by a close connection with DP models used for solving pricing problems. In fact, a DP pricing model can also be used to define a DD, which can yield a dual bound without solving multiple iterations of column generation. We introduce two approaches to compute dual bounds from DD-based route relaxations: a network flow model with side constraints, and a Lagrangian relaxation. The following simple and yet fundamental observation highlights the relationship between our approaches and the CG approach for solving the set partitioning formulation (its mathematical counterpart is Theorem 2).

THEOREM 1 **(Informal)**. *Both the decision diagram-based network flow and Lagrangian relaxation approaches are equivalent to column generation in the sense that they all compute the same lower bound under the same route relaxation.*

Based on the above result, we furthermore propose to dynamically adjust the DD-based route relaxations via iterative refinement of the diagram to produce increasingly stronger dual bounds. We numerically evaluate the performance of our framework on a new and challenging VRP variant called the *Traveling Salesman Problem with Drone* (TSP-D), where cooperation between the truck and the drone makes the problem difficult to model and to solve. Computational experiments show that our approaches are able to generate lower bounds that 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 valid lower bound within an hour, our approaches are shown to outperform lower bounding methods based on constraint programming and mixed-integer programming.

The rest of this paper is organized as follows. In Section 2 we review route relaxations used by BCP algorithms and relevant background on DDs. In Section 3 we provide a formal definition of the TSP-D. Section 4 provides preliminary information that connects DDs with DP models applied to a single truck and a single drone case, as well as the set partitioning formulation which forms the basis of our approaches. Section 5 describes our DD-based route relaxations and techniques to refine those relaxations. Section 6 describes alternative approaches for solving the DD-based master problem and Section 7 describes our iterative frameworks in detail. We report the experimental evaluation in Section 8, and present a discussion in Section 9.

2. Related Work

In this section we first review route relaxations that are used in BCP methods for VRPs. We then provide relevant background on decision diagrams. Lastly, we present related work on the traveling salesman problem with drone.

2.1. Pricing and Route Relaxation

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, Contardo, and Desaulniers (2019) for other important components in BCP algorithms, such as cutting and branching.

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, Desrosiers, and Solomon (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, Mingozzi, and Toth (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, Mingozzi, and Toth 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, Lessard, and Hadjar (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, Mingozzi, and Roberti (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.

2.2. Decision Diagrams

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, van Hoeve, and Hooker 2011, Bergman et al. 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, Cire, and van Hoeve 2017), or a stand-alone exact solver for combinatorial optimization problems (Bergman et al. 2014a, 2016b, Castro, Cire, and Beck 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.

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, 2022), which is used to computer lower bounds and refine relaxed DDs for graph coloring problems. Bergman, Cire, and van Hoeve (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 (2019) and Castro, Cire, and Beck (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, 2022).

We evaluate our proposed methods on the TSP-D, which we review next.

2.3. Traveling Salesman Problem with Drone

The TSP-D 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, Sah, and Lee (2020). In general, the hybrid truck and drone model should capture the following three relations between the two vehicles: parallelization where each vehicle may move independently; containment where the truck may carry the drone and synchronization where the truck may have to wait for the drone, or vice versa. Altogether this makes modeling and solving drone-assisted problems quite challenging.

Such a 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, Bouman, and Schmidt (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, Golden, and Wasil (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, Agatz, and Schmidt (2018), which can optimally solve instances with up to 15 customers. Vásquez, Angulo, and Klapp (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 (2021) 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, Poikonen, and Golden (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, Wang, and Golden (2017) extend Wang, Poikonen, and Golden (2017) to different cases by incorporating cost, limited battery life and different metrics respectively.

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 is defined as $V = \{0, 0'\} \cup N$ 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^T_{ij} and t^D_{ij} 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. $t^D_{ij} \leq t^T_{ij}, \forall (i, j) \in A$. 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.

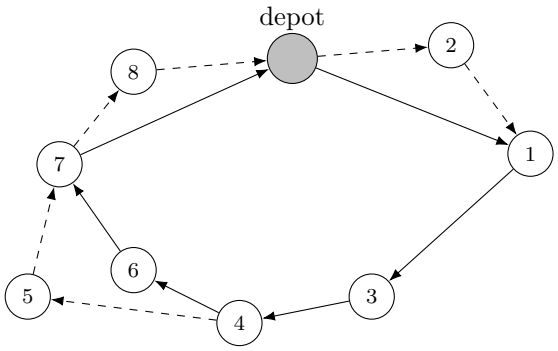


Figure 1 A feasible solution to a TSP-D instance with eight customers.

EXAMPLE 1. Figure 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 $t^T_{01} > t^D_{02} + t^D_{21}$), or the drone waits for the truck (when $t^T_{01} < t^D_{02} + t^D_{21}$). 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:

$$t = \max\{t^T_{01}, t^D_{02} + t^D_{21}\} + t^T_{13} + t^T_{34} + \max\{t^T_{46} + t^T_{67}, t^D_{45} + t^D_{57}\} + \max\{t^T_{70'}, t^D_{78} + t^D_{80'}\}. \tag{1}$$

In the remainder of this paper, we adopt the same notations used by Roberti and Ruthmair (2021). 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 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 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. An *operation* is a synchronized pair of a truck leg and

a drone leg in between the same pair of 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 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 $(1 \to 3 \to 4)$. Figure 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 1 can be represented as $([0 \to 1, 0 \dashrightarrow 2 \dashrightarrow 1], 1 \to 3 \to 4, [4 \to 6 \to 7, 4 \dashrightarrow 5 \dashrightarrow 7], [7 \to 0', 7 \dashrightarrow 8 \dashrightarrow 0'])$.

Based on the above definitions, a route can be formally defined as follows:

DEFINITION 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.

4. Preliminaries

In this section, we describe a dynamic programming (DP) model for the TSP-D and give a formal definition of decision diagrams (DDs). 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.

4.1. Dynamic Programming Model

We slightly modify the DP model used by Roberti and Ruthmair (2021) 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 paper. The model is based on the idea that each TSP-D solution can be decomposed into a set of truck arcs, drone legs, and combined arcs (as illustrated in Example 2). 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. We define $\Gamma(A, \omega)$ 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 $\gamma(A, C_j) = t^T_{i^T j}$;

   - 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 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 1. As a special case, it is allowed to add a drone leg when $i^C = i^T$, i.e., when the drone is parked on the truck. This corresponds to the case where the truck stays still while the drone visits its targeted customer before returning to the truck.

REMARK 2. 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 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.

REMARK 3. The above DP model can be easily extended if the drone is allowed to carry multiple packages and has a weight limit: one needs to extend the DP to include a new type of control and the state space to include the weight capacity of the drone.

