FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Solving the Multi-Objective Steiner Tree Problem with Resources

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Georg Brandstätter
Matrikelnummer 0825052

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.-Prof. Dipl.-Ing. Dr. Günther Raidl
Mitwirkung: Dipl.-Ing. Dr. Markus Leitner
Dipl.-Ing. Dr. Mario Ruthmair

Wien, January 22, 2015

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Solving the Multi-Objective Steiner Tree Problem with Resources

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Georg Brandstätter
Registration Number 0825052

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.-Prof. Dipl.-Ing. Dr. Günther Raidl
Assistance: Dipl.-Ing. Dr. Markus Leitner
Dipl.-Ing. Dr. Mario Ruthmair

Vienna, January 22, 2015 _____ _____
(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Georg Brandstätter
Laudongasse 56/15, 1080 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Danksagung

An erster Stelle möchte ich mich bei meinen Betreuern Günther Raidl, Markus Leitner und Mario Ruthmair für die großartige Betreuung während der gesamten Dauer meiner Diplomarbeit bedanken. Sie hatten stets Zeit für mich und meine Anliegen und halfen mir, alle auftretenden Probleme letztendlich erfolgreich zu bewältigen. Ohne ihr motivierendes und konstruktives Feedback wäre diese Arbeit nie zu dem geworden, was sie heute ist.

Weiters danke ich all meinen Freunden dafür, dass sie über die Jahre hinweg immer für mich da waren. Besonderer Dank gilt meinen Freunden und Studienkollegen Fabian, Johannes, Martin und Max, die mir darüber hinaus auch bei allen Schwierigkeiten im Studium zur Seite standen.

Abschließend möchte ich mich bei meiner Familie, allen voran bei meinen Eltern und Großeltern, dafür bedanken, dass sie mir mein Studium ermöglicht haben. Dank ihrer ununterbrochenen Unterstützung konnte ich mich immer voll auf das Studium selbst konzentrieren, ohne mir Sorgen um andere Aspekte meines Lebens machen zu müssen.

Abstract

Network design problems are an important class of combinatorial optimization problems, with applications ranging from the design of telecommunication networks to the planning of a city's street and power grid. One of these problems is the Steiner Tree Problem on Graphs (STP), a well-known NP-hard combinatorial optimization problem that consists in finding a subgraph of the given input graph that connects a given subset of its vertices, the set of *terminal* vertices, as cheaply as possible. In real-world problems, however, it is often important to consider further attributes when evaluating a solution.

To allow for the modeling of such problems, we define the Multi-objective Steiner Tree Problem with Resources, which is a multi-objective generalization of the STP. Given a set of resource demands associated with each edge, the problem not only seeks to minimize a solution's cost, but also the maximum of each resource's cumulative consumption along each path between the root and a terminal vertex.

We develop a series of algorithms for solving the bi-objective variant of this problem, the so-called Bi-objective Steiner Tree Problem with Delays. These algorithms use the ε-constraint method to decompose the bi-objective input instance into a series of instances of the single-objective Rooted Delay-constrained Steiner Tree Problem. To solve these, we encode them as integer linear programs according to the formulations developed for this problem, which are then solved by branch-and-cut. Since we only use exact methods, our algorithms compute the exact Pareto frontier of our original instance, given enough time and memory.

To improve the performance of our algorithms, we preprocess the subproblem graphs before each iteration. Additionally, we reuse information from previous iterations, such as optimal solutions and inequalities added by branch-and-cut, during subsequent ones. To enable the reuse of solutions that are no longer feasible for the next iteration, we develop a heuristic to transform them into feasible ones.

We test our implementations of the developed algorithms on a set of benchmark instances. These tests show that in addition to an instance's size, its structure (i.e., how an edge's cost and delay are determined) can have a significant impact on the time necessary to find its complete Pareto frontier. The tests also show that preprocessing and the reuse of information both have an often quite significant positive impact on the performance of our algorithms.

Finally, we describe how the aforementioned algorithms for the bi-objective case can be adapted to solve the multi-objective problem. We note, however, that the generalization towards multiple objectives introduces significant challenges, including the problem of finding a suitable starting point for the ε-constraint method, the large number of subproblem instances that need to be solved and the likely high difficulty of solving these instances.

Kurzfassung

Netzwerkentwurfsprobleme bilden eine Klasse von kombinatorischen Optimierungsproblemen, deren Anwendungsgebiete vom Entwurf von Telekommunikationsnetzwerken bis zur Planung von städtischer Infrastruktur reichen. Ein bekanntes Problem dieser Klasse ist das NP-schwere Steinerbaumproblem auf Graphen (STP), welches darin besteht, den kostengünstigsten Teilbaum des Eingangsgraphen zu finden, der alle Knoten aus der gegebenen Menge an Terminalknoten verbindet. In real auftretenden Problemen ist es jedoch oft nötig, Lösungen anhand mehrerer Gesichtspunkte zu bewerten.

Um die Modellierung solcher Probleme zu ermöglichen, definieren wir das Multikriterielle Steinerbaumproblem mit Ressourcen (MOSTPR), welches eine Generalisierung des einfachen STP auf mehrere Zielfunktionen ist, bei dem jeder Kante zusätzlich eine Menge an Ressourcen, die sie verbraucht, zugeteilt ist. Unsere zusätzlichen Ziele sind, den maximalen Gesamtverbrauch jeder einzelnen Ressource entlang der Pfade vom Wurzel- zu den Terminalknoten zu minimieren.

Zur Lösung der Problemvariante mit zwei Zielfunktionen (BOSTPD) entwickeln wir auf der ε-Constraint Methode basierende Algorithmen, welche die Instanz in Teilinstanzen mit nur einer Zielfunktion (RDCSTP) zerlegen, diese gemäß zweier ILP-Formulierungen codieren und mittels Branch-and-Cut lösen. Da die Algorithmen nur exakte Verfahren verwenden, können wir mit ausreichend Zeit und Speicher die vollständige Paretofront finden.

Um die Laufzeit der Algorithmen zu verbessern, entfernen wir vor Beginn jeder Iteration alle Knoten und Kanten, die nicht Teil einer optimalen Lösung sein können. Weiters verwenden wir Informationen aus vorhergehenden Iterationen, wie etwa deren optimale Lösung oder mittels Branch-and-Cut hinzugefügte Ungleichungen, zur Beschleunigung des Lösungsvorgangs.

Wir testen die Implementierung der zuvor erwähnten Algorithmen auf einer Reihe von Beispielinstanzen. Diese Tests zeigen, dass nicht nur die Größe einer Instanz, sondern auch ihre Struktur einen starken Einfluss auf die Laufzeit hat, die zum Berechnen der Paretofront nötig ist. Weiters zeigen wir, dass sowohl das Entfernen überflüssiger Knoten und Kanten als auch die Wiederverwendung von Information aus Voriterationen die benötigte Laufzeit stark reduzieren können.

Abschließend beschreiben wir, wie die von uns für das BOSTPD entwickelten Algorithmen modifiziert werden können, um das allgemeine Problem mit beliebiger Anzahl an Ressourcen zu lösen. Wir weisen dabei auch auf die zu erwartenden Probleme hin, die derartige Modifikationen mit sich bringen können, wie etwa die größere Anzahl an zu lösenden Teilproblemen und deren höhere Schwierigkeit.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The design of efficient networks is one of the most frequently encountered optimization problems. Whether we are designing telecommunication networks or planning a logistics company's supply routes, we want to find the overall cheapest network connecting all endpoints of interest. These problems can often be modeled as instances of the Steiner Tree Problem on Graphs (STP) [63], which is a well-studied combinatorial optimization problem (COP). Many of its instances can be solved to proven optimality within reasonable computing time with methods like *branch-and-cut* (see, e.g. [19, 35], as well as Section 2.2.2 of this thesis).

However, the simple STP can often fail to adequately model the complexity of real-life network design problems. Specifically, we can only consider one parameter of a connection (often its cost) and must disregard all others it might have. It would, for example, be impossible to design an optimal road network for a country with regard to both low construction costs and short travel times. We would have to choose between optimizing with respect to cost (and likely end up with a road network consisting solely of dirt roads) or travel time (which we probably would not be able to afford, since building motorways everywhere is prohibitively expensive), neither of which is what we actually want.

To overcome some of the shortcomings of the regular STP, Kompella et al. [36, 37] proposed an extension to the STP that introduces the concept of delay bounds. Here, in addition to the aforementioned costs, every edge has an associated delay and the cumulative delay along each path in the solution is bounded by an arbitrarily chosen integer parameter B. These edge delays can, for instance, be interpreted as travel times in cases of road networks or as transmission times in cases of telecommunication networks. While this allows us to enforce a certain quality of service, the problem of finding appropriate values for B remains. Setting it too high might not guarantee sufficient quality, whereas setting it too low might drive up costs too much.

It would obviously be best to find *all* optimal solutions to the STP with delays (i.e., to find a cheapest Steiner Tree for every possible delay bound). This would enable us to choose the one that represents the best compromise between the competing optimization objectives.

The main objective of this thesis is to present an algorithm that solves this bi-objective generalization of the STP with delays. Using the ε-constraint method, we split the bi-objective problem into several subsequent iterations of the single-objective STP with delay constraints, which are then solved using the methods proposed in [55]. We place special emphasis on the efficient reuse of information from previous iterations, such as (partial) solutions and added inequalities from the branch-and-cut procedure. Besides evaluating the speed-up obtained from this information reuse, we evaluate different formulations of the problem with respect to their performance in subsequent iterations.

While the aforementioned bi-objective variant is of great interest for evaluating the performance of our approach, the problem can easily be generalized to the multi-objective case, where each edge has multiple resource demands instead of just one (the delay). We therefore also given an explanation of how the algorithm can be, in principle, be generalized to solve problem instances with an arbitrary number of objectives.

1.2 Problem definition

An instance of the Multi-objective Steiner Tree Problem with Resources (MOSTPR) consists of a graph $G = (V, E)$, a cost function $c : E \to \mathbb{N}$ and a resource demand function $d : E \to \mathbb{N}^k$. The set of vertices $V = S \cup T \cup \{r\}$ is the disjoint union of the set of potential Steiner vertices S, the set of terminal vertices T and the dedicated root vertex r. The cost function c assigns a non-negative integer cost c_e to each edge $e \in E$. Similarly, the vector-valued resource demand function d assigns a k-dimensional resource demand vector $\mathbf{d}_e = (d_e^1, \ldots, d_e^k)$ with positive integer components to each $e \in E$. Component d_e^j defines the demand of resource j for edge e. An exemplary instance of the MOSTPR for $k = 1$ (i.e., the bi-objective case) is given in Figure 1.1.

A feasible solution of the problem is a connected, cycle-free subgraph $G' = (V', E')$ of G (i.e., $V' \subseteq V$ and $E' \subseteq E$) that contains the root vertex $r \in V$, as well as all terminal vertices

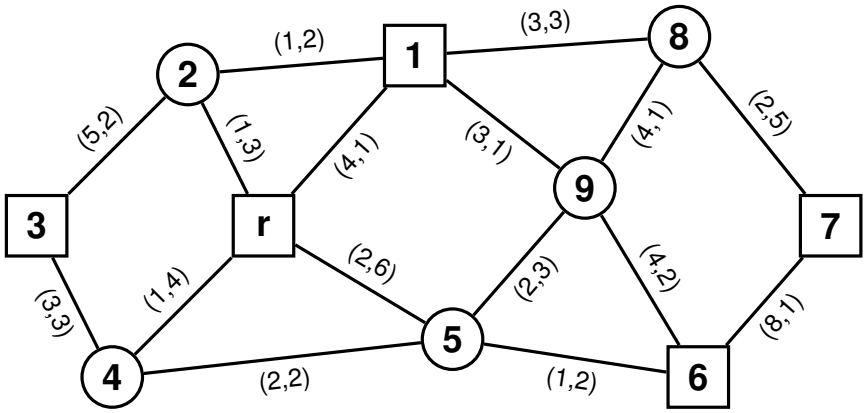


Figure 1.1: Example instance of the MOSTPR with $T = \{1, 3, 6, 7\}$. Edges are labeled $(cost, delay)$.

$T \subseteq V$ (i.e., G' is a feasible Steiner tree for G). Note that G' therefore contains a unique path p_t from r to each $t \in T$.

Our objective is to find a feasible solution that

a. minimizes the total cost of the selected edges, i.e.,

$$\min \sum_{e \in E'} c_e$$

b. minimizes the maximum demand of each resource, i.e.,

$$\min\{\max_{t \in T} \sum_{e \in p_t} d_e^j\} \quad \forall 1 \leq j \leq k$$

Exemplary optimal solutions for the instance given in Figure 1.1 are given in Figures 1.2 and 1.3.

Intuitively, the cost of an edge is how much we must spend to *create* a link between two vertices, whereas the resources indicate how much we must spend to *use* it. Therefore, we want to find a Steiner tree that is both cheap (w.r.t. edge cost) and has "short" routes (w.r.t. resource demands) to each terminal.

Since cost and resource demands of the edges are uncorrelated in general, we cannot expect the existence of a single optimal solution that minimizes all objectives simultaneously. We therefore aim to identify one Pareto-optimal solution for each point on the so called Pareto frontier. The latter is defined by the objective vectors of all efficient solutions, i.e. solutions that cannot be improved with respect to one of the objectives without deteriorating with respect to at least one other objective. See Section 2.3 for a brief introduction into these concepts.

In the bi-objective case, which we study in most detail, the resource demand vector $\mathbf{d}$ is one-dimensional and therefore represented by a simple integer d. Following the terminology of our motivating example [55], we will usually refer to d as *delay* instead of *resource demand*.