Lastly, we define the cost function of $f(A)$ for state A recursively as

$$f(A) = \min_{(A',\omega):\Gamma(A',\omega)=A} f(A') + \gamma(A',\omega) \tag{2}$$

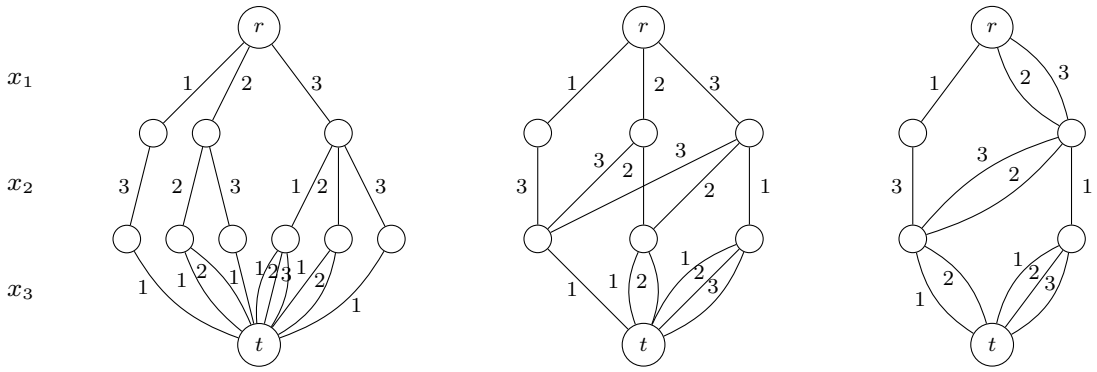
and we initialize the cost of the root state as $f(\{0\},0,0,0) = 0$. In other words, for state $A = (S, i^C, i^T, \tau)$, $f(A)$ represents the minimum duration of any partial TSP-D solution that starts from the depot and visits the set of customers S. Based on this and the observation that $(N \cup \{0,0'\}, 0', 0', 0)$ is the only terminal state (Remark 2), it follows immediately that:

LEMMA 1. *Value $f(N \cup \{0,0'\}, 0', 0', 0)$ is equal to the minimum completion time of the TSP-D.*

4.2. Basic Definitions of Decision Diagrams

We next introduce relevant concepts and definitions of decision diagrams for discrete optimization, following (Bergman et al. 2016a). For our purposes, a decision diagram 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 by $\text{Sol}(P)$.

A *decision diagram* for P is a layered weighted 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 s in layer



a. An exact DD b. The reduced exact DD c. A relaxed DD

Figure 2 Decision diagrams for the problem in Example 4. Figure (a) depicts an exact decision diagram representing all solutions. Figure (b) depicts the reduced exact decision diagram in which all equivalent nodes have been merged. Figure (c) depicts a relaxed decision diagram in which some non-equivalent nodes have been merged.

j to a node in layer $j+1$ and has an associated label $l(a)$ to denote the value of decision variable x_j. The weight of arc a is denoted by w_a. Each arc, and each node, must belong to a path from r to t. Each arc-specified r-t path $Q=(a_1,a_2,\ldots,a_m)$ defines a variable assignment X_Q by letting $x_j=l(a_j)$ for $j=1,\ldots,m$. We let $c(X_Q)$ denote the cost of solution X_Q to optimization problem P and let $w(Q):=\sum_{a\in Q} w_a$ denote the total weight of path Q. We slightly abuse the notation and denote by $\mathrm{Sol}(D)$ the collection of variable assignments defined by all r-t paths in D.

Definition 2. For a minimization problem P, a decision diagram D is *exact* if $\mathrm{Sol}(D)=\mathrm{Sol}(P)$ and $w(Q)=c(X_Q)$ for all arc-specified r-t paths Q. A decision diagram D is *relaxed* for problem P if $\mathrm{Sol}(D)\supseteq\mathrm{Sol}(P)$ and $w(Q)\leq c(X_Q)$ for all arc-specified r-t paths.

As a consequence of Definition 2, a shortest r-t path in an exact decision diagrams corresponds to an optimal solution to P. Likewise, a shortest r-t path in a relaxed decision diagram yields a lower bound on the optimal objective value (Bergman et al. 2016a).

We define the set of *completions* of a node $u\in N_D$ as the set of partial solutions corresponding to all u-t paths and their associated weight. Two nodes $u,v\in N_D$ are called *equivalent* if they have the same set of completions. Equivalent nodes can be merged, yielding a more compact representation of the solution set. A decision diagram is called *reduced* if no two nodes in a layer are equivalent, i.e., all equivalent nodes have been merged.[2] 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.

[2] For each problem and a given variable ordering, there exists a unique reduced decision diagram (Wegener 2000).

EXAMPLE 4. Let $X = \{x_1, x_2, x_3\}$ be an ordered set of variables, with domains $x_i \in \{1, 2, 3\}$ for $i = 1, 2, 3$. Consider the problem $\{\min \sum_{i=1}^{3} x_i \mid x_1 + x_2 \geq 4; x_2 + x_3 \leq 4\}$. Figure 2.a represents the set of all solutions in an exact decision diagram. In Figure 2.b, all equivalent nodes have been merged, resulting in the reduced decision diagram. A shortest r-t path in this diagram, e.g., $(x_1, x_2, x_3) = (1, 3, 1)$, yields an optimal solution with objective value 5. Figure 2.c depicts a smaller relaxed decision diagram in which some non-equivalent nodes have been merged. All solutions are still present, but the diagram now also includes non-solutions. In particular, the shortest path $(x_1, x_2, x_3) = (2, 1, 1)$ yields a lower bound of value 4.

4.3. DD Compilation Based on DP for TSP-D

Bergman et al. (2016a) developed a generic DD compilation scheme based on a dynamic programming formulation, analogous to the *state transition graph* of a DP model. The main idea is to associate each node in the DD with a state in the DP formulation. Using the DP model for the TSP-D from Section 4.1 as input, we start by associating the root node r of the DD with the initial state $(\{0\}, 0, 0, 0)$. We then recursively define the remainder of the DD as follows. First, we let $\theta(A)$ be the set of *feasible controls*, i.e., those that can be applied to A. For a state A that corresponds to a node of the DD and a control $\omega \in \theta(A)$, we define an arc from that node to the node corresponding to $\Gamma(A, \omega)$. The arc has label ω and weight equal to the transition cost $\gamma(A, \omega)$. Nodes with the same associated state are merged. Observe that each layer (except the terminal layer) corresponds to a stage of the recursion.

We convert the above recursive definition into an algorithm, presented in Algorithm 1. To represent the decision diagram, we use a vector of $n + 2$ 'layers' $L_1, \ldots, L_{n+2}$. The data structure for each layer is an unordered map, i.e., it 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 denotes 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 our purposes, a key in each layer is a state in the DP model and a value is the decision diagram 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 again an unordered map, where keys are arc labels and values are (arc length, arc head) pairs.

Algorithm 1: Exact DD compilation

```
1  create root layer L_1 and root node u_r
2  u_r.A ← ({0}, 0, 0, 0)
3  L_1 ← {u_r.A, u_r}
4  for k = 1, ..., n+1 do
5      Initialize layer L_{k+1}
6      for all u ∈ L_k and ω ∈ θ(u.A) do
7          A' ← Γ(u.A, ω)
8          if A' ∉ L_{k+1}.keys then
9              create node u'
10             u'.A ← A'
11             L_{k+1}.insert({A', u'})
12             u.arcs.insert({ω, (u', γ(A, ω))})
```

Algorithm 1 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).

LEMMA 2. *Let D be the decision diagram compiled by Algorithm 1 for a TSP-D instance P. There is a one-to-one correspondence between a root-terminal path in D and a feasible route for P, and moreover D is exact.*

Proof. By construction, an arc label represents a control ω in the DP model. Each arc-specified r-t path $(a_1, \ldots, a_{n+1})$ in D thus corresponds to a sequence $(l(a_1), \ldots, l(a_{n+1}))$ of operations and combined legs. Because the algorithm only uses feasible DP transitions to compile D (lines 6-7), each sequence represents a feasible solution to P. Conversely, for each solution to P represented in D there exists an associated sequence of DP transitions, by definition of the dynamic program. Lastly, because the arc weights are defined using the DP cost function $\gamma(A, \omega)$ (line 12), the weight of each r-t path corresponds to the cost of the associated solution, and D is exact. Q.E.D.

4.4. Lower Bound from Set Partitioning

Our framework utilizes the *set partitioning* (SP) formulation, where an element represents a customer and a set represents a route (Definition 1). Let R be a valid route relaxation for the TSP-D, 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. The binary

variable z_r represents whether r is chosen ($z_r = 1$) or not ($z_r = 0$). The set partitioning formulation is defined as follows:

$$\min \sum_{r \in R} c_r z_r \tag{3}$$

$$\text{s.t.} \sum_{r \in R} z_r = 1 \tag{4}$$

$$\sum_{r \in R} a_{ir} z_r = 1 \qquad \forall i \in N \tag{5}$$

$$z_r \in \{0,1\} \qquad \forall r \in R \tag{6}$$

where constraints (4) ensure that exactly one route is selected, constraints (5) ensure that each customer is visited exactly once, and constraints (6) are integrality constraints.

A feasible integer solution to model (3)-(6) corresponds to a feasible route. However, solving the set partitioning problem is NP-hard in general. A common approach is to solve the continuous linear programming (LP) relaxation instead, to obtain a lower bound and use it in a branch-and-bound framework. For notational ease, we denote this bound 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 (2021) use column generation to solve the LP, 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 to use the *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 5.1 for more details).

5. Route Relaxation

First we briefly describe the ng-route relaxation for the TSP-D by Roberti and Ruthmair (2021) and explain how 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.

5.1. ng-Route Relaxation

Recall from Section 2.1 that the ng-route relaxation uses a neighborhood $N_i \subseteq N$ for each location i, typically representing the locations closest to i (Baldacci, Mingozzi, and Roberti 2011). The neighborhoods are used to restrict the possible subtours for each route; the larger the size of the neighborhoods, the stronger the ng-route relaxation. When $N_i = N$, no subtours are allowed.

Roberti and Ruthmair (2021) adapt their DP model for the TSP-D (see Section 4.1) to the ng-route relaxation. The state definition is updated by replacing the set S of forbidden customers to be a new set ng, and adding a new parameter k representing the number of customers visited thus far. This results in the tuple (ng, k, i^C, i^T, τ). The DP model is updated by replacing the occurrence of S appropriately and adapting the transition function as follows:

- in conditional statements, a reference to $|S|$ is replaced by k, and a reference to S is replaced by ng;
- the state transition function $\Gamma(A, \omega)$ where $\omega \in \{T_j, D_j, C_j\}$ is updated to

$$\Gamma((ng, k, \cdot, \cdot, \cdot), \omega) = ((ng \cup \{j\}) \cap N_j, k+1, \cdot, \cdot, \cdot),$$

leaving the last three elements of the state transition unchanged. The DP cost function $\gamma(A, \omega)$ is also unchanged. As a result, we can compile a DD in a similar fashion as Algorithm 1. Let us call it D_{ng}. We have:

PROPOSITION 1. *D_{ng} is a relaxed DD for the TSP-D.*

Proof. By definition of the ng-set, the set of feasible routes is contained in the set of ng-routes. Furthermore, by definition of the cost function $\gamma(A, \omega)$ the weight of each r-t path in D_{ng} is equal to the length of the associated route. Therefore D_{ng} is a relaxed DD by Definition 2. Q.E.D.

We note that BCP algorithms do not explicitly construct D_{ng} to solve the pricing problem. Instead they solve the DP directly, and rely heavily on dominance rules to reduce the number of states to examine. More details can be found in Roberti and Ruthmair (2021).

5.2. DD-based Route Relaxation

In light of Proposition 1, we next investigate how to construct relaxed DDs that are different from D_{ng} and that can potentially yield stronger lower bounds. 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 4.3. Additionally, for relaxed DDs one should also have a rule describing how to *merge* nodes and perhaps how to adjust transition costs 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 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 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 4. 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 4.1). In particular, the larger τ becomes, the smaller the transition cost becomes. Therefore we define τ of state A' as $\max_{i=1,\ldots,q} \tau_i$. In summary, the merging operation is defined as:

Definition 3 (Merging Operation). A given collection of states $A_1 = (S_1, i^C, i^T, \tau_1), \ldots, A_q = (S_q, i^C, i^T, \tau_q)$ is merged into a single state via the 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 a relaxed DD D 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 5, it may happen that $\gamma_D(P) < c_{r(P)}$.

Example 5. Consider a TSP-D instance with four customers $\{1, 2, 3, 4\}$ and depot 0, and (symmetric) distances $t_{04}^T = t_{42}^T = 2, t_{03}^T = 3$ while all other arcs have distance 1. Assume that the truck and drone have equal speed. Consider the following two routes: $r_1 = (T_1, T_2, D_4, C_3, C_{0'})$ and $r_2 = (T_3, T_2, D_4, C_1, C_{0'})$. 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 Definition 3, this yields 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 pass 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.
The following result follows immediately from the discussion above:

Lemma 3. *Application of the merging operation produces a relaxed decision diagram.*

For a TSP-D instance, Lemma 3 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 8.1, however, this initial bound is typically not competitive with the state-of-the-art bound from the ng-route relaxation. To bridge this performance gap, we further refine the DD as described below in Section 5.3.

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 5.2, and P, a root-terminal path with a conflict which we wish to refine. We defer the discussion on how to find such paths to Section 6.

Our techniques are very similar to constraint separation in (van Hoeve 2020, 2022). 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 5 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 an 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 $j \le j' < k' < k$. 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.

Example 6. Recall in Example 5 we obtained a merged state $A' = (\{0,2\},0,2,4)$ by merging state $A_1 = (\{0,1,2\},0,2,2)$ and $A_2 = (\{0,2,3\},0,2,4)$. Recall A_1 and A_2 are intermediate states of route $r_1 = (T_1,T_2,D_4,C_3,C_{0'})$ and $r_2 = (T_3,T_2,D_4,C_1,C_{0'})$ respectively. By taking a control that visits customer 1 from A', we can create a path $(T_1,T_2,T_1,T_3,T_{0'})$ with a repetition conflict. Next we show a path with an inexact distance conflict: $(T_1,T_2,D_3,C_4,C_{0'})$. To see why it has an inexact distance conflict, consider the cost of taking control D_3 from the merged state A': $\max\{0, t^D_{03} + t^D_{31} - 4\} = 0$. However the actual cost of taking D_3 after T_1 and T_2 is $\max\{0, t^D_{03} + t^D_{31} - 2\} = 2$.