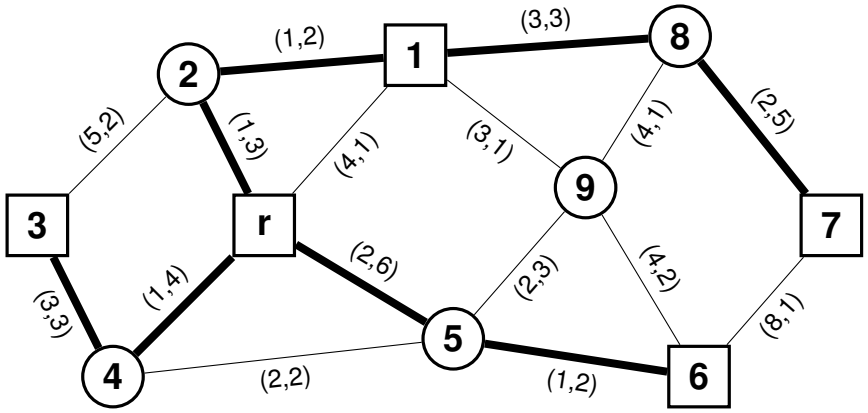


Figure 1.2: Solution for the instance in Figure 1.1 that is optimal w.r.t. total cost (which is 14). The maximum delay is 13 (for vertex 7). Edges that are part of the solution are highlighted.

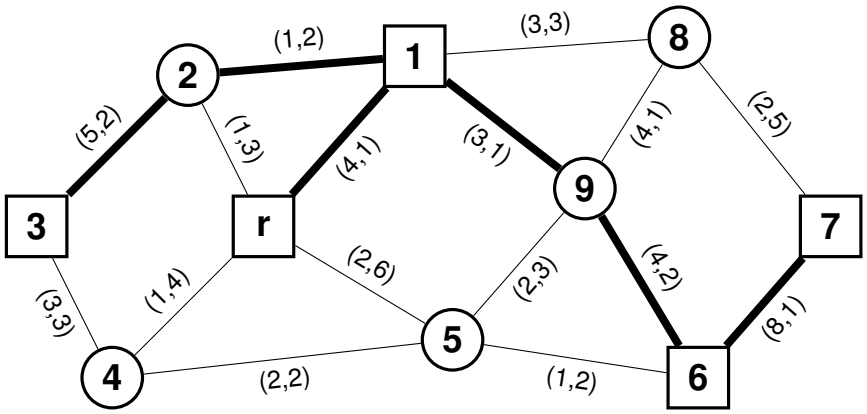


Figure 1.3: Solution for the instance in Figure 1.1 that is optimal w.r.t. maximum delay (which is 5 for vertices 3 and 7). The total cost is 25. Edges that are part of the solution are highlighted.

Similarly, we will refer to the bi-objective problem variant as *Bi-Objective Steiner Tree Problem with Delays* (BOSTPD).

1.2.1 Computational complexity of the MOSTPR

The decision variant of the STP is well-known to be NP-complete [32]. Since the MOSTPR is a generalization of the optimization variant of the STP (which is itself NP-hard), it must obviously be NP-hard itself. The following reduction and proof sketch show that this is the case.

Consider an arbitrary instance $G = (V, E)$, c with $V = S \cup T$ of the optimization variant of the STP. From this, we construct an instance of the MOSTPR $G' = (V', E')$, c', d with $V' = S' \cup T' \cup \{r'\}$ by setting $S' = S$, $T' = T$, $c'_e = c_e, \forall e \in E'$, $\mathbf{d}_e = \mathbf{0}, \forall e \in E'$, and selecting an arbitrary $r' \in T'$ as root vertex (i.e, simply setting the resource demands of every edge to zero). Obviously, this can be done in polynomial time and space.

If G has an optimal solution Sol, it must necessarily also be an optimal solution of G', since the objective functions of both problems are identical and Sol is a valid solution by definition (Note that while normally multi-objective problems do not have *optimal* solutions, our instance of the MOSTPR does, since for each solution, all objective values except one are zero). By a similar argument, if G' has an optimal solution Sol', it is also an optimal solution for G.

1.3 Outline

The remainder of this thesis is structured as follows:

Chapter 2 introduces basic concepts that will be used afterwards. More precisely, it introduces the ε-constraint method, integer linear programming and its variants, layered graphs and an algorithm to solve the delay-constrained shortest path problem.

Chapter 3 presents previous works that relate to this thesis. While to our knowledge no publication has covered this newly defined problem yet, a large number of publications on the topics of multi-objective optimization and the STP and its variants exist.

Chapter 4 describes the implementation of an algorithm that solves the Bi-objective Steiner Tree Problem with Delays (BOSTPD). Based on the ε-constraint method, it encodes the subproblems arising from it as ILPs and solves them by using an ILP solver. Computational enhancements like preprocessing and information reuse throughout the iterative procedure also form an important aspect of the algorithm. An evaluation of the algorithm's performance on a set of test instances, together with an interpretation of these results, is given in Chapter 5.

Chapter 6 describes how the algorithm from Chapter 4 can, in principle, be generalized to the multi-objective case, as well as the problems arising from such a generalization.

Finally, Chapter 7 provides concluding remarks and discusses possibilities for future research.

CHAPTER 2

Preliminaries

This chapter introduces the concepts and definitions that will be needed in the following chapters. This includes both very general concepts like solving linear programs or optimization problems with multiple objectives and more specific ones like algorithms for finding (resource constrained) shortest paths and the maximum flow in a network. All of these are covered by a large body of literature, such as [14, 34, 46, 60].

Since we focus on solving the proposed problems to proven optimality by using exact methods, other methods such as metaheuristics are not discussed here. Also note that we do not intend to give a full introduction into each topic mentioned in the following sections, but instead aim to briefly describe the methods and concepts that will be used in the later chapters of this thesis.

2.1 Linear programming

Linear programming (LP) deals with minimizing or maximizing a linear objective function over a convex polytope that is described by a set of linear constraints. Such an instance is called a linear program (LP).

In the relevant literature, linear programs are defined in several different ways [46, 60]. Due to the fact that the problem we aim to solve is a minimization problem, we will use the following definition for our thesis.

Definition 2.1. A *linear program* in its general form is defined as

$$\begin{aligned} \min \quad & \mathbf{c} \cdot \mathbf{x} \\ s.t. \quad & \mathbf{A}\mathbf{x} \geq \mathbf{b} \\ & x \in \mathbb{R}^n \end{aligned}$$

Here, $\mathbf{x} = (x_1, \ldots, x_n)$ is the vector of decision variables and $\mathbf{c} = (c_1, \ldots, c_n)$ is the cost vector which assigns each x_i a cost c_i with which it is weighted in the objective function. $\mathbf{A}$

is the $m \times n$ coefficient matrix that, together with vector $\mathbf{b} = (b_1, \ldots, b_m)$, defines the linear program's constraints.

Without loss of generality, we assume that we minimize the objective function, since we can trivially transform any instance where the objective function is maximized into one where it is minimized by negating the objective function.

Similarly, we must not restrict ourselves to $\geq$ constraints. Any kind of linear equality or inequality, as well as any restriction on a variable's range, can be encoded as a $\geq$ constraint by applying the following transformation rules:

- $\mathbf{a_i'x} = b_i \Leftrightarrow \mathbf{a_i'x} \leq b_i \wedge \mathbf{a_i'x} \geq b_i$
- $\mathbf{a_i'x} \leq b_i \Leftrightarrow -\mathbf{a_i'x} \geq -b_i$
- $x_j \geq 0 \Leftrightarrow \mathbf{a_i'x} \geq 0$, where $\mathbf{a_i}$ is the j-th unit vector
- $x_j \leq 0 \Leftrightarrow -\mathbf{a_i'x} \geq 0$, where $\mathbf{a_i}$ is the j-th unit vector

2.1.1 Simplex method

The by far most commonly used method for solving linear programs is the *Simplex algorithm* proposed by Dantzig [7]. Despite the fact that in the worst case, it might require an exponential number of steps to find the solution, its good empirical performance on a wide range of instances often makes it the algorithm of choice. Alternatively, one may use algorithms based on the ellipsoid method or interior-point methods (which are also called barrier methods), which guarantee polynomial runtime, but do not perform as well in practice (see [31, 33]).

Implementations of the aforementioned algorithms are available in a number of commercial and free software packages.

2.2 Integer linear programming

Many of the optimization problems that are relevant in practice require us to make decisions on integral objects. Therefore, we would like to have a way of encoding these problems as variants of linear programs.

Integer linear programming is an extension of linear programming that allows us to do just that by adding a new type of constraint: *integrality constraints*. These enable us to ensure that certain variables only take integer values. Linear programs with these constraints are called integer linear programs (ILPs) if all their variables are constrained to be integral, and mixed integer linear programs (MIPs) if only some of them are.

Definition 2.2. An *integer linear program* in its general form is given as

$$\begin{aligned} \min \quad & \mathbf{c} \cdot \mathbf{x} \\ s.t. \quad & \mathbf{Ax} \geq \mathbf{b} \\ & x \in \mathbb{Z} \qquad\qquad\qquad\qquad \forall x \in X \end{aligned}$$

MIPs are defined similarly. The only difference between the two is that for ILPs, all variables are constrained to be in $\mathbb{Z}$, whereas for MIPs, only the variables in a subset $X_Z \subset X$ are. All other variables $X \setminus X_Z$ may take non-integral values.

Integer linear programs offer a natural way of encoding instances of many NP-hard problems like the Traveling Salesman Problem (TSP) and solving them is therefore NP-hard as well. Despite this, many ILPs can be solved relatively efficiently in practice. An introduction into the methods commonly used to facilitate this is provided in the next subsection.

2.2.1 LP-based branch-and-bound

Integer linear programs can often be solved rather efficiently with *LP-based branch-and-bound* algorithms [39]. They are based on solving a relaxed (i.e., less restrictive) version of an ILP called the *LP relaxation*.

Definition 2.3. The *LP relaxation* of an ILP is obtained by relaxing all integrality constraints, i.e. allowing all variables to take non-integral values within their respective ranges. The LP relaxation of an ILP in general form is given as

$$\begin{aligned} \min \quad & \mathbf{c} \cdot \mathbf{x} \\ s.t. \quad & \mathbf{A}\mathbf{x} \geq \mathbf{b} \\ & x \in \mathbb{R} \qquad\qquad\qquad\qquad \forall x \in X \end{aligned}$$

These LP relaxations can be solved like any other LP, using the methods described in Section 2.1.1.

Algorithm 2.1 describes a generic version of an LP-based branch-and-bound algorithm for an ILP with an objective function that should be minimized.

We start by initializing the list of active problems as the LP relaxation of the initial ILP that we want to solve. The initial incumbent solution is undefined (since we do not know any feasible solution for IP yet) and the initial dual bound is infinity.

While there are still active problems in our list, we select one and solve it with an LP solver. Since L only ever contains LPs (we start with an LP and only ever add other LPs to it) x^P is the relaxed problem's solution and z^P its corresponding objective value. We now start the *bounding* phase of the algorithm.

If we were unable to find a solution for P, we prune it by infeasibility. This means that we explored a part of the search tree where the added constraints prevent us from finding a solution. Thus, we want to explore it no further.

If z^P is larger than (or equal to) our incumbent bound z, we know that the current part of the search tree can never yield an optimal solution for P, since we already found a solution that is both better and feasible for it. Thus, we prune P by bound.

If x^P is feasible for IP, we found a new candidate solution. If it is better than the incumbent $\mathbf{x}^*$, we update it and its corresponding bound z. Either way, we prune P by optimality, since there is nothing more to be done in this part of the search tree.

Data: an ILP IP
Result: a solution $\mathbf{x}^*$ for IP, if one exists

list of active problems $L : \{LP\_relax(IP)\}$;
incumbent solution $\mathbf{x}^*$;
minimum upper bound $z = \infty$;
while $L \neq \emptyset$ **do**
    choose a $P \in L$ and remove it from L;
    $x^P = solve(P)$;
    $z^P =$ objective value of x^P;
    if P *is infeasible* **then**
        prune P by infeasibility;
    else if $z^P \geq z$ **then**
        prune P by bound;
    else if x^P *is feasible for* IP **then**
        if $z^P \leq z$ **then**
            $z = z^P$;
            $\mathbf{x}^* = x^P$;
        end
        prune P by optimality;
    else
        select integer variable x_i for which $x_i^P \notin \mathbb{Z}$;
        $P_1 = P$ with add. constraint $x_i \leq \lfloor x_i^P \rfloor$;
        $P_2 = P$ with add. constraint $x_i \geq \lceil x_i^P \rceil$;
        $L = L \cup \{P_1, P_2\}$;
    end
end
return $\mathbf{x}^*$

Algorithm 2.1: LP-based branch-and-bound

Finally, if we were unable to prune the current problem P, we *branch*. We do this by selecting a variable x_i that should be integral in a solution for IP, but isn't in x^P. We generate two new problems P_1 and P_2, which split P's search space into two distinct parts.

P_1 covers the parts where x_i is less than the value found by x_i^P (rounded down), whereas P_2 covers the part where it is larger than x_i^P (rounded up). Since the part of the solution space that is removed by adding these constraints is infeasible for IP (x_i would be non-integral there), we are guaranteed to not lose feasible solutions this way.

2.2.2 Branch-and-cut

Many combinatorial optimization problems (COP) that are of practical relevance lend themselves to being modeled as integer linear programs with an exponential number of constraints. Additionally, many ILP models of polynomial size can be strengthened by adding constraint families of exponential size to them.

Since these models are too large to be efficiently handled by previously introduced methods such as branch-and-bound, as even generating them takes exponential time, researchers have devised the *branch-and-cut* method [47, 48] to solve these ILPs.

Branch-and-cut is an extension of the regular branch-and-bound procedure that allows the user to add additional inequalities during the iterative solution procedure. By intelligently choosing which of the exponentially many constraints to add to the model, an optimal solution can often be found without having to consider most of these inequalities.

A generic branch-and-cut algorithm is given in Algorithm 2.2.