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. In these algorithms, we use the same data structures to define the DD as in Section 4.3. Function refineRep (Algorithm 2) 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 corresponding control 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 2: 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

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

Algorithm 3: 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

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

LEMMA 4. *Given a relaxed decision diagram D as input, the application of Algorithm 2 or Algorithm 3 results in a relaxed decision diagram.*

Proof. Algorithm 2 eliminates a path with a repetition conflict from D but does not introduce new paths. It does not change the cost function. Algorithm 3 similarly eliminates a path (with the inexact distance conflict) by redirecting the arc according to the actual transition cost. No new paths are introduced; if the arc is redirected to an already existing state, by the definition of state equivalence the paths through this node already existed in D. Q.E.D.

6. Lower Bound Computation

In the remainder of this paper, 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 $D = (V_D, A_D)$.

6.1. Constrained Flow Model

We construct a network flow model with side constraints where the network is D, similar to (van Hoeve 2020, 2022). 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 1 if a is chosen and 0 otherwise. The constrained flow model is defined as follows:

$$\min \quad \sum_{a \in A_D} l_a y_a \tag{7}$$

$$\text{s.t.} \quad \sum_{a \in \delta^+(u)} y_a = \sum_{a \in \delta^-(u)} y_a, \quad \forall u \in V_D, u \neq r, t \tag{8}$$

$$\sum_{a\in\delta^+(r)} y_a = 1 \tag{9}$$

$$\sum_{a\in\delta^-(t)} y_a = 1 \tag{10}$$

$$\sum_{l(a)\in\xi(i)} y_a = 1, \quad \forall i \in N \tag{11}$$

$$y_a \in \{0,1\}, \quad \forall a \in A_D \tag{12}$$

where constraints (8) (9) (10) ensure flow conservation on intermediate nodes, out-flow from the root and in-flow into the terminal, respectively. Constraint (11) 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 yields a lower bound for the TSP-D, since its arc length can be underestimated due to the construction process of the underlying relaxed DD. However, in general it is NP-hard to solve the constrained flow model. To derive a lower bound, we relax integrality constraints (12). We refer to this linear relaxation as CFLP(D). We can solve CFLP(D) by an off-the-shelf LP solver.

6.2. Lagrangian Relaxation

Another way of deriving lower bounds is to apply Lagrangian relaxation to the above constrained flow model by dualizing constraints (11). 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:

$$\psi_D(\boldsymbol{\lambda}) = \min \quad -\sum_{i\in N}\lambda_i + \sum_{a\in A_D}(l_a + \sum_{i\in N}\{\omega(a)\in\xi(i)\}\lambda_i)y_a$$
$$y \text{ subject to constraint (8) (9) (10)}$$

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:

$$\max_{\lambda} \psi(\boldsymbol{\lambda}) \tag{13}$$

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:

$$\boldsymbol{\lambda}^{(t+1)} = \boldsymbol{\lambda}^{(t)} + \alpha_t \boldsymbol{g}^{(t)}, \tag{14}$$

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 $s_i - 1$. Our step size strategy can be found in the online supplement (Appendix A).

6.3. 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), the LP bound from the set partitioning formulation. For notational ease, let $v(P)$ denote the optimal value of an optimization problem P. We show that

THEOREM 2. *$v(SPLP(R_D)) = v(CFLP(D)) = v(LR(D))$.*

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 polyhedron. Q.E.D.

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 8.3. 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 7.

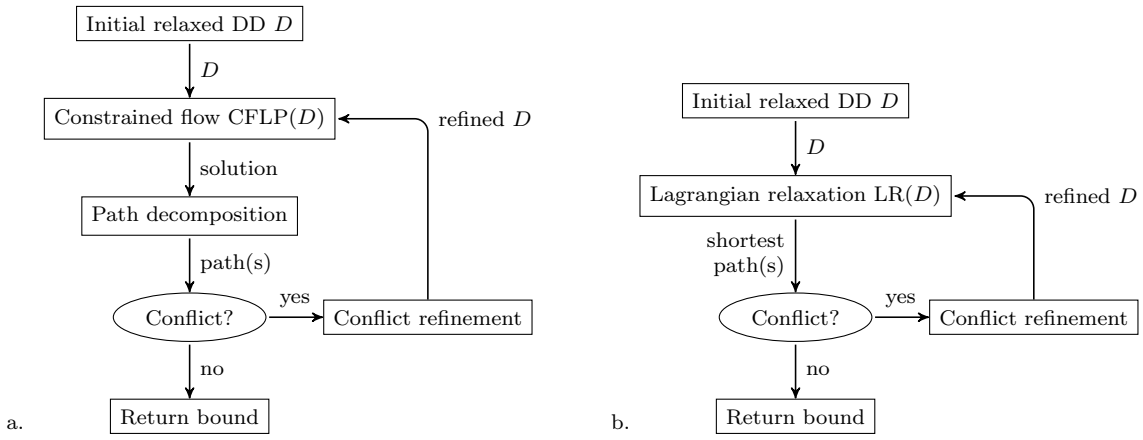


Figure 3 Overview of our iterative framework using either (a) the constrained network flow linear program or (b) the Lagrangian relaxation.

7. Iterative Framework

In this section, we describe two iterative frameworks based on the constrained flow model (CFLP) and Lagrangian relaxation (LR) respectively. Both frameworks follow the same pattern as shown in Figure 3: We start from an initial route relaxation represented as a decision diagram. Then we iteratively solve the associated linear program using either CFLP (Fig. 3.a) or LR (Fig. 3.b), identify conflicts in the solution, and strengthen the relaxation by refining the decision diagram.

Throughout this section, we assume that a relaxed DD D is initially compiled according to the merging rule in Section 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 8.

7.1. Combining 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 4). 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 5) 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

Algorithm 4: Function flowRefine(D): iterative refinement with CFLP

Input: relaxed decision diagram D

Output: refined decision diagram

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

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 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 update the residual flow on each arc in this path (line 11 - line 12).

Function refinePath (Algorithm 6) 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 the implementation are discussed in the online supplement (Appendix A).