To understand the branch-and-cut procedure, the notion of a *reduced problem* is important. Consider the following integer linear program IP:

$$\begin{array}{lll} \min & \mathbf{c} \cdot \mathbf{x} & \\ s.t. & \mathbf{A}\mathbf{x} \geq \mathbf{b} & \\ & \mathbf{D}\mathbf{x} \geq \mathbf{e} & \\ & x \in \mathbb{Z} & \forall x \in X \end{array}$$

IP has two sets of constraints: the polynomially sized $\mathbf{A}\mathbf{x} \geq \mathbf{b}$ and the exponentially sized $\mathbf{D}\mathbf{x} \geq \mathbf{e}$. We now define the corresponding reduced problem IP_R as

$$\begin{array}{lll} \min & \mathbf{c} \cdot \mathbf{x} & \\ s.t. & \mathbf{A}\mathbf{x} \geq \mathbf{b} & \\ & x \in \mathbb{Z} & \forall x \in X \end{array}$$

that is, IP without the exponentially many constraints defined by $\mathbf{D}\mathbf{x} \geq \mathbf{e}$. Since IP_R is of polynomial size, we can solve it by using the methods described in the preceding sections.

The algorithm starts out like a regular branch-and-bound with IP_R as the initial problem. We use IP_{LP} to refer to the LP relaxation of the original problem IP with all its exponentially many constraints.

The key aspect of branch-and-cut are lines 14 – 17. Here, after solving the current LP relaxation and ensuring that it is neither infeasible nor suboptimal, we check whether it violates any of the constraints that were removed during the construction of IP_R. If we find such violated constraints, which are called *cutting planes* or simply *cuts*, we add them to the current problem P and resolve it.

Once no more cuts can be found, x^P is feasible for IP_{LP}. We then proceed by checking for integrality, updating our incumbent solution and bound as necessary, and branching in case x^P is not integral.

The problem of finding constraints within a family of constraints that are violated by a candidate solution x^P or proving that none exist is called the *separation problem*. In many cases, the separation problem can be solved as a COP itself, e.g. by solving a number of problems defined

Data: a reduced ILP IP_R
Result: a solution $\mathbf{x}^*$ for the original problem IP, if one exists

```
1  list of active problems L : {LP_relax(IP_R)};
2  incumbent solution x*;
3  minimum upper bound z = ∞;
4  while L ≠ ∅ do
5      choose a P ∈ L and remove it from L;
6      repeat
7          x^P = solve(P);
8          z_i = objective value of x^P;
9          if P is infeasible then
10             prune P by infeasibility;
11         else if z_i ≥ z then
12             prune P by bound;
13         else
14             try to find cuts in IP_LP that are violated by x^P;
15             if new cuts were found then
16                 add new cuts to P
17             end
18         end
19     until no more new cuts found or P pruned;
20     if x^P is feasible for IP then
21         if z_i ≤ z then
22             z = z_i;
23             x* = x^P;
24         end
25         prune P by optimality;
26     else
27         select integer variable x_i for which x_i^P ∉ ℤ;
28         P_1 = P with add. constraint x_i ≤ ⌊x_i^P⌋;
29         P_2 = P with add. constraint x_i ≥ ⌈x_i^P⌉;
30         L = L ∪ {P_1, P_2};
31     end
32 end
33 return x*
```

Algorithm 2.2: branch-and-cut

according to x^P like shortest path or minimum cut. While they can be NP-hard in general, the separation problems associated with commonly used constraint families of exponential size like *cutset constraints*, which will be used in the algorithms described in Chapter 4, are solvable in polynomial time.

It is important to distinguish two kinds of constraints that are separated during branch-and-cut:

- **model constraints:** These inequalities are required to accurately encode the desired COP within IP. Any solution that violates one of these constraints is not a valid solution for the original COP that we are trying to solve. Therefore, these constraints must be separated exactly, i.e., none may be violated by an incumbent solution. For performance reasons, these constraints are sometimes only separated for integral solutions.

- **strengthening constraints:** These inequalities are only used for strengthening the LP relaxations of the problem. Any solution that is valid for the reduced ILP with all model constraints added to it is guaranteed to satisfy any of these constraints. Therefore, these constraints may be separated heuristically, which means that even constraint families with NP-hard associated separation problems can be considered in practical applications. Since these constraints can only be violated by non-integral solutions, they must be separated for all candidate solutions to fulfill their purpose.

2.3 Multi-objective optimization

As argued in Chapter 1, not all real-life optimization problems of practical interest can be modeled as single-objective optimization problems, since they aim to optimize several independent, typically conflicting aspects of a solution. Consider, for instance, the problem of buying a new car. The buyer might be interested in several aspects of the car, like purchase price, fuel economy and safety rating. To adequately model this problem, the notion of a *multi-objective optimization problem* is needed (see, e.g., [14, 51] for a more exhaustive introduction into the topic of multi-objective optimization).

Definition 2.4. A multi-objective optimization problem (MOOP) consists of a vector $\mathbf{x} = (x_1, \ldots, x_n)$ of decision variables, a set of feasible solutions $\mathbf{X} \subseteq \mathbb{R}^n$ and the objective functions $z_1, \ldots, z_m : \mathbb{R}^n \to \mathbb{R}$. Formally, an instance of a MOOP is given as

$$\begin{aligned} \min \quad & z_1(\mathbf{x}), \ldots, z_m(\mathbf{x}) \\ s.t. \quad & \mathbf{x} \in \mathbf{X} \end{aligned}$$

The set $\mathbf{X}$ is usually defined implicitly by a set of constraints. Note that we may, without loss of generality, assume the minimization of all objective functions, since we can easily transform any maximizing objective function into a minimizing one by negating it.

Multi-objective linear programs are a subclass of multi-objective optimization problems where both the objective functions $z_1, \ldots, z_m$ and the constraints describing the set of feasible solutions $\mathbf{X}$ are linear functions. They can be considered a multi-objective generalization of regular linear programs described in Section 2.1. Note that the MOSTPR as defined in Section 1.2 is *not* a multi-objective linear program, since only the cost objective function is a linear function, whereas all delay objective functions are bottleneck functions. However, as we will show in Chapters 4 and 6, it is possible to reformulate both the BOSTPD and the MOSTPR as multi-objective integer linear programs.

It is convenient to distinguish the decision (or criterion) space $\mathbb{R}^n$ from the objective space $\mathbb{R}^m$. Every point in the decision space, like candidate solutions $\mathbf{x} \in \mathbf{X}$, can be mapped to a point in the objective space by the vector-valued function $\mathbf{z}(x) = (z_1(x), \ldots, z_m(x))$. The components of $\mathbf{z}$ are the individual objective functions $z_1, \ldots, z_m$. Note that different solutions may be mapped to the same point in the objective space.

Since the individual objective functions are, in general, conflicting, we cannot expect the existence of a single solution $\mathbf{x}^*$ that is optimal w.r.t. every objective function. Instead, we are interested in finding a set of *efficient* solutions, specifically one solution for every point on the *Pareto frontier*.

Definition 2.5. A solution $\mathbf{x}$ of a MOOP with minimizing objective function $\mathbf{z}$ is called *dominated* by $\mathbf{x}'$ if $\mathbf{z}(\mathbf{x}') \leq \mathbf{z}(\mathbf{x})$ and $\exists i \in \{1, \ldots, m\} : z_i(\mathbf{x}') < z_i(\mathbf{x})$.

This notion of dominance leads to the definition of *efficient* solutions.

Definition 2.6. A solution $\mathbf{x}$ that is not dominated by any other solution in $\mathbf{X}$ is called *efficient* or *non-dominated*. The set of all points in the objective space that are the image of an efficient solution is the *Pareto frontier*.

The set of efficient solutions can be divided further into different categories, depending on where in the objective space their images lie. To this end, let $\mathbf{Z}$ be the feasible region of the objective space, which contains the image of every feasible solution in $\mathbf{X}$. Also, let $conv(\mathbf{Z})$ be the convex hull of $\mathbf{Z}$.

- The images of *supported* efficient solutions lie on the boundary of $conv(\mathbf{Z})$. These may be further divided into
  - efficient solutions whose images lie on a *vertex* of $conv(\mathbf{Z})$, which are called *extreme supported solutions*, and
  - efficient solutions whose images lie on the boundary of $conv(\mathbf{Z})$, but *not* on a vertex of it.
- The images of *non-supported* efficient solutions do *not* lie on the boundary of $conv(\mathbf{Z})$.

The *weighted sum method* (see, e.g., [12]) is a widely used procedure for solving multi-objective optimization problems. It transforms the original multi-objective problem into a single-objective problem whose objective function is the weighted sum of the original problem's objective functions. Formally, a MOOP as defined in Definition 2.4 is transformed by the weighted sum method into the following single-objective optimization problem:

$$\begin{aligned} \min \quad & \sum_{i=1}^{m} \lambda_i z_i(\mathbf{x}) \\ s.t. \quad & \mathbf{x} \in \mathbf{X} \\ & \lambda_i > 0 \qquad\qquad\qquad\qquad 1 \leq i \leq m \end{aligned}$$

Depending on the choice of the λ, different supported efficient solutions are found.

This method is especially convenient for multi-objective linear programs, since the resulting single-objective problem remains a linear program, which can be solved using the techniques presented in Sections 2.1 and 2.2.

However, the weighted sum method can only find supported efficient solutions. It can therefore not be used for computing the complete Pareto frontier of a multi-objective integer linear program, only its convex hull. Furthermore, while finding supported efficient solutions on vertices of $conv(\mathbf{Z})$ is simple, since each of these solutions is the unique optimal solution of the transformed problem with a certain selection of λ, the same cannot be said about the remaining supported efficient solutions. Since they are not the unique optimal solution of the transformed problem for any choice of λ, but merely one its several optimal solutions, a complete enumeration of all these optimal solutions is required for finding them.

2.3.1 ε-constraint method

In contrast to the weighted sum method, the *ε-constraint method* (see [12] again) can be used to find all efficient solutions of a multi-objective optimization problem, whether they are supported or non-supported. It converts the original problem into a single-objective problem by transforming all but one objective functions into constraints. Formally, a MOOP as defined in Definition 2.4 is transformed into the following single-objective optimization problem:

$$\begin{aligned} \min \quad & z_i(\mathbf{x}) \\ s.t. \quad & \mathbf{x} \in \mathbf{X} \\ & z_j \leq \varepsilon_j \qquad\qquad \forall j \in \{1, \ldots, m\} \setminus \{i\} \end{aligned}$$

Depending on the choice of the ε parameters, different efficient solutions can be found.

An algorithm implementing the ε-constraint method for a bi-objective optimization problem is given in Algorithm 2.3.

Let P' be a single-objective variant of P with the second objective function z_2 removed. The variable δ determines how much of the objective space of z_2 must not be considered during the next iteration, since we cannot expect to find an efficient solution there. If z_2 only takes integer values, δ is usually one.

The optimal solution $\mathbf{x}'$ of P' without any added ε-constraints determines the starting point of our search. We add $\mathbf{x}'$ to S and begin the iterative procedure.

During every iteration, we solve the ε-constraint variant of P' where z_2 is constrained to be at least δ less than the z_2 value of the last optimal solution we found. After removing all those

Data: a bi-objective optimization problem $P = \min\{(z_1, z_2) : \mathbf{x} \in \mathbf{X}\}$, δ
Result: all efficient solutions of P
set of efficient solutions $S = \emptyset$;
$P' = \min\{z_1 : \mathbf{x} \in \mathbf{X}\}$; // P with z_2 removed
solve $\mathbf{x}' = P'$;
$S = S \cup \{\mathbf{x}'\}$;
while $\mathbf{x}'$ *is feasible* **do**
    solve $\mathbf{x}' = \min\{z_1 : \mathbf{x} \in \mathbf{X}, z_2(\mathbf{x}) \leq z_2(\mathbf{x}') - \delta\}$; // P' with added ε-constraint for z_2
    $S = S \setminus \{\mathbf{x} : \mathbf{x}'$ dominates $\mathbf{x}\}$;
    $S = S \cup \{\mathbf{x}'\}$;
end
return S;

Algorithm 2.3: ε-constraint method for bi-objective optimization problems

solutions from S that are dominated by our newly-found $\mathbf{x}'$, which can happen if $\mathbf{x}'$ has the same z_1 value as the last optimal solution, we add $\mathbf{x}'$ to S and proceed to the next iteration.

Once z_2 is constrained to the point where no more feasible solutions for P' with the ε-constraint can be found, we end our search and return the set of efficient solutions S.

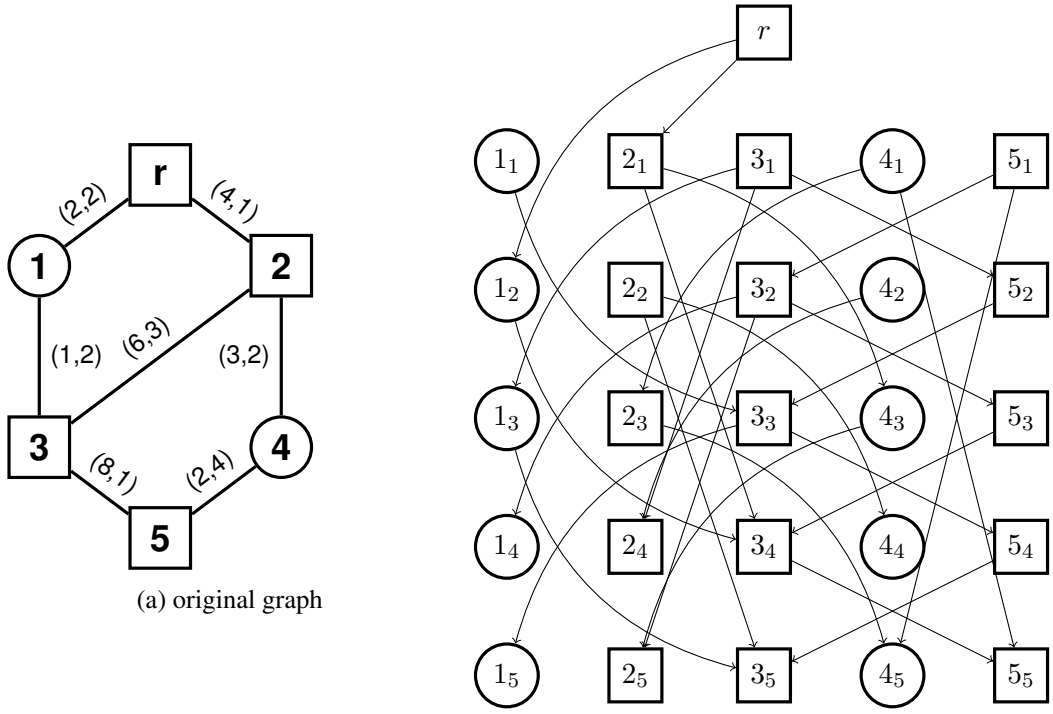
2.4 Layered graphs

Using the ε-constraint method for solving the MOSTPR introduces a new kind of constraint to our formulations, namely constraints on the maximum length (w.r.t. one of the resources) of a path between the root vertex r and each terminal vertex $t \in T$. Thus, we require a way of encoding this in a linear model. One way of doing so is implicitly encoding this maximum length in the data structure on which we describe the model. This leads us to the following transformation of the original graph into a *layered graph*, which was introduced in [55] and [26]. Figure 2.1 shows this transformation on an example graph of an instance of the bi-objective problem.

Let $G = (V, A)$ be a directed variant of the original graph $G' = (V, E)$ of an instance of the BOSTPD, where every root edge $\{r, v\} \in E$ is replaced by an arc $(r, v) \in A$ and every other edge $\{u, v\} \in E$ is replaced by two arcs $(u, v), (v, u) \in A$. Cost and delay of these arcs are equal to the cost and delay of their corresponding edge in the undirected graph, respectively. Also, let $B \in \mathbb{N}$ be the maximum allowed delay, which corresponds to the number of layers below the root vertex in the layered graph. We now transform G into a layered directed graph $G_L = (V_L, A_L)$.

First, we create B copies of every non-root vertex, one for each layer. These newly-created vertices inherit their type, i.e., whether they are terminal vertices or not, from the vertex they are copied from. Thus, $V_L = \{r_L\} \cup T_L \cup S_L$, where

- $T_L = \{v_l | v \in T, 1 \leq l \leq B\}$ is the set of terminal vertices and



(a) original graph

(b) layered graph

Figure 2.1: Example instance of the BOSTPD, with $T = \{2, 3, 5\}$ and edges labeled $(cost, delay)$, and its corresponding layered graph for $B = 5$, with its vertices labeled $vertex_{layer}$.

- $S_L = \{v_l | v \in S, 1 \leq l \leq B\}$ is the set of non-terminal vertices.

Next, we connect these vertices according to their connections in the original graph, i.e., if an arc $a = (u, v)$ exists in the original graph, arcs are added between their copies in the layered graph. Which specific copies are connected by an arc is determined by the original arc's associated delay d_a: an arc connects the source vertex on layer l to the target vertex on layer $l + d_a$. Formally, $A_L = A_L^r \cup A_L^g$, where

- $A_L^r = \{(r_L, v_{d_{rv}}) | (r, v) \in A, d_{rv} \leq B\}$ is the set of root arcs and
- $A_L^g = \{(u_l, v_{l+d_{uv}} | (u, v) \in A, u, v \neq r, 1 \leq l \leq B - d_{uv}\}$ is the set of general arcs.

These newly-created arcs have an associated cost equal to the cost of their corresponding original arc. Since the arcs' delays are now implicitly encoded in the layered graph's structure, they must not be explicitly considered when dealing with the layered graph.

Since we introduce a copy of every vertex for each layer in the layered graph, it contains $|V_L| = O(B \cdot |V|)$ vertices. By a similar argument, it contains $|A_L| = O(B \cdot |A|)$ arcs in the worst case (specifically, when G is a complete graph and all arcs have delay $d_e = 1$). This size

can, however, be reduced significantly by removing all arcs and vertices that are not required by the application at hand. For our example (encoding an instance of the Rooted Delay-constrained Steiner Tree Problem (RDCSTP) as an ILP, see Section 4.3.2), this includes removing all vertices without incoming arcs, as well as all copies of potential Steiner vertices that are leafs (i.e., that have no outgoing arcs).

While the transformation was only described for the bi-objective case, the transformation of graphs for the multi-objective case with m resource constraints works analogously. Given bounds $B_1, \ldots, B_m$, we can again define $G_L = (V_L, A_L)$ as

- $V_L = \{r_L\} \cup T_L \cup S_L$, where
  - $T_L = \{v_{l_1,\ldots,l_m} | v \in T, 1 \leq l_1 \leq B_1, \ldots, 1 \leq l_m \leq B_m\}$
  - $S_L = \{v_{l_1,\ldots,l_m} | v \in S, 1 \leq l_1 \leq B_1, \ldots, 1 \leq l_m \leq B_m\}$
- $A_L = A_L^r \cup A_L^g$, where
  - $A_L^r = \{(r_L, v_{d_{rv}^1,\ldots,d_{rv}^m}) | (r,v) \in A, 1 \leq d_{rv}^1 \leq B_1, \ldots, 1 \leq d_{rv}^m \leq B_m\}$
  - $A_L^g = \{(u_{l_1,\ldots,l_m}, v_{l_1+d_{uv}^1,\ldots,l_m+d_{uv}^m}) | (u,v) \in A, u, v \neq r, 1 \leq l_1 \leq B_1 - d_{uv}^1, \ldots, 1 \leq l_m \leq B_m - d_{uv}^m\}$

We use d_{uv}^i to refer to the associated resource demand w.r.t. objective function i of arc $(u,v) \in A$.

One significant disadvantage of multi-dimensional layered graphs is their potentially large size: in general, we have $V_L = O(|V| \cdot B_1 \cdot \ldots \cdot B_m)$ and $A_L = O(|A| \cdot B_1 \cdot \ldots \cdot B_m)$. As for the two-dimensional case, however, preprocessing can help to reduce the size.

2.5 Construction heuristics

A *construction heuristic* is a heuristic algorithm that finds an initial solution for an optimization problem by starting with an empty solution and extending it until some stopping criterion is reached. This solution is often used as a starting point in iterative solution procedures such as LP-based branch-and-bound or local search, where it is gradually improved during the course of the algorithm.

Unlike exact methods, which are guaranteed to find the optimal solution as long as they are given enough computational resources, and approximation algorithms, which are guaranteed to find a solution that is within some bound of an optimal solution, heuristics provide no guarantee for the quality of their solution. However, those that are widely used usually find a solution much faster than the aforementioned methods.

One simple and generally applicable construction heuristic is to select a random solution. As long as such a solution can be found efficiently, i.e., in polynomial time, this procedure can be used for any optimization problem. However, this procedure is usually outperformed by problem specific heuristics w.r.t. solution quality.

So-called *greedy* construction heuristics are an important class of construction heuristics. These algorithms start with an empty solution and, during each step, extend the current partial

solution with an element whose addition is optimal w.r.t. some greedy criterion and that partial solution. Once an element is part of the solution, it is never removed during the course of the algorithm. Algorithm 2.4 illustrates a generic version of a greedy construction heuristic.

Data: an optimization problem P
Result: a solution x for P
$x =$ empty solution;
repeat
  $e =$ locally optimal extension for x;
  $x = x + e$;
until *stopping criterion reached*;
return x

Algorithm 2.4: generic greedy construction heuristic

The stopping criterion usually depends on whether we are dealing with a minimization or a maximization problem. In most cases, adding new elements to a solution increases its objective value in both kinds of optimization problems. Minimization problems usually constrain valid solutions to contain at least some of the possible elements, since otherwise, an empty solution would likely be optimal. Therefore, the empty solution is often invalid for them and construction heuristics for minimization problems commonly add elements until the solution is feasible. On the other hand, the constraints of a maximization problem often require that we not select every possible element, since otherwise, a solution containing them all would trivially be optimal, while an empty solution would be valid, but suboptimal in general. Thus, construction heuristics for maximization problems usually add elements to the solution until further additions would make that solution infeasible.

2.6 Shortest path problem

The *shortest path problem in graphs* is the combinatorial optimization problem of finding a shortest path between vertices of an edge-weighted graph, where a path's length is determined by the total weight of its edges. It is encountered in a variety of scenarios, like routing network packets or finding the fastest way to travel from point A to point B. The most common variant of the shortest path problem is the Single-source Shortest Path problem (SSSPP), which is concerned with finding a shortest path between a designated source vertex and every other vertex.

Formally, an instance of the single-source shortest path problem consists of a directed graph $G = (V, A)$, a designated source vertex s and a cost function $c : A \to \mathbb{R}$ that assigns each arc a "length". Undirected graphs $G = (V, E)$ may also be considered as input graphs, since they can be converted into directed graphs by replacing each edge $\{u, v\} \in E$ with two arcs $(u, v), (v, u) \in A$ and assigning both arcs the original edge's cost. A feasible solution of the SSSPP is a subset $x \subseteq A$ that contains a path P_v from s to every $v \in V$. The objective is to find a solution that contains a minimal cost path P_v^* for every $v \in V$, where a path's cost is the sum of the edge cost of all edges contained in it [34].

One popular algorithm for solving the SSSPP that is now in widespread use was introduced by Dijkstra [8]. It performs very well in practice and has a worst-case runtime of $O(|V|^2)$ [60], which can be further improved to $O(|E| \log |V|)$ by using priority queues as data structures for storing adjacency lists [34]. However, it is limited to instances with non-negative arc costs.

Another algorithm, introduced by Bellman, Ford and Moore [4, 16, 45], can handle instances with negative arc costs, as long as no cycles with negative total cost exist, but has a worse runtime performance of $O(|E||V|)$ [34].

Another variant of the shortest path problem, the All-pair Shortest Path problem (APSPP), is concerned with finding a shortest path between all vertex pairs $s, t \in V$. Algorithms for solving this problem have been introduced by Floyd, Warshall and Roy [15, 54, 62], as well as Johnson [29].

2.6.1 Resource-constrained shortest path problem

The *resource-constrained shortest path problem* generalizes the regular shortest path problem by introducing additional resource constraints on the paths, similar to the delay constraints we impose on the individual ε-constrained instances of the BOSTPD. Formally, an instance consists of a directed graph $G = (V, A)$, a designated source vertex s, a cost function $c : A \to \mathbb{N}$, a delay function $d : A \to \mathbb{N}$ and a delay bound B. A feasible solution is a subset $x \subseteq A$ that contains a path P_v from s to every $v \in V$ whose length w.r.t. the path's total delay is less than or equal to B, i.e., $\sum_{(i,j)\in P} d_{ij} \leq B$. Again, the objective is to find a solution that contains a feasible minimal cost path P_v^* for every $v \in V$ [10].

While the problem is weakly NP-hard, dynamic programming algorithms that can solve it in $O(|A| \cdot B)$ time exist. One such algorithm, described by Gouveia [25], is presented in Algorithm 2.5

The set S_b contains all vertices that can be reached with delay B. $f(i, d)$ denotes the length of the currently shortest path to i with delay d, whereas $MinCost(i)$ is the minimum over all $f(i, d), 0 \leq d \leq b$, i.e., the length of the currently shortest path to i that is within our current delay bound b. The variable $pred(i, d)$ simply saves the predecessor of i for delay d, so that we may reconstruct the path after the algorithm has finished.

We initialize S_0 as the set containing only the source vertex, whereas all other sets S_b are initialized as empty sets. Variable $f(0, 0)$ is set to zero, whereas all other $f(i, d)$, as well as all $MinCost(i)$, are initialized as infinite. All predecessors are initialized as null references.

The outer loop of the algorithm simply iterates through all possible delay bounds b. For each b, we find all all vertices that are reachable at this delay and at a cost that does not exceed the cost of reaching k, and update their $MinCost$ if necessary.

In the second inner loop, we again only consider those vertices that are reachable at this delay and at a cost that does not exceed the cost of reaching k. For each of them, we find all outgoing arcs whose delay, when added to the current bound b, remains within the global bound B. If such an arc can be part of an optimal solution, we add its target vertex j to $S_{b+d_{ij}}$ and set the new path to j at delay $b + d_{ij}$ appropriately. If necessary, we update $MinCost(k)$.

Data: a directed graph $G = (V, A)$ with arc costs c_{ij} and arc delays d_{ij}, target vertex $k \in V$

Result: the shortest path (w.r.t. cost) from source vertex 0 to k

```
1  S_0 = {0};
2  S_b = ∅ for 1 ≤ b ≤ B;
3  f(0,0) = 0;
4  f(i,d) = ∞ for i ∈ V \ {0} and 0 ≤ d ≤ B;
5  MinCost(i) = ∞ for i ∈ V;
6  pred(i,d) = null for i ∈ V and 0 ≤ d ≤ B;
7  for b ∈ {0,...,B−1} do
8      for i ∈ S_b where f(i,b) ≤ MinCost(k) do
9          MinCost(i) = Min{MinCost(i), f(i,b)};
10     end
11     for i ∈ S_b where f(i,b) ≤ MinCost(k) do
12         for (i,j) ∈ A where b + d_ij ≤ B do
13             if c_ij + f(i,b) < Min{MinCost(j), MinCost(k), f(j,b+d_ij)} then
14                 S_{b+d_ij} = S_{b+d_ij} ∪ {j};
15                 f(j,b+d_ij) = f(i,b) + c_ij;
16                 pred(j,b+d_ij) = i;
17                 if j = k then
18                     MinCost(k) = f(j,b);
19                 end
20             end
21         end
22     end
23 end
```

Algorithm 2.5: delay-constrained shortest path

2.7 Maximum flow and minimum cut problem

The *maximum flow problem* and the *minimum cut problem* are two optimization problems that are closely connected to each other. Formally, they are defined as follows:

An instance of both the maximum flow and the minimum cut problem consists of a directed graph $G = (V, A)$, designated source and target vertices $s, t \in V$ and a capacity function $c : A \to \mathbb{R}^+$ that assigns each arc (u, v) a non-negative capacity c_{uv} [34].