7.2. Combining 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 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 7) 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. 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

Algorithm 5: 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
```

Algorithm 6: Function refinePath(D, P): refine conflicts along a prescribed path

Input: relaxed decision diagram D, a prescribed path P

Output: refined decision diagram

```
1  i ← findInexactDist(P)
2  if i ≠ −1                                                // i = −1 means no such conflict exists
3  then
4      refineInexactDist(D, P, i)
5  (j, k) ← findRep(P)
6  if k ≠ −1                                                // k = −1 means no such conflict exists
7  then
8      refineRep(D, P, j, k)
```

of the latest κ iterations, the improvement percentage of the best lower bound is below μ. 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 the online supplement (Appendix A).

Function lagRestart (Algorithm 8) restarts the subgradient method periodically according to the *buffer size* H chosen a priori. 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,

Algorithm 7: Function lagAdapt(D): iterative refinement that adaptively terminates the subgradient method

Input: relaxed decision diagram D

Output: refined decision diagram

```
1  λ ← 0
2  bestBound ← 0
3  bestMultipliers ← λ
4  while termination criterion is not met do
5      while improvement criterion holds do
6          (P, lb) ← findShortestPath(D, λ)
7          if bestBound < lb then
8              bestBound ← lb
9              bestMultipliers ← λ
10         updateMultipliers(λ)
11         refinePath(D, P)
12     (P, lb) ← findShortestPath(D, bestMultipliers)
13     refinePath(D, P)
```

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

8. Computational Experiments

We implemented our iterative refinement algorithms in C++ and performed an 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 $\alpha \in \{2, 4\}$.

Algorithm 8: 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
```

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 proposed by Tang, van Hoeve, and Shaw (2019) by CP Optimizer with a time limit of one hour. Best lower bounds in the literature are computed by the ng-route-based BCP algorithm (Roberti and Ruthmair 2021). 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 ng-route column generation to solve the root LP. In our implementation, column generation is terminated when the 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.

8.1. 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 5.2. 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 (2021), we set the neighborhood size $|N_i| = 5$ for all $i \in N$ in our ng-route relaxation.

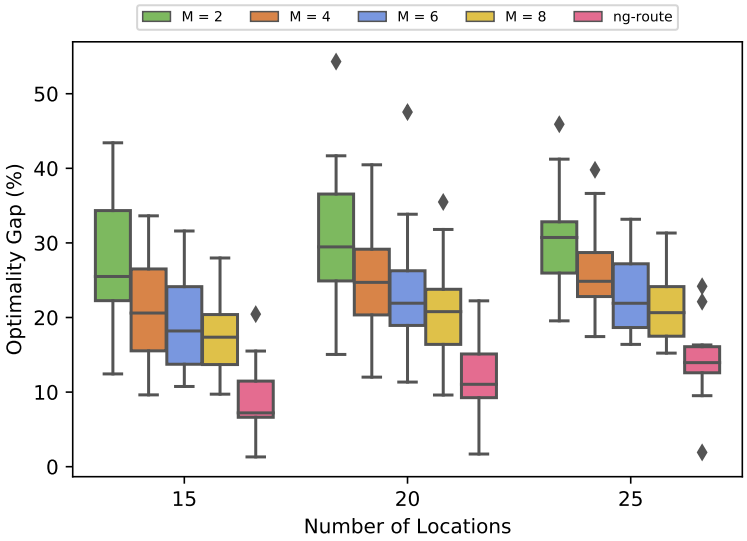


Figure 4 **Optimality gap (%) of the initial DD route relaxation and the ng-route relaxation. M denotes the maximum bucket size used in the merging process.**

The lower bound from our route relaxation is computed by solving CFLP(R_D). The lower bound from the ng-route relaxation R_{ng} is SPLP(R_{ng}) computed via CG. In our implementation, column generation exceeds the time limit for all instances with $m \geq 30$. We report our findings for instances with $m \in \{15, 20, 25\}, \alpha \in \{2, 4\}$, and bucket sizes $M \in \{2, 4, 6, 8\}$.

	Number of locations		
Route relaxation	15	20	25
relaxed DD with $M = 2$	4,974	12,734	26,043
relaxed DD with $M = 4$	9,736	25,086	51,485
relaxed DD with $M = 6$	14,498	37,438	76,928
relaxed DD with $M = 8$	19,260	49,790	102,370
ng-route	24,114	73,554	174,725

Table 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 DP labels after dominance rules are applied.**

Table 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 DP 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.

Figure 4 represents the optimality gaps for the different route relaxations, where the horizontal axis represents the number of locations and the vertical axis represents the optimality gap (%). The

boxplots are divided into three groups based on the number of locations. Within each group, we vary bucket size M and compare with the ng-route relaxation. Figure 4 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 4 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.

8.2. Lower Bound Improvement

Recall that all of our iterative algorithms in Section 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 for SPLP(R_{ng}), 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. The online supplement (Appendix A) details how the parameters for our proposed algorithms (flowRefine, lagAdapt, lagRestart) are set.

Figure 5 shows bound improvement over time for flowRefine, lagAdapt and lagRestart. In the figure, the horizontal axis represents the runtime of each algorithm measured in percentages w.r.t T_{CG} and the vertical axis represents the average optimality gap (%). As we go from left to right along the horizontal 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 the ng-route relaxation is shown as a solid line starting from $100\% \cdot T_{\text{CG}}$.

Figure 5 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

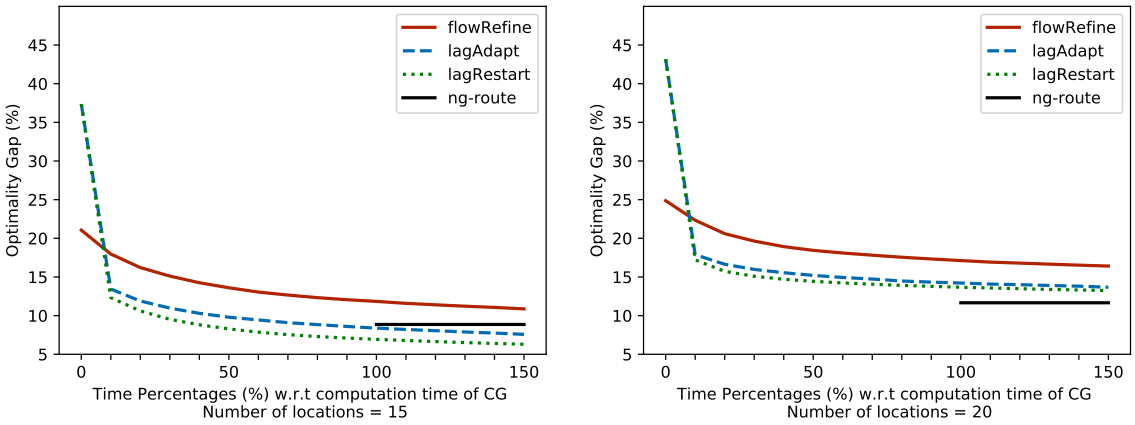


Figure 5 **Optimality gap (%) over time for flowRefine, lagAdapt and lagRestart. Each interval on the horizontal axis represents 10% of CG (ng-route) computation time. The optimality gap from the ng-route relaxation appears after 100%.**

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.

8.3. 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 $m \geq 30$. The only other source of lower bounds available in the literature is the mixed integer programming (MIP) model proposed by Roberti and Ruthmair (2021). Their original MIP model uses big-M constraints which results in relatively weak 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. Details on this model can be found in the online supplement (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. Recall that the online supplement (Appendix A) details how parameters for our proposed algorithms (flowRefine, lagAdapt, lagRestart) are set.