A feasible flow assigns each arc $(u, v) \in A$ a flow f_{uv} such that the following constrains are satisfied

- $f_{uv} \leq c_{uv}$, i.e., the flow on an arc never exceeds that arc's capacity
- $\sum_{(u,v)\in A} f_{uv} = \sum_{(v,w)\in A} f_{vw} \quad \forall v \in V \setminus \{s, t\}$, i.e., the flow coming into any non-source/non-target vertex must be equal to the flow going out of the vertex

The goal of the maximum flow problem is to find a flow that maximizes the flow entering the target, which is equal to the flow exiting the source.

A *cut* partitions the set of vertices V into two disjoint sets S and T, where $s \in S$ and $t \in T$. The capacity of a cut is the sum of the capacities of all arcs crossing the cut, i.e., $\sum_{(u,v)\in A, u\in S, v\in T} c_{uv}$. The set of arcs that crosses a cut is called a *cutset*.

The goal of the minimum cut problem is finding a cutset with minimal total capacity.

Ford and Fulkerson [17], as well as Elias, Feinstein and Shannon [13] have shown that the two aformentioned problems are equivalent, i.e., that by finding a maximum flow in a flow network, one also solves the problem of finding a minimum cut within it, and vice versa. The furthermore showed that the value of a maximum flow is equal to the value of a minimum cut.

One algorithm for solving the maximum flow and minimum cut problem has been introduced by Ford and Fulkerson [17]. It is based on the idea of *augmenting paths*, i.e., finding a non-saturated path between s and t and adding flow on all its edges until it is saturated. Thus, during every iteration of the algorithm, the current flow is feasible. The algorithm's runtime is $O(|E| \cdot f^*)$, where f^* is the value of the maximum flow [34]. An improved version of this algorithm by Edmonds and Karp improves this to $O(|V| \cdot |E|^2)$ [11].

The *preflow-push algorithm*, also called *push-relabel algorithm*, introduced by Goldberg and Tarjan [20] relaxes the requirement that the preliminary flow found during each iteration of the algorithm must be feasible. Instead, more flow may temporarily enter a vertex than leave it. Once the algorithm terminates, the resulting flow is guaranteed to be feasible again. Its runtime is $O(|V|^2|E|)$, which can be further improved to $O(|V|^3)$.

CHAPTER 3

State of the Art

To the best of our knowledge, the MOSTPR has been newly defined in this thesis. Thus, no scientific literature that deals with this problem has been published so far. It is, however, closely connected to several other well-studied optimization problems, both single- and multi-objective.

First and foremost, the MOSTPR is a multi-objective generalization of the STP. This optimization problem, which was described by Dreyfus and Wagner [9], by Hakimi [27] and by Levin [43], is concerned with finding an overall cheapest subtree that connects all so-called terminal vertices of a graph. The decision variant of the STP, which is concerned with determining whether a Steiner tree of total cost $\leq k$ exists, was shown to be NP-complete by Karp [32], which implies that the optimization variant must be NP-hard as well.

The STP is itself a generalization of the Minimum Spanning Tree Problem (MSTP), where a cheapest subtree connecting all of a graph's vertices is to be found. The MSTP is notable for being solvable in polynomial time, e.g., by algorithms developed by Kruskal [38] or by Prim [50].

As the STP is NP-hard, we do not know of an algorithm that can solve it to optimality in polynomial time. We therefore have to resort to using algorithms with an exponential worst-case runtime if we want to find an exact solution. Specifically, encoding an STP instance as an integer linear program (cf. Section 2.2) and then solving the resulting ILP is an approach that has been used successfully for a number of instances. Works by Koch and Martin [35], by Goemans and Myung [19] and by Polzin [49] give overviews on different ILP formulations for the problem. A directed cut-based ILP formulation, upon which our formulations in Chapter 4 are based, was presented by Wong [64]. Additionally, Hwang et al. give a general overview on the topic [28].

Since some optimization problems arising in practice, such as finding a cheap computer network layout that limits the delay between clients and a server, cannot be modeled as instances of the regular STP, extended variants of the problem are needed. One such extended problem, the RDCSTP, was introduced by Kompella et al. [36, 37]. Here, a constraint on the maximum delay between a designated root vertex and every terminal vertex is imposed (cf Section 4.1). Methods for solving this problem to proven optimality that are based on branch-and-cut [55, 56] and branch-and-price [42] have been proposed by Ruthmair et al. and Leitner et al., respectively.

As for the regular STP, variants of both the hop- and the delay-constrained extensions exist where all vertices must be connected in the subtree induced by the solution: the Hop-constrained Minimum Spanning Tree Problem (HCMSTP) and the Rooted Delay-constrained Minimum Spanning Tree Problem (RDCMSTP). The latter was introduced by Salama et al. [58], where the authors also proved that in contrast to the regular MSTP, the extended variant is NP-hard.

To solve the HCMSTP to proven optimality, Gouveia introduced ILP models based on Miller-Tucker-Zemlin inequalities [21] and on multi-commodity flow inequalities [22]. The approach from [21] was later extended to the RDCSTP by Leggieri et al. [40], which also describes a construction heuristic based on delay-constrained shortest paths that we used for the heuristic repair of solutions from previous iterations (cf. Section 4.4.1). Another ILP-based approach for the RDCMSTP that works on a layered graph transformation of the original instance (cf. Section 2.4) was proposed by Gouveia et al. [25].

In addition to these exact methods, a number of (meta-)heuristic algorithms for solving the aforementioned constrained variants of the MSTP and the STP have been proposed (see, e.g., [18, 53, 59, 65])

In his PhD thesis, Ruthmair [55] proposed a layered graph model for the RDCSTP that is based on [25], as well as a branch-and-cut approach for infeasible paths for the same problem. Our approaches for solving the single-objective subproblems arising during the ε-constraint algorithm are based on his work. The thesis also presents several other ILP formulations for the RDCSTP, in addition to solution approaches for the RDCMSTP.

Layered graphs are frequently used to model constraints that limit the cumulative consumption of some resource along paths in a graph. Examples include solution approaches for the HCMSTP and the RDCMSTP by Gouveia et al. [26], as well as for the Hop-constrained Steiner Tree Problem (HCSTP) by Gouveia et al. [23]. They have also be used for solving other combinatorial optimization problems like the Hop-constrained Facility Location Problem by Ljubic and Gollowitzer [44] or the Diameter-constrained Steiner Tree Problem [24] by Gouveia et al.

As a general method for solving bi- and multi-objective optimization problems, the ε-constraint method is used in a number of scientific publications. For instance, Bérubé et al. use it to solve a bi-objective variant of the TSP [5]. An overview on multi-objective optimization in general is given by Chankong and Haimes [6].

In a survey, Ruzika and Hamacher give an overview on how to solve a class of multi-objective MSTPs [57]. Notably, they consider both sum and bottleneck objectives for their optimization problem, as do we in the MOSTPR. However, their bottleneck objective functions only consider individual edges, whereas we consider complete root-terminal paths. Vujosevic and Stanojevic use a similar combination of sum and bottleneck objective functions to solve bi-objective STP [61].

Approaches for solving the Bi-objective Prize-collecting Steiner Tree Problem (BOPCSTP), where maximizing profits from a graph's selected vertices and minimizing the cost of the selected edges are the two competing objectives, is given by Leitner et al. [41]. Similar to our approach, they reuse previous solutions to speed up their solution algorithm.

CHAPTER 4

Solution approaches for the Bi-objective Steiner Tree Problem with Delays

In this chapter, we detail the algorithm developed for solving the BOSTPD as defined in Section 1.2. It is based on the ε-constraint approach described in Section 2.3.1, where each instance of the bi-objective problem is transformed into a series of instances for the single-objective delay-constrained Steiner tree problem. Each of these is then solved by performing the following steps: (1) preprocessing, (2) encoding as an integer linear program (ILP) and (3) solving this ILP. As will be shown in Chapter 5, the algorithm's performance can be significantly improved by reusing information from previous iterations. Specifically, we reuse partial and complete solutions, as well as constraints added by branch-and-cut.

4.1 ε-constraint method for the BOSTPD

Recall that the BOSTPD has the following two objective functions, both of which are to be minimized:

1. the total cost of all selected edges $e \in E'$: $\sum_{e \in E'} c_e$
2. the maximum of all paths' total delays $\max_{t \in T} \sum_{e \in p_t} d_e$

In accordance with the ε-constraint method, one of the objective functions remains as the objective function of the single-objective instances, whereas the other is transformed into a constraint. Since we want to encode the single-objective instances as ILPs, we are interested in both a linear objective function and linear constraints.

First, note that the first objective function (the total edge cost) is already a linear function and can therefore directly be used as both an objective function and as a constraint in our ILP model.

In contrast, the second objective function (the maximum path delay) is a bottleneck function, which needs to be linearized before it can be used in an ILP model in any way. Since linear models for the delay-constrained Steiner tree problem already exist (see Chapter 3), we chose to keep the first objective function as such and transform the second one into a set of constraints. Thus, the resulting single-objective problem, the rooted delay-constrained Steiner tree problem (RDCST), can be defined as follows (cf. Section 1.2, [55]):

Definition 4.1. An instance of the RDCST is a graph $G = (V, E, c, d)$, with the set of vertices $V = S \cup T \cup \{r\}$, the set of edges E, the cost function $c : E \to \mathbb{N}$ and the delay function $d : E \to \mathbb{N}$. Additionally, an instance contains a delay bound B. A feasible solution of the problem is a connected, cycle-free subgraph $G' = (V', E')$ of G containing r and every terminal vertex $t \in T$, where each path p_t between the root vertex r and a terminal t has a cumulative delay of at most B, i.e.,

$$\sum_{e \in p_t} d_e \leq B \quad \forall t \in T$$

Note that while the original delay objective function was concerned with minimizing a solution's maximum path delay, we instead ensure that *no* path exceeds the delay bound B, which automatically limits the longest path's delay.

Our objective is to find a feasible solution that minimizes the total cost of all edges in E', i.e.,

$$\min \sum_{e \in E'} c_e \tag{4.1}$$

$$\text{s.t. } G' \text{ is a valid Steiner tree for } G \tag{4.2}$$

$$\sum_{e \in p_t} d_e \leq B \quad \forall t \in T \tag{4.3}$$

We refer to the total edge cost $c_{G'}$ of a solution as its *cost* and to the maximum path delay $d_{G'}$ as its *delay*.

4.1.1 High-to-low delay bound

One major aspect of efficiently implementing the ε-constraint method is choosing the right ε parameters (in our case, the delay bound B) for the different instances of the resulting single-objective problem (the RDCST in our case), since this determines which of the efficient solutions can be found. To ensure the identification of a solution for every point on the Pareto frontier, which is our objective, we must consider all possible delay bounds B for which an efficient solution may exist.

One way of doing so is to start at the highest reasonable delay bound $B = B_{max}$ and decreasing B with every step until it reaches the lowest reasonable bound $B = B_{min}$. Algorithm 4.1 describes this procedure in more detail.

Let $d(Sol)$ be the maximum path delay of a feasible solution Sol.

Data: an instance of BOSTPD with $G = (V, E, c, d)$
Result: all efficient solutions S

$S = \emptyset$;
$Sol_{STP} =$ minimum Steiner tree of G, disregarding delays;
$S = S \cup \{Sol_{STP}\}$;
$B = d(Sol_{STP}) - 1$;
$B_{min} = \max($shortest path from r to every t w.r.t. delay$)$;
while $B \geq B_{min}$ **do**
    $Sol =$ minimum Steiner tree of G with maximum path delay $\leq B$;
    $S = S \setminus \{Sol' : Sol$ dominates $Sol'\}$;
    $S = S \cup \{Sol\}$;
    $B = d(Sol) - 1$;
end
return S;

Algorithm 4.1: ε-constraint method for the BOSTPD, going from high to low delay bound

Our algorithm first solves the regular (i.e, not delay-constrained) Steiner tree problem on graph G. The solution Sol_{STP} is our first candidate for being a Pareto-efficient solution and is therefore added to S. Furthermore, $d(Sol_{STP})$ provides us with an upper bound B_{max} for the maximum delay of any efficient solution. The latter holds, since Sol_{STP} dominates every solution Sol_{over} with $d(Sol_{over}) > d(Sol_{STP})$. By definition, Sol_{STP} has lower maximum path delay than any Sol_{over}, whereas no Sol_{over} can have lower total edge cost than Sol_{STP}, since the latter is an optimal solution for the STP on G.

To find a lower bound B_{min} for the delay of any efficient (in fact, any feasible) solution, we calculate the shortest path (see Section 2.6) w.r.t. delay from r to every terminal vertex $t \in T$ and set B_{min} to the maximum of these paths' lengths. Since this is the lowest delay bound at which we can reach every terminal vertex, we need not consider any lower delay bounds, as no feasible solution for them exists.

After finding the aforementioned upper and lower bounds for B, we start our iterative procedure by setting $B = B_{max} - 1$. We then obtain a candidate solution Sol by solving the RDCST problem on G with delay bound B. To see if Sol dominates any existing solution in S, we compare its cost to that of the last solution Sol' we found. If both costs are equal, Sol dominates Sol' (since it has the same cost, but lower maximum path delay) and Sol' is removed from S. Note that it is sufficient to compare Sol to the solution from the previous iteration, since any other solution with equal cost would have already been removed when we added Sol' to S.

For the next iteration, we set the delay bound B to $d(Sol) - 1$. We may skip all intermediate delay bound values, since no efficient solution Sol'' with $d(Sol) \leq d(Sol'') \leq d(Sol')$ can exist. Any such Sol'' is covered by one of the following cases:

1. $d(Sol) = d(Sol'')$ and $c(Sol) = c(Sol'')$: In this case, both Sol and Sol'' are solutions for the same point on the Pareto frontier, whereas we are content with finding only one solution for every such point.

2. $d(Sol) \leq d(Sol'')$ and $c(Sol) \leq c(Sol'')$, with at least one of the inequalities being strict: Sol dominates Sol'', since either $d(Sol) < d(Sol'')$ or $c(Sol) < c(Sol'')$.

3. $d(Sol) \leq d(Sol'')$ and $c(Sol) > c(Sol'')$: If such a solution Sol'' were to exist, our algorithm for solving the RDCST would have found *it* as the current iteration's solution instead of Sol, since it is clearly a feasible solution for delay bound $B = d(Sol')$, but has lower cost than Sol.

Once $B = B_{min}$, we return S as the algorithm's solution, since it must now contain a solution for every point on the Pareto frontier.

4.1.2 Low-to-high delay bound

While the algorithm described in Section 4.1.1 starts with an unconstrained instance of the RDCST and gradually constrains its maximum delay further and further until the instance becomes infeasible, we can also go in the opposite direction by starting with the most strongly constrained instance possible and gradually relaxing its delay constraint until we reach B_{max}. Algorithm 4.2 describes such an algorithm:

Data: an instance of BOSTPD with $G = (V, E, c, d)$
Result: all efficient solutions S
$S = \emptyset$;
Sol_{STP} = minimum Steiner tree of G, disregarding delays;
$S = S \cup \{Sol_{STP}\}$;
$B = d(Sol_{STP})$;
repeat
  Sol = minimum Steiner tree of G with $d(Sol) \leq B$;
  if Sol *is cheaper than the last added efficient solution* **then**
    $S = S \cup \{Sol\}$;
  end
  $B = B + 1$;
until $c_{Sol} = c_{Sol_{STP}}$;
if $d(Sol_{STP}) = d(Sol)$ **then**
  $S = S \setminus \{Sol_{STP}\}$;
end
return S;

Algorithm 4.2: ε-constraint method for the BOSTPD, going from low to high delay bound

The algorithm starts similarly to Algorithm 4.1: We solve a regular Steiner tree problem and a shortest path problem to find the start and end conditions of our iterative process. However, we now start this process by setting $B = B_{min}$.

As before, we solve the RDCST problem for G with delay bound B next. If the newly found solution Sol is cheaper than the last solution we added to S, i.e., the last Pareto-efficient solution we found, we also add Sol to S. Since such a solution is guaranteed to be Pareto-efficient, solutions are never removed from S during the loop.

After every iteration, we increase the delay bound B by one and start a new iteration. Once we find a solution whose cost is equal to that of Sol_{STP}, we end the loop and check whether this solution dominates Sol_{STP}. If it does, we remove Sol_{STP} from S before returning the set S as the algorithm's result.

The two approaches differ mainly in the number of iterations that is necessary to find all efficient solutions, as well as their potential for reusing information from previous iterations in future ones (cf. Section 4.4).

4.1.3 Solving the Steiner tree problem

The two aforementioned approaches require the solution of both a shortest path and a Steiner tree problem before we can begin our iterative process. While the shortest path problem can easily be solved in polynomial time (e.g., by Dijkstra's algorithm), the Steiner tree problem is NP-hard and thus requires a different approach (cf. Chapter 3).

To solve the regular Steiner tree problem for our instances, we encode them as the following integer linear program, which is similar to the one used in [35, 64]:

We consider the directed variant of our BOSTPD instance, where each edge $\{i, j\} \in E$ has been replaced by two arcs $(i, j), (j, i) \in A$. The binary decision variables x_{ij} determine whether the arc $(i, j) \in A$ is part of the solution, while the variables y_i determine whether a potential Steiner vertex $i \in S$ is part of it. The sets $\delta^-(i)$ and $\delta^+(i)$ refer to the set of incoming and outgoing arcs of vertex i, respectively.

$$\min \sum_{(i,j)\in A} c_{ij} x_{ij} \tag{4.4}$$

$$\text{s.t.} \sum_{(i,j)\in\delta^-(j)} x_{ij} = 1 \qquad \forall j \in T \tag{4.5}$$

$$\sum_{(i,r)\in\delta^-(r)} x_{ir} = 0 \tag{4.6}$$

$$\sum_{(r,j)\in\delta^+(r)} x_{rj} \geq 1 \tag{4.7}$$

$$\sum_{(i,j)\in\delta^-(j)} x_{ij} = y_j \qquad \forall j \in S \tag{4.8}$$

$$\sum_{(j,i)\in\delta^+(j)} x_{ji} \geq y_j \qquad \forall j \in S \tag{4.9}$$

$$x_{ij} + x_{ji} \leq 1 \qquad \forall (i, j) \in A \tag{4.10}$$

$$\sum_{(i,j)\in A, i\in W, j\notin W} x_{ij} \geq 1 \qquad \forall W \subset V, r \in W, \overline{W} \cap T \neq \emptyset \tag{4.11}$$

$$x_{ij} \in \{0, 1\} \qquad \forall (i, j) \in A \tag{4.12}$$

$$y_i \in \{0, 1\} \qquad \forall i \in S \tag{4.13}$$

The inequalities (4.5) ensure that every terminal has exactly one incoming arc. Similarly, inequalities (4.6) and (4.7) ensure that the root vertex has no incoming arcs, but at least one outgoing arc. Inequalities (4.8) ensure that a selected Steiner vertex has exactly one incoming arc, whereas an unselected Steiner vertex has none. The constraints (4.9) then force the selection of at least one outgoing arc for a selected Steiner vertex. Constraints (4.10) forbid the selection of both an arc and its reverseS arc.

The exponentially large set of constraints described by (4.11) are so-called *directed cutset constraints*, which ensure that the solution is connected. Finally, constraints (4.12) and (4.13) ensure that the aforementioned decision variables are binary (i.e., integers taking either the value 0 or 1, corresponding to *false* and *true*, respectively).

4.1.3.1 Constraint separation

As noted above, the number of directed cutset constraints (4.11) is exponential in the size of G. Therefore, we cannot add all of them before starting our solution procedure, but instead only add them dynamically when they are violated by using branch and cut. Doing so requires a *separation procedure* to find inequalities in that set that are violated by a current candidate solution.

Depending on whether a candidate solution is integral or not, we use a different separation procedure to find violated inequalities.

Integer solutions For integer solutions, we use a separation procedure based on finding the connected components in the support graph defined by the current LP solution.

Let $(\mathbf{x}, \mathbf{y})$ be the current solution to the LP relaxation of the aforementioned ILP, i.e., the values assigned to variables $x_{ij}, \forall (i,j) \in A$, and $y_i, \forall i \in S$, by the LP solver. From this, we construct a support graph $G' = (V', A')$ as follows: $V' = \{r\} \cup \{i \in S | y_i \geq 1 - \varepsilon\} \cup T$ and $A' = \{(i,j) \in A | x_{ij} \geq 1 - \varepsilon\}$, where ε is an implementation-specific tolerance constant that allows the LP solver to deviate by at most $\pm\varepsilon$ from an integer value for reasons related to the limits of the numerical precision of floating point operations (i.e., any value in $[1 - \varepsilon, 1 + \varepsilon]$ is considered to be equal to 1). Intuitively, G' is the subgraph of G that consists of root vertex, all terminal vertices and those arcs and potential Steiner vertices that were selected by the LP solver.

Next, we find all weakly connected components $G'_i = (V'_i, A'_i), 0 \leq i \leq k$ of G' by using depth-first search (DFS) on an undirected version of our graph (i.e., disregarding the direction of its arcs) [34].

Once G' is partitioned into its connected components $G'_i, 0 \leq i \leq k$, we add the following cutset constraints to our ILP model:

For every connected component $G'_i, i \geq 1$ that contains at least one terminal vertex, we add the constraint

$$\sum_{(i,j) \in \delta^-(V'_i)} x_{ij} \geq 1$$

to our model. This forces the selection of at least one of the component's incoming arcs in every integral solution.

Additionally, if at least one such component exists, we add the constraint

$$\sum_{(i,j)\in\delta^+(V_0')} x_{ij} \geq 1$$

to our model, which, again, forces the selection of at least one of the root component's outgoing arcs in an integer solution.

If all terminal vertices are assigned to the root component, G' is a connected subgraph of G. We therefore stop the separation and accept it as a feasible solution w.r.t. the cutset constraints (4.11).

Fractional solutions For fractional solutions, we use a separation procedure that is based on finding minimum cuts between the root vertex and every terminal vertex.

Again, let $(\mathbf{x}, \mathbf{y})$ be the current solution for the LP relaxation of the ILP model. We now assign every arc $(i, j) \in A$ a capacity $\varsigma_{ij} = x_{ij}$ to construct a flow network $G_f = (V, A, \boldsymbol{\varsigma})$.

Next, we find the maximum flow f^* between r and every $t \in T$. Note that finding the maximum flow also gives us a minimum cut that separates V into the two disjoint subsets R (containing r) and $\bar{R}$ (containing t), with $\delta^+(R) = \delta^-(\bar{R})$ being the set of edges crossing the cut from R to $\bar{R}$, i.e., the cutset of the cut.

If $f^* < 1 - \varepsilon$, we add the constraint

$$\sum_{(i,j)\in\delta^+(R)} x_{ij} \geq 1$$

to our model.

If the flow between r and every t is greater than or equal to $1 - \varepsilon$, the current solution is connected and we stop the separation procedure.

Note, however, that since the variables x_{ij} may be fractional, the solution may not necessarily be a tree. If such a solution is encountered, branching on one of the fractional variables is necessary to proceed with the solution algorithm for the ILP.

In general, multiple minimum cuts between r and any t can exist. Our algorithm for finding the maximum flow [1], which is based on the push-relabel approach (cf. Section 2.7), finds both the first (the one closest to r) and the last (the one closest to t) cut between the two vertices. Therefore, unless these two cuts are identical, we add the appropriate cutset constraint for both of them to our model.

4.2 Preprocessing the individual RDCST instances

Before solving the RDCST instances that are generated by the ε-constraint method, we can apply preprocessing techniques to reduce their size and to facilitate the fast discovery of an optimal solution. Specifically, we can remove any vertex and any arc that is

- infeasible, i.e., cannot be part of any feasible solution

- suboptimal, i.e., cannot be part of an optimal solution.

All preprocessing techniques described in this section are taken from [55]. Note that while the input instance is always an undirected graph, we consider the transformed directed variant during preprocessing. This allows us, for instance, to remove only one of an edge's corresponding arcs if it is infeasible or suboptimal, but the reverse arc is not (i.e., the edge can only be traversed in one direction in a feasible or optimal solution, respectively).

4.2.1 Infeasible arcs

When analyzing the infeasibility of arcs, we need only consider the delays of a graph's arcs, since their costs do not have any bearing on the feasibility of a solution (only on its quality).

First, we delete every arc (i, j) with $d_{ij} > B$, i.e, every arc whose delay exceeds the delay bound by itself. Clearly, such an arc can never be part of a feasible solution, since selecting it always makes the root-terminal-path on which it lies infeasible. This step can be done at the start of the preprocessing, since the feasibility of an arc is determined only by its own delay and not in any part influenced by the graph's structure.

Next, we compute the shortest path w.r.t. delay from the root vertex r to every other vertex $i \in S \cup T$. Let d_i^{min} be the shortest delay with which we can reach vertex i in our graph. We can now delete every arc (i, j) for which $d_i^{min} + d_{ij} > B$ holds, because we can never select that arc in any feasible solution. This holds since we cannot reach its source vertex i with a delay of less than d_i^{min}, and even with this minimal delay to its source, selecting the arc itself always yields a path that exceeds the current delay bound B.

4.2.2 Infeasible vertices

First, note that we can obviously not remove any terminal vertex from our graph during preprocessing, since a feasible solution must contain all of them. Thus, we are only interested in finding potential Steiner vertices that cannot be part of a feasible solution.

Clearly, we can remove any vertex $i \in S$ that has no outgoing arcs, i.e., $|\delta^+(i)| = 0$, since in our ILP models, every Steiner vertex must have at least one selected outgoing arc. We enforce this constraint because a solution that has a Steiner vertex as a leaf can never be cheaper than that very same solution with the vertex and the arc connecting it to the rest of the tree removed (since arc costs are always non-negative).

By a similar argument, we can remove any potential Steiner vertex j with degree 1 on the corresponding undirected graph, i.e., only one incoming arc (i, j) and only one outgoing arc (j, i), which is the incoming arc's reverse. Again, our models forbid the selection of both arcs that correspond to an edge, since doing so can never yield an advantage over a solution that doesn't contain them.

Next, we compute the shortest path w.r.t. delay from every potential Steiner vertex $s \in S$ to every terminal vertex $t \in T$. Let d_s^T be the minimum delay with which we can reach any terminal vertex from s. We can now remove every vertex $s \in S$ for which $d_s^{min} + d_s^T > B$ holds, as every path from r to any $t \in T$ that passes through s exceeds the delay bound B. Since

every Steiner vertex that is selected must lie on a path between r and some t, and since s cannot have that property in any feasible solution, we may remove it from our graph.

4.2.3 Suboptimal arcs

In contrast to the previously mentioned infeasibility tests, we must consider both an arc's cost and its delay when attempting to prove that it cannot be part of an optimal solution, since even arcs with very high delay or cost can be part of an optimal solution if their other parameter is adequately low.

4.2.3.1 Suboptimality relative to root arcs

Consider an arbitrary arc $(i, j) \in A$. We may remove this arc from our graph if $c_{rj} \leq c_{ij}$ and $d_{rj} \leq d_i^{min} + d_{ij}$, i.e., if going to j directly from the root vertex r is neither more expensive nor longer w.r.t. delay than going via arc (i, j).

4.2.3.2 Suboptimality due to alternative path

To determine whether an arc (i, j) is suboptimal because of the existence of some other path $P(i, j)$ from i to j, we remove (i, j) from our graph temporarily and calculate the delay-constrained shortest path between i and j with d_{ij} as delay bound. If we find such a path whose total cost does not exceed c_{ij}, i.e., if $c_{P(i,j)} \leq c_{ij}$ and $d_{P(i,j)} \leq d_{ij}$, we can remove (i, j) from our graph permanently, since we can always substitute $P(i, j)$ for (i, j) in any solution and be guaranteed to not deteriorate our solution w.r.t. either cost or delay.

However, this last preprocessing step requires a potentially large computational effort, since we must calculate $O(|A|)$ delay-constrained shortest paths, which is a weakly NP-hard problem [10].