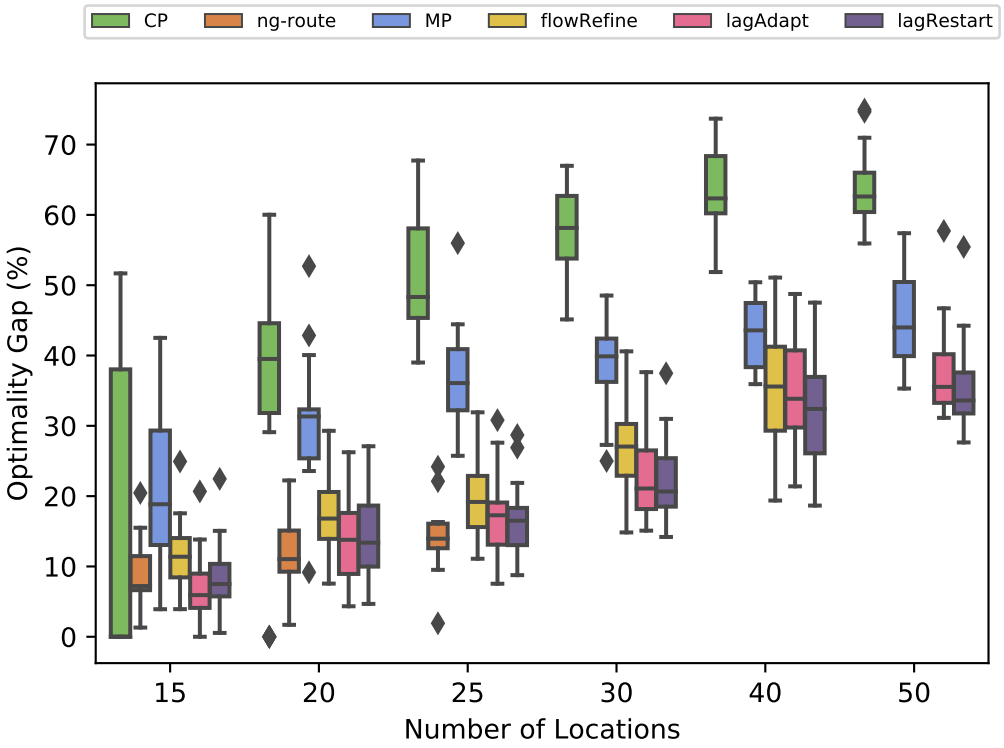


Figure 6 **Optimality gap (%) from different algorithms, where labels CP, ng-route, MP denote the optimality gap achieved by the constraint programming model, the column generation approach for the ng-route relaxation and the modified MIP model, respectively. We set the time limit of our algorithm as the computation time of CG, with a time limit of one hour. CG does not terminate in one hour for $m \geq 30$.**

In Figure 6, the horizontal axis indicates the number of customers and the vertical 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 6 shows that lagAdapt and lagRestart are competitive to CG when $m \leq 25$. flowRefine is slightly worse. When $m \geq 30$, CG does not 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.

Route relaxation	Number of locations 15	20	25
initial relaxed DD	9,736	25,086	51,485
refined DD via flowRefine	10,725	26,538	51,967
refined DD via lagRestart	15,567	37,197	59,204
ng-route	24,114	73,554	174,725

Table 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.

REMARK 6. 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, we may integrate CG with conflict refinements in our future work.

Table 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 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.

8.4. Effect of Drone-Truck Speed Ratio

Lastly, 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 7, 8 and 9 shows the optimality gap for flowRefine, lagAdapt and lagRestart respectively, where the horizontal axis represents the number of locations ranging between 15 and 40 and the vertical axis represents the optimality gap (%). The speed ratio is 2 and 4.

Figure 7 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.

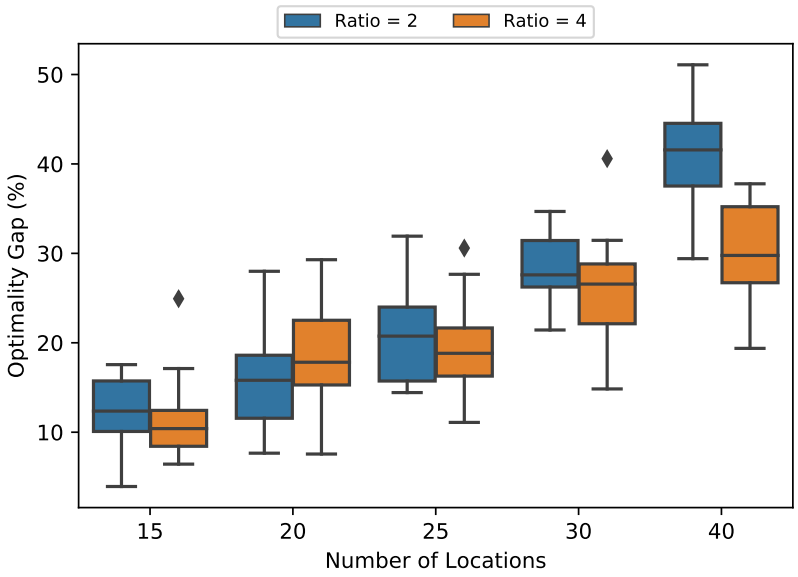


Figure 7 **Optimality gap from flowRefine w.r.t. the number of locations and drone-truck speed ratio. We set the time limit of our algorithm as the computation time of CG, with a time limit of one hour. CG does not terminate in one hour for $m \geq 30$.**

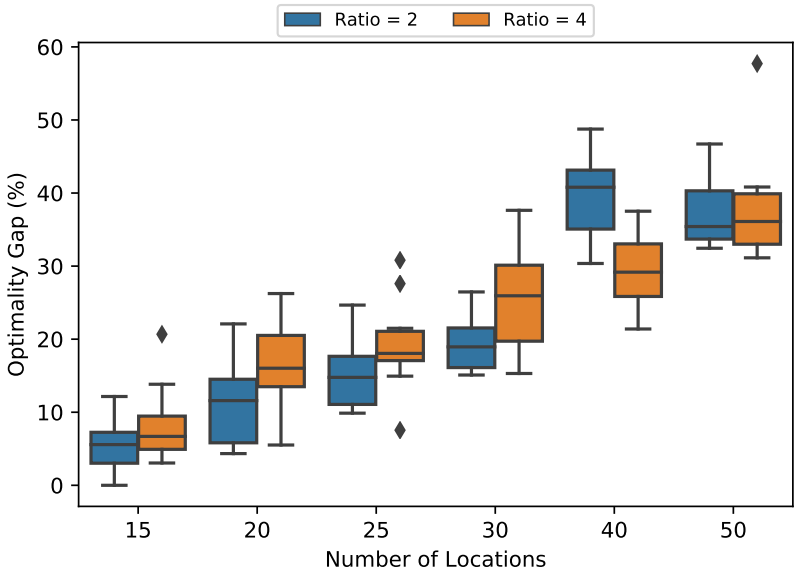


Figure 8 **Optimality gap from lagAdapt w.r.t. the number of customers and drone-truck speed ratio. We set the time limit of our algorithm as the computation time of CG, with a time limit of one hour. CG does not terminate in one hour for $m \geq 30$.**