4.3 Solving the individual RDCST instances

To find solutions for the BOSTPD, we must solve instances of the RDCST as they are generated by the ε-constraint method described in Section 4.3. Since the RDCST is NP-hard [55], we do not know whether an algorithm exists that can solve it in polynomial time. Due to the good empirical performance of ILP approaches, we encode these instances as ILPs and solve them using a state of the art ILP solver.

The ILP models we used are the *path-cut formulation* and the *layered graph formulation*, both of which were originally proposed by Ruthmair [55].

4.3.1 Path-cut formulation

This model is similar to the ILP model for regular Steiner trees described in Section 4.1.3. Again, binary decision variables x_{ij} determine whether the arc $(i, j) \in A$ is part of the solution, binary decision variables y_i determine whether a potential Steiner vertex $i \in S$ is part of the solution, and $\delta^-(i)$ and $\delta^+(i)$ denote vertex i's set of incoming and outgoing arcs, respectively.

$$\min \sum_{(i,j)\in A} c_{ij} x_{ij} \tag{4.14}$$

$$\text{s.t.} \sum_{(i,j)\in\delta^-(j)} x_{ij} = 1 \qquad \forall i \in T \tag{4.15}$$

$$\sum_{(i,r)\in\delta^-(r)} x_{ir} = 0 \tag{4.16}$$

$$\sum_{(r,j)\in\delta^+(r)} x_{rj} \geq 1 \tag{4.17}$$

$$\sum_{(i,j)\in\delta^-(j)} x_{ij} = y_j \qquad \forall j \in S \tag{4.18}$$

$$\sum_{(i,j)\in\delta^+(i)} x_{ij} \geq y_i \qquad \forall i \in S \tag{4.19}$$

$$x_{ij} + x_{ji} \leq 1 \qquad \forall (i,j) \in A \tag{4.20}$$

$$\sum_{(i,j)\in A, i\in W, j\notin W} x_{ij} \geq 1 \qquad \forall W \subset V, s \in W, \overline{W} \cap T \neq \emptyset \tag{4.21}$$

$$\sum_{i=1}^{k} x_{v_i v_{i+1}} \leq k-2 \qquad \forall v_k \in T, P(v_1, v_k) \in P_{inf} \tag{4.22}$$

$$x_{ij} \in \{0,1\} \qquad \forall (i,j) \in A \tag{4.23}$$

$$y_i \in \{0,1\} \qquad \forall i \in S \tag{4.24}$$

Constraints (4.15) to (4.21) have the same interpretation as their equivalent constraints from the STP model, specifically (4.5) to (4.11). Similarly, constraints (4.23) and (4.24) impose the same range and integrality restrictions on their variables as constraints (4.12) and (4.13).

The key additions to this model are the constraints (4.22), which forbid all paths whose cumulative delay exceeds the delay bound B. Here, P_{inf} is the set of all paths that exceed this delay bound B. Recall from Section 1.2 that p_t describes a path between the root vertex r and a terminal vertex t, with $|p_t|$ describing the number of edges that p_t consists of. Since the number of potential paths between two vertices in a graph is exponential in its size, we cannot expect to add all these constraints before starting the solution procedure. Therefore, we have to use branch-and-cut to add them dynamically during the ILP solver's run.

4.3.1.1 Lifted path-cut inequalities

Based on previous work by Ascheuer et al. [2, 3] for the traveling salesman problem with time windows and by Kallehauge et al. [30] for the vehicle routing problem with time windows, Ruthmair argues that standard infeasible path constraints can be weak for the RDCST [55]. He therefore proposes the following set of strengthened infeasible path inequalities to improve the performance of a branch-and-cut algorithm for solving the path-cut ILP model.

First, let $P(v_1, v_k)$ be an infeasible path of length $k-1$, i.e., $\sum_{i=1}^{k-1} d_{v_i v_{i+1}} > B$. For this path, we define the following $k-2$ sets of vertices:

$$V_P^i = \{u \in V : (u, v_i) \in A, u \neq v_{i-1}, v_{i+1}, (u, v_i) \cup P(v_i, v_k) \in P_{inf}\} \qquad \forall i \in \{2, \ldots, k-1\}$$

Intuitively, V_P^i can be interpreted as follows: we consider all possible subpaths of $P(v_1, v_k)$ where arcs are removed from the front of the path, i.e., $P(v_2, v_k), \ldots, P(v_{k-1}, v_k)$. Let i be the index of the subpath's first vertex within $P(v_1, v_k)$, e.g., $i = 2$ for $P(v_2, v_k)$.

For each subpath, we consider all vertices $u \in V$ for which an arc (u, v_i) exists and for which u is not a predecessor or successor of v_i in $P(v_1, v_k)$. If the path consisting of (u, v_i) and the subpath $P(v_i, v_k)$ form an infeasible path themselves, i.e., $d_{uv_i} + \sum_{j=i}^{k-1} d_{v_j v_{j+1}} > B$, we add u to V_P^i.

The infeasible path inequalities described by (4.22) can now be strengthened by adding the x-variables corresponding to these (u, v_i) arcs to the left-hand side. Additionally, we add the x-variables corresponding to the reverse arcs of $P(v_1, v_k)$ on the same side. This yields the following set of lifted infeasible path inequalities:

$$\sum_{i=1}^{k-1} x_{v_i v_{i+1}} + \sum_{i=2}^{k-1} \sum_{u \in V_P^i} x_{uv_i} + \sum_{i=1}^{k-1} x_{v_{i+1} v_i} \leq k-2 \qquad \forall v_k \in T, P(v_1, v_k) \in P_{inf} \qquad (4.25)$$

4.3.1.2 Constraint separation

Both the directed cutset constraints defined by (4.21) and the (lifted) infeasible path inequalities defined by (4.22) and (4.25) are exponential in number and must therefore be added dynamically via branch-and-cut during the solution procedure.

The separation procedure for constraints (4.21) is identical to the one for constraints (4.11) of the STP model, which is described in Section 4.1.3.1. It is important that this set of constraints is separated *before* separating the infeasible path inequalities to ensure that the solution considered in the separation procedure of the (lifted) infeasible path inequalities is connected.

Once no more violated cutset constraints are found, we begin our search for infeasible paths. Note that the (lifted) infeasible path inequalities are only separated for integer solutions.

Again, let $(\mathbf{x}, \mathbf{y})$ be the current LP solution for our model. From this, we construct a support graph $G' = (V', A')$ only contains those vertices and arcs that are selected by $(\mathbf{x}, \mathbf{y})$, i.e., $V' = \{r\} \cup T \cup \{s \in S | y_s > \varepsilon\}$ and $A' = \{(i, j) \in A | x_{ij} > \varepsilon\}$.

We now compute the shortest path w.r.t. delay from r to every $t \in T$ in G'. If the delay of path $P(r, t)$ is $> B$, we consider it an infeasible path.

Next, we consider all subpaths $P(v_2, v_k), \ldots, P(v_{k-1}, v_k)$ of each infeasible path $P(v_1, v_k), v_1 = r, v_k = t$ by removing arcs from its front one by one. For each subpath $P(v_i, v_k)$, we compute the set V_P^i. Once we have all sets V_P^i, we add a lifted infeasible path constraint as defined by (4.25)for path $P(v_1, v_k)$ to our model.

When no more infeasible path inequalities are found for an integer solution $(\mathbf{x}, \mathbf{y})$, that solution is feasible for our model.

4.3.2 Layered graph formulation

While the path-cut formulation enforces the delay constraints by explicitly forbidding all paths with a delay greater than the delay bound B, the *layered graph formulation* (4.26) – (4.42) enforces them implicitly via its underlying data structure. As the name suggests, this data structure is a layered graph as introduced in Section 2.4.

In addition to the binary arc decision variables $x_{ij}, \forall (i,j) \in A$, for all arcs and the binary vertex decision variables $y_i, \forall i \in S$, for all potential Steiner vertices in the original graph G, this model uses binary arc decision variables $x_{ij}^l, \forall (i_l, j_k) \in A_L$, for the layered graph G_L's arcs and binary vertex decision variables $y_i^l, \forall i_l \in V_L \setminus \{r\}$, for its vertices. Note that in contrast to the y_i variables, y_i^l variables exist for all vertices of the layered graph, not only its potential Steiner vertices.

Variable y_i^l is set to 1 if and only if vertex i is part of the solution and the path from the root vertex r to i has delay l. Similarly, x_{ij}^l is set to 1 if and only if arc (i,j) is part of the solution and the paths from r to i and j have delay l and $l + d_{ij}$, respectively.

$$\min \sum_{(i,j)\in A} c_{ij} x_{ij} \tag{4.26}$$

$$\text{s.t.} \sum_{(i,j)\in\delta^-(i)} x_{ij} = 1 \qquad \forall i \in T \tag{4.27}$$

$$\sum_{(i,j)\in\delta^-(r)} x_{ij} = 0 \tag{4.28}$$

$$\sum_{(i,j)\in\delta^+(r)} x_{ij} \geq 1 \tag{4.29}$$

$$\sum_{(i,j)\in\delta^-(i)} x_{ij} = y_i \qquad \forall i \in S \tag{4.30}$$

$$\sum_{(i,j)\in\delta^+(i)} x_{ij} \geq y_i \qquad \forall i \in S \tag{4.31}$$

$$x_{ij} + x_{ji} \leq 1 \qquad \forall (i,j) \in A \tag{4.32}$$

$$\sum_{i_l \in V_L} y_i^l = 1 \qquad \forall i \in T \tag{4.33}$$

$$\sum_{i_l \in V_L} y_i^l = y_i \qquad \forall i \in S \tag{4.34}$$

$$x_{ri}^0 = x_{ri} \qquad \forall (r,i) \in A \tag{4.35}$$

$$\sum_{(i_k,j_l)\in A_L} x_{ij}^k = x_{ij} \qquad \forall (i,j) \in A, i \neq r \tag{4.36}$$

$$\sum_{(i_k,j_l)\in A_L} x_{ij}^k = y_i^l \qquad \forall i_l \in V_L \setminus \{r\} \tag{4.37}$$

$$\sum_{(i_k,j_l)\in A_L, i\neq h} x_{ij}^k \geq x_{jh}^l \qquad \forall (j_l, h_g) \in A_L^g \tag{4.38}$$

$$x_{ij} \in \{0,1\} \qquad \forall (i,j) \in A \tag{4.39}$$

$$y_i \in \{0,1\} \qquad \forall i \in V \tag{4.40}$$

$$x_{ij}^k \in \{0,1\} \qquad \forall (i_k, j_l) \in A_L \tag{4.41}$$

$$y_i^l \in \{0,1\} \qquad \forall i_l \in V_L \setminus \{r\} \tag{4.42}$$

Constraints (4.27) to (4.32) correspond to their equivalent constraints from the path-cut model, specifically (4.15) to (4.20).

Equations (4.33) and (4.34) ensure for every terminal vertex, exactly one of its layered copies is selected, whereas for potential Steiner vertices, exactly one copy is selected if and only if the corresponding original vertex is selected as well.

Constraints (4.35) ensure that a root arc in the layered graph is selected if and only if the corresponding arc was selected in the original graph. Similarly, constraints (4.36) ensure that for every arc in the original graph that was selected, exactly one of its layered copies is selected as well.

Equations (4.37) ensure that every vertex in the layered graph that is selected has exactly one incoming edge, whereas non-selected vertices have no incoming edges. Finally, inequalities (4.38) ensure that for every selected arc in the layered graph, at least one of its source vertex' incoming arcs is selected as well. Since the layered graph is acyclic, this ensures its connectivity and, because of the arc linking constraints, the connectivity of the original graph as well.

Similar to the path-cut model, constraints (4.39) to (4.42) ensure that all variables are binary.

4.3.2.1 Valid inequalities

To strengthen the layered graph model, the following directed cutset inequalities are added to it. Note that because both these sets of constraints are exponential in size, their constraints should be added dynamically via branch-and-cut.

$$\sum_{(i,j)\in A, i\in W, j\notin W} x_{ij} \geq 1 \qquad \forall W \subset V, r \in W, \overline{W} \cap T \neq \emptyset \tag{4.43}$$

$$\sum_{(i_k,j_l)\in A_L, i_k\in W_L, j_l\notin W_L} x_{ij}^k \geq 1 \qquad \forall W_L \subset V_L, r \in W_L, \overline{W_L} \cap T_L \neq \emptyset \tag{4.44}$$

Constraints (4.43) correspond to the cutset constraints (4.21) of the path-cut model and work on the original graph's arc variables. In contrast, constraints (4.44) work on the layered graph's variables.

4.3.2.2 Constraint separation

First, note that both (4.43) and (4.44) must only be separated for fractional solutions, since inequalities (4.38) ensure that every integral solution is both connected and cycle-free, which implies that no violated cutset constraints can be found.

For the cutset constraints (4.43) on the original arc variables, the separation procedure is, again, identical to the one for the regular Steiner tree model's cutset constraints (4.11), which is described in Section 4.1.3.1.

In case we do not find any violated cutset constraints on the original graph, we search for violated ones on the layered graph. To do that, we first augment our layered graph with one additional artificial vertex for every terminal vertex in our original graph. Each of the vertices corresponding to the original terminal vertex is connected to the new artificial terminal vertex by a new outgoing arc with capacity 1.

Formally, let $G' = (V', A')$ be the augmented layered graph, with $V' = V_L \cup \{i' : i \in T\}$ and $A' = A_L \cup \{(i_l, i') : i \in T, 1 \leq l \leq B\}$. Furthermore, let $\varsigma_{i_l i'} = 1$.

We now compute the maximum flow between r_L and each artificial vertex corresponding to a terminal vertex in the original graph. If the maximum flow is less than $1 - \varepsilon$, we have found a minimum cut with capacity less than that. Let A_{cut} be the cutset corresponding to that cut. We now add the constraint

$$\sum_{(i,j) \in A_{cut}} x_{ij} \geq 1$$

to our model.

4.4 Reusing information throughout the iterations

Throughout the run of our ε-constraint algorithm, we must solve a number of instances for the RDCST problem. These instances differ in the delay bound B (and therefore possibly in the number of arcs that can be removed by preprocessing), but are otherwise identical. Thus, the idea of reusing data from RDCST instance i and its solution when solving instance $i + 1$ seems reasonable. We specifically focus on reusing previous partial and complete solutions (see Section 4.4.1), the instance graph (see Section 4.4.2) and previously identified violated inequalities (see Section 4.4.3). We discuss these aspects for both the low-to-high and the high-to-low delay bound approach.

4.4.1 Reusing previous solutions

4.4.1.1 Low-to-high delay bound

When starting at the lowest possible delay bound and increasing it during successive iterations, we can directly reuse any solution from a previous iteration, since any solution that is feasible for delay bound B is automatically feasible for delay bound $B + 1$. Thus, the optimal solution for delay bound B serves as the starting solution for the next iteration with delay bound $B + 1$.

4.4.1.2 High-to-low delay bound

If, on the other hand, we start at the highest possible delay bound and progressively reduce it during subsequent iterations, directly reusing a previous solution is impossible, since we explic-

itly set the new delay bound to be one less than the delay required by the previous solution. Thus, that solution is infeasible for the next iteration.

We therefore have to repair the solution before being able to reuse it. Let $(\mathbf{x}, \mathbf{y})$ be the variable values corresponding to the previous iteration's solution Sol_{old} and B_{old} be the maximum delay of a path from r to any terminal vertex t.

Our aim is to find a solution Sol' that is similar to Sol_{old}, but feasible for the new delay bound $B_{old} - 1$. Let T_{inf} be the set of terminal vertices that are reached with delay B_{old} in Sol_{old}. We construct Sol' as follows:

First, we set $Sol' = Sol_{old}$. Next, we remove the incoming arcs of every $t \in T_{inf}$ from Sol'. If this results in any potential Steiner vertices becoming leaf vertices in our solution, we recursively remove them and their incoming arcs from the solution until no more such vertices exist.

Once this is done, we compute the delay-constrained shortest path (with delay bound $B_{old} - 1$) P_t from r to every $t \in T_{inf}$ and add it to Sol'. If P_t contains a vertex v' that is already part of Sol', this may introduce a cycle into our solution. We prevent this by removing the previous incoming arc of v' from Sol', again recursively removing potential Steiner leaf vertices as necessary. This yields a feasible solution for the RDCST with delay bound $B_{old} - 1$.

4.4.2 Reusing the previous iteration's graph

In principle, all iterations operate on the same original graph – the input graph of the BOSTPD. However, because of the different delay bounds, preprocessing may have removed different vertices and arcs.

4.4.2.1 Low-to-high delay bound

Since the delay bound increases from one iteration to the next, arcs and vertices that were removed during the previous iteration's preprocessing because of infeasibility or suboptimality might now no longer be either. Thus, we cannot reuse that already preprocessed graph and must restart the preprocessing from the original input graph. For the same reason, we cannot reuse the layered graph and have to recreate it from scratch after preprocessing.

4.4.2.2 High-to-low delay bound

Here, the delay bound decreases with subsequent iterations. Thus, any arcs and vertices removed during the last iteration's preprocessing would still be infeasible and/or suboptimal. We therefore use that graph as a starting point for the current iteration's preprocessing stage.

Similarly, a layered graph from the previous iteration can be reused for the current iteration with delay bound B. After removing all those arcs and vertices from it whose corresponding original arcs and vertices were removed during this iteration's preprocessing, we remove all arcs and vertices belonging to layers greater than B from last iteration's layered graph and recursively remove all Steiner vertices with no outgoing arcs (if any). The resulting layered graph is a fully preprocessed input graph for the layered graph model.

4.4.3 Reusing cuts from previous iterations

Whenever an inequality that can be reused in any way is found by a separation procedure, it is added to a global pool of inequalities. At the beginning of each iteration, before solving its ILP model, we add all those inequalities from the pool to our model that can be reused and discard the rest.

4.4.3.1 Infeasible path cuts

Low-to-high delay bound Here, some of the previously added infeasible path cuts may become invalid in the new model, since the paths forbidden by them may now be feasible, thanks to the increased delay bound. Specifically, those paths with length $B_{old} + 1 = B_{new}$ were forbidden in the previous model, but are now feasible. Thus, we cannot reuse these inequalities in this solution approach.

High-to-low delay bound Since the delay bound is decreasing during subsequent iterations, all infeasible path cuts from previous iterations remain valid, since a path that is invalid for delay bound B_{old} is certainly invalid for all delay bounds less than that. However, some of the arcs of some paths may have been removed during the current preprocessing state, thus rendering the corresponding path cut obsolete, since it is now trivially always satisfied. These obsolete cuts can be removed from the pool of previous cuts. All others are added to the current model.

4.4.3.2 Directed connection cuts

Low-to-high delay bound Since new arcs may be added during subsequent iterations, connection cuts from previous iterations cannot be reused, as these new arcs are not covered by them. Selecting one of these new arcs for the new solution would not satisfy these constraints, even if it were part of the new cutset for the cut that the old arcs were the cutset for (since the constraint would demand that one of the old arcs be used for connecting the two sets of the cut).

High-to-low delay bound All connection cut inequalities remain valid when the delay bound is reduced, since this step can only remove arcs from the graph. The removed arcs' decision variable would simply always be set to 0, which does not influence the fact that one of the other arcs in the constraint has to be selected. However, for performance reasons, we remove these decision variables from the constraint before adding it to the new model.

Let C_{old} be the set of arcs corresponding to the original constraint's cutset in graph $G = V, A)$. From this, we construct a new set of arcs C_{new}, corresponding to the same cutset in the new graph $G' = (V', A')$ as follows:

$$C_{new} = \{(i, j) \in C_{old} : (i, j) \in A'\}.$$

CHAPTER 5

Computational results for the Bi-objective Steiner Tree Problem with Delays

In this chapter, we present a performance evaluation of the solution algorithms for the BOSTPD that we described in Chapter 4. We compare the performance of the two approaches for the ε-constraint method (high-to-low and low-to-high delay bound), the performance of both ILP models (infeasible path and layered graph model), as well as the effects of computational enhancements such as preprocessing, solution and cut reuse on the runtime performance.

5.1 Implementation details and test setup

All parts of the aforementioned solution algorithm were implemented in C++11 and compiled with Clang 3.4, using optimization level `-O3`.

We used the following external libraries and components:

Component	*used for*
CPLEX 12.5	LP solver, branch-and-cut
OGDF 2012.07	graph data structures, Dijkstra's algorithm, connected components
BOOST 1.55	command line parameter handling
push-relabel maxflow CS2 [1]	maximum flow/minimum cut

All test runs were performed on Intel Xeon E5-2670v2 10-core CPUs, each running at 2.5GHz. Each run was limited to one concurrent thread, as well as a total runtime of 10000 seconds.

Additionally, we configured CPLEX to use at most 3GiB of memory for the branch-and-cut search tree and to not add any strengthening cuts on its own. A full list of the parameters we set can be seen in Table 5.1.

Table 5.1: CPLEX parameters

CPLEX Parameter	**value**	*effect*
Threads	1	limits CPLEX to one thread
TreLim	3072	limits B&C tree size to 3GiB
WorkMem	3072	limits working memory of CPLEX to 3GiB
EachCutLim	0	disables all non-Gomory cut generation
FracCuts	-1	disables Gomory fractional cut generation

5.2 Test instances

Our test instances can be partitioned into the following two classes:

Class R: randomly generated complete graphs The instances in this class were generated by us, similar to the self-generated random instances for the RDCSTP in [55]. They are partitioned into three sets according to their size, which can be 10, 20 or 50 vertices. Since all instances are based on a complete graphs, this automatically determines their number of edges.

For each size category, five different graphs were generated, each of which assigns a cost $c_e \in \{1, 2, \ldots, 100\}$ to each edge at random. From each of these graphs, four actual BOSTPD instances were created by assigning, respectively, a delay $d_e \in \{1, \ldots, 5\}$, $\{1, \ldots, 10\}$, $\{1, \ldots, 20\}$ and $\{1, \ldots, 50\}$ to each edge at random. The maximum of an instance's set of candidate edge delays is referred to as that instance's *maximum edge delay* d_e^{max}, whereas the number of the graph from which it was generated is that instances *index* i. Cost and delay of an edge are assigned independently from each other, and are not correlated.

For all instances in class R, the first vertex is selected as the root vertex.

We refer to the set of all random instances of a given size as R<size>, e.g., R10. Subsets of these sets consisting of all instances with the same d_e^{max} are referred to as R<size>-D<d_e^{max}> (e.g., R10-D20), whereas subsets containing all instances with equal index are referred to as R<size>-<index> (e.g., R10-1). A specific instance is referred to as R<size>-<index>-D<d_e^{max}> (e.g., R10-1-D20).

Class T: instances by Gouveia et al. The instances in this class were proposed by Gouveia et al. in [25]. Like our randomly generated instances, they are based on complete graphs and can be partitioned into two sets according to their size. While we refer to these size categories as 20 and 40, they actually contain 21 and 41 vertices, respectively, since the root vertex is not included in the first number.

These instance sets can be further partitioned into three structural subsets each, depending on how their edges are assigned their corresponding edge costs.

- structure R: Like for our own randomly generated instances, the edges of these instances are assigned an edge cost $c_e \in \{1, \ldots, 100\}$ at random.

- structure C: For these instances, each non-root vertex is assigned hypothetical coordinates in two-dimensional Euclidean space at random. An edge's cost is then set according to the Euclidean distance between its two incident vertices. Finally, the root vertex is placed in the center of the other vertices, with the cost of its incident edges being set according to the metric described above.

- structure E: These instances are constructed like those from subset C, with the only difference between them being that the root vertex in these instances is placed on a corner of the non-root vertices instead of at the center.

For each combination of size and structure, five distinct graphs were generated. From each of these, four BOSTPD instances were created by assigning, respectively, a delay $d_e \in \{1, 2\}$, $\{1, \ldots, 5\}$, $\{1, \ldots, 10\}$ and $\{1, \ldots, 100\}$ to each edge at random.

We refer to the set of all instances in this class of a given size as T<size>, e.g., T20. Similarly, we refer to the set of all instances with a given structure as T<structure>, e.g., TR, and to the set of instances with a given size and structure as T<structure><size>, e.g., TR20. For each of these sets, subsets containing all instances with a given d_e^{max} are referred to as <setname>-D<d_e^{max}>, e.g., TR20-D5 or T40-D100, subsets containing all instances with a given index are referred to as <setname>-<index>, e.g., TE40-2 or TC-3, and individual instances are referred to as <setname>-<index>-D<d_e^{max}>, e.g., TR40-1-D10.

For all instances of both classes, the set of terminal vertices consists of the first 40% of vertices. However, the method of determining these first 40% differs between the two classes.

For instances in class R, all vertices (including the root vertex r) are considered when determining which vertices are terminals and which are not. Since r is always the first vertex, it is selected as a terminal vertex for every instance. Thus, for each instance in R10, R20 and R50, its set of terminal vertices consists of the root vertex r and the next 3, 7 or 19 vertices, respectively.

For instances in class T, only the non-root vertices are considered when determining which vertices are terminals. However, to remain consistent with instances from class R, the root vertex is also considered to be a terminal vertex. Therefore, the set of terminal vertices for instances in T20 and T40 consists of the root vertex and the next 8 or 16 vertices, respectively.

Note, however, that even though the root vertex is considered to be a part of the set of terminal vertices, it is treated differently by our program than the other *true* terminal vertices, most notable in the ILP models (where r must not have an incoming edge, whereas the other terminals must have one).

5.3 Analysis of the individual algorithms

This section analyzes the runtime performance of the algorithms that we developed and implemented during the course of this thesis. While all these algorithms are based on the general procedure laid out in Chapter 4, they differ in two aspects:

1. Whether they explore an instance's delay interval $[d^{min}, d^{max}]$, in which Pareto-efficient solutions may be found, in a high-to-low (cf. Section 4.1.1) or a low-to-high (cf. Section 4.1.2) delay bound order.

2. Whether they solve the resulting subproblems, which are instances of the RDCSTP, by using the pathcut (cf. Section 4.3.1) or the layered graph (cf. Section 4.3.2) ILP formulation.

We will consider the following three algorithms:

algorithm PC uses the pathcut formulation and explores $[d^{min}, d^{max}]$ in a high-to-low delay bound order.

algorithm LH uses the layered graph formulation and explores $[d^{min}, d^{max}]$ in a high-to-low delay bound order.

algorithm LL uses the layered graph formulation and explores $[d^{min}, d^{max}]$ in a low-to-high delay bound order.

Since the data that we gathered for algorithm PC suggests that using the pathcut formulation without reusing previously found cuts will likely result in bad runtime performance, and since such reuse is not possible when exploring the objective space in a low-to-high order, we decided against implementing an algorithm that uses the pathcut model in that order.

For each of these algorithms, we will consider three algorithmic variants, which differ in the types of computational enhancements that are used to speed up the solution procedure.

basic (B) variant We only solve the instance's corresponding ILP, including all strengthening and lifted inequalities, with CPLEX. Thus, the individual iterations of the ε-constraint method are completely independent from each other.

preprocessing (P) variant At the beginning of each iteration, the instance is preprocessed (cf. Section 4.2) according to the current delay bound B before solving its corresponding ILP.

reuse (R) variant After preprocessing, we provide CPLEX with an initial feasible solution to our problem that is based on the previous iteration's solution (cf. Section 4.4.1). We also add all inequalities found by previous branch-and-cut iterations to our ILP model before solving it.

A specific variant of a particular algorithm is called an *approach*, which we refer to by its name that consists of the algorithm's and the variant's name in the following way: <algorithm><variant>. For instance, the basic variant of the pathcut algorithm would be named **PCB**.

The results of our test runs are presented in tables throughout this section. Each table contains the following columns:

Set The set of instances to which the instance belongs (cf. Section 5.2). It describes an instance's class, its size and, for