Figure 8 and 9 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

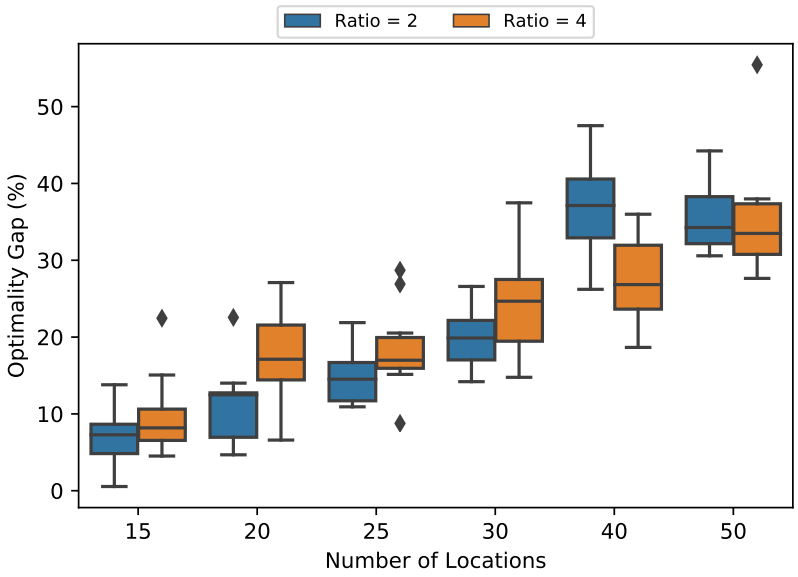


Figure 9 **Optimality gap from lagRestart w.r.t. the number of customers and truck-drone speed ratio. We set the time limit of our algorithm as the computation time of CG, with a time limit of one hour. CG does not terminate in one hour for $m \geq 30$.**

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.

9. Discussion

This section discusses several components (dominance rules, branching and cutting) that makes a traditional BCP algorithm successful. As we mentioned in Section 1, one of the advantages of incorporating DD-based dual bounds into a BCP algorithm is that a lower bound is always available from the relaxed model, instead of having to solve potentially many iterations of column generation. Below we discuss how other components of a BCP algorithm come into play when we incorporate DD-based dual bounds.

9.1. Dominance Rules

During pricing, route relaxations are typically combined with dominance rules to limit the size of the state space. The idea of dominance rules is to prune a node as early as possible when it can be detected that optimal solutions with respect to the underlying route relaxation will not traverse that node. As a consequence, a 'true' optimal solution with respect to the original problem may be discarded when dominance rules are applied. Indeed, consider the following example:

EXAMPLE 7. Consider the complete graph on four nodes $\{0, 1, 2, 3\}$ representing a parallelogram, given in Fig. 10, with truck traveling time t_{ij}^T labeled on arcs representing Euclidean distances,

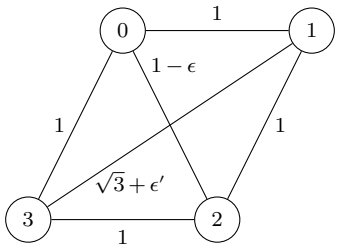


Figure 10 An example of optimal solutions being discarded after applying dominance rules, per Example 7.

and speed differential $\alpha = 1 - \epsilon$ ($\epsilon > 0$ is a fixed parameter close to 0, and therefore $\epsilon' > 0$ is also close to 0 due to the geometric properties). Consider the following three states on the same layer: $A = (\{0,1,2\},1,1,0), B = (\{0,1,3\},1,1,0), C = (\{0,1,2\},2,2,0)$ which correspond respectively to the partial routes $([0 \to 1, 0 \dashrightarrow 2 \dashrightarrow 1]), ([0 \to 1, 0 \dashrightarrow 3 \dashrightarrow 1]), ([0 \to 2, 0 \dashrightarrow 1 \dashrightarrow 2])$. Recall $f(\cdot)$ is the minimum duration of any partial route arriving at that particular state. So $f(A) = (2-\epsilon)/(1-\epsilon), f(B) = (\sqrt{3}+1+\epsilon')/(1-\epsilon), f(C) = 2/1-\epsilon$. Now if we merge A with B we have $A \oplus B = (\{0,1\},1,1,0)$ and $f(A \oplus B) = (2-\epsilon)/(1-\epsilon)$. By definition of our dominance rule, state $A \oplus B$ dominates state C and is thus pruned, which results in two optimal solutions $([0 \to 2, 0 \dashrightarrow 1 \dashrightarrow 2], [2 \to 0, 2 \dashrightarrow 3 \dashrightarrow 0])$ and $([0 \to 2, 0 \dashrightarrow 3 \dashrightarrow 2], [2 \to 0, 2 \dashrightarrow 1 \dashrightarrow 0])$ traversing state C being discarded.

We remark that the purpose of dominance rules is to speed up one iteration of the pricing process. As long as the route relaxation is valid, the corresponding set partitioning problem will output a valid lower bound regardless of whether optimal solutions are discarded during a particular iteration. In contrast, the purpose of DDs is to compactly store the set of feasible (in particular, optimal) solutions. As long as such a property is desired, dominance rules, at least as they are currently defined, cannot be applied.

9.2. Cutting Planes

It is important to distinguish the underlying MIP model to which we apply cuts, since we have presented two MIP models in this work, including the set partitioning (SP) model in Section 4.4 and the constrained flow model in Section 6.1.

For the set partitioning model, it is indeed possible to rewrite subtour elimination constraints which prevent the truck or the drone tour to form a cycle in the form of set variables. This will not increase the complexity of pricing because dual variables of such constraints can be handled during pricing by changing weights for truck/drone arcs. It is also possible to add subset-row inequalities (Jepsen et al. 2008). However, for the TSP-D, there is one and only one tour that will be selected. Therefore SR inequalities are redundant in this case. When the problem is extended to multiple trucks and drones, such inequalities may be useful and separation of such inequalties

need to be modified. Even though it is possible to add valid cuts to the SP model, we did not do so in this work because our focus is to investigate the quality of DD-based route relaxations. Adding such inequalities improves dual bounds from the SP model which may in turn improve the performance of a full BCP solver, it does not change the structure of route relaxations obtained from our relaxed DDs, because such constraints do not directly work on route relaxations.

For the constrained flow model, it is also possible to add subtour elimination constraints. However if there exists a subtour that is not allowed in the added constraints but is present in the underlying relaxed DD, the flow model may become infeasible. Therefore, the key question here is how to eliminate certain subtours in the underlying relaxed DD. We think this may be achieved in two ways – one is to apply conflict refinement as is done in this work and the other is to merge carefully so that certain subtours never appear in the merged DD. The latter approach poses interesting challenges to the merging operation, which we leave as future work.

9.3. Branching

Although we limit our focus in this paper to derive dual bounds for routing problems, it is rather straightforward to incorporate variable branching schemes such as the three-level hierarchical branching proposed in Roberti and Ruthmair (2021) with our DD-based approach because fixing variables to 1 or 0 corresponds to fixing/removing arcs in a relaxed DD. Since we keep a relaxed DD in memory throughout the solving process, solving the subproblem after branching is relatively quick if we do not perform too many rounds of iterative refinement. As for special branching schemes tailored for set partitioning models (Falkner and Ryan 1992, Barnhart et al. 1998), it is not useful for the TSP-D because all customers need to be visited in the selected tour exactly once. For multiple trucks and drones, such branching schemes can be very helpful, which we leave as future work.

10. Conclusion

In this work, we proposed novel route relaxations for vehicle routing problems (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 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 8.1. 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 2 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 BCP does not because the BCP 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. We also note that there are other motivating examples outside of drones where the interaction between two or more types of vehicles makes the problem more difficult to solve. For example, an emerging

application in industry exists in mobile food truck. Food trucks travel to serve meal preparation requests at locations nearby end customers, and replenishment vehicles travel to restock the food trucks. The replenishment to food truck relationship may be one-to-many or many-to-many. This closely resembles the structure we have here and is worth exploring in the future work.

References

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

Andersen HR, Hadzic T, Hooker JN, Tiedemann P, 2007 *A constraint store based on multivalued decision diagrams.* Bessiere C, ed., *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, 118–132 (Springer).

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

Barnhart C, Johnson EL, Nemhauser GL, Savelsbergh MW, Vance PH, 1998 *Branch-and-price: Column generation for solving huge integer programs. Operations research* 46(3):316–329.

Becker B, Behle M, Eisenbrand F, Wimmer R, 2005 *BDDs in a Branch and Cut Framework.* Nikoletseas SE, ed., *Experimental and Efficient Algorithms, 4th InternationalWorkshop, WEA 2005, Santorini Island, Greece, May 10-13, 2005, Proceedings*, volume 3503 of *Lecture Notes in Computer Science*, 452–463 (Springer).

Bergman D, Ciré AA, Sabharwal A, Samulowitz H, Saraswat VA, van Hoeve WJ, 2014a *Parallel combinatorial optimization with decision diagrams.* Simonis H, ed., *Integration of AI and OR Techniques in Constraint Programming - 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings*, volume 8451 of *Lecture Notes in Computer Science*, 351–367 (Springer).

Bergman D, Cire AA, van Hoeve WJ, 2015 *Lagrangian bounds from decision diagrams. Constraints* 20(3):346–361.

Bergman D, Cire AA, Van Hoeve WJ, Hooker J, 2016a *Decision diagrams for optimization*, volume 1 (Springer).

Bergman D, Ciré AA, van Hoeve WJ, Hooker JN, 2012 *Variable Ordering for the Application of BDDs to the Maximum Independent Set Problem.* Beldiceanu N, Jussien N, Pinson E, eds., *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems - 9th International Conference, CPAIOR 2012, Nantes, France, May 28 - June1, 2012. Proceedings*, volume 7298 of *Lecture Notes in Computer Science*, 34–49 (Springer).

Bergman D, Cire AA, Van Hoeve WJ, Hooker JN, 2016b *Discrete optimization with decision diagrams. INFORMS Journal on Computing* 28(1):47–66.

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

Bergman D, van Hoeve WJ, Hooker JN, 2011 *Manipulating MDD relaxations for combinatorial optimization.* Achterberg T, Beck JC, eds., *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems - 8th International Conference, CPAIOR 2011, Berlin, Germany, May 23-27, 2011. Proceedings*, volume 6697 of *Lecture Notes in Computer Science*, 20–35 (Springer).

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

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

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

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

Chung SH, Sah B, Lee J, 2020 *Optimization for drone and drone-truck combined operations: A review of the state of the art and future directions. Computers & Operations Research* 123:105004.

Cire AA, Van Hoeve WJ, 2013 *Multivalued decision diagrams for sequencing problems. Operations Research* 61(6):1411–1428.

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

de Freitas JC, Penna PHV, 2020 *A variable neighborhood search for flying sidekick traveling salesman problem. International Transactions in Operational Research* 27(1):267–290.

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

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

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

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

Falkner JC, Ryan DM, 1992 *Express: Set Partitioning for Bus Crew Scheduling in Christchurch.* Desrochers M, Rousseau JM, eds., *Computer-Aided Transit Scheduling*, 359–378 (Springer Berlin Heidelberg).

Ferrandez SM, Harbison T, Weber T, Sturges R, Rich R, 2016 *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.

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

Geoffrion AM, 1974 *Lagrangean relaxation for integer programming. Approaches to integer programming*, 82–114 (Springer).

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

Ha QM, Deville Y, Pham QD, Hà MH, 2018 *On the min-cost traveling salesman problem with drone. Transportation Research Part C: Emerging Technologies* 86:597–621.

Ha QM, Deville Y, Pham QD, Hà MH, 2020 *A hybrid genetic algorithm for the traveling salesman problem with drone. Journal of Heuristics* 26(2):219–247.

Hooker JN, 2019 *Improved job sequencing bounds from decision diagrams*. Schiex T, de Givry S, eds., *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings*, volume 11802 of *Lecture Notes in Computer Science*, 268–283 (Springer).

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

Jeon I, Ham S, Cheon J, Klimkowska AM, Kim H, Choi K, Lee I, 2019 *A real-time drone mapping platform for marine surveillance. The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* XLII-2/W13:385–391.

Jepsen M, Petersen B, Spoorendonk S, Pisinger D, 2008 *Subset-row inequalities applied to the vehicle-routing problem with time windows. Operations Research* 56(2):497–511.

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

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

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

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

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

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

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

Ponza A, 2016 *Optimization of drone-assisted parcel delivery.* Master's thesis, Università degli studi di Padova.

Roberti R, Ruthmair M, 2021 *Exact methods for the traveling salesman problem with drone. Transportation Science* 55(2):315–335.

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

Tang Z, van Hoeve W, Shaw P, 2019 *A Study on the Traveling Salesman Problem with a Drone.* Rousseau L, Stergiou K, eds., *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4-7, 2019, Proceedings*, volume 11494 of *Lecture Notes in Computer Science*, 557–564 (Springer).

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

Tjandraatmadja C, van Hoeve WJ, 2020 *Incorporating bounds from decision diagrams into integer programming. Mathematical Programming Computation* 1–32.

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

van Hoeve W, 2020 *Graph Coloring Lower Bounds from Decision Diagrams.* Bienstock D, Zambelli G, eds., *Integer Programming and Combinatorial Optimization - 21st International Conference, IPCO 2020, London, UK, June 8-10, 2020, Proceedings*, volume 12125 of *Lecture Notes in Computer Science*, 405–418 (Springer).

van Hoeve WJ, 2022 *Graph coloring with decision diagrams. Mathematical Programming* 192(1):631–674.

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

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

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

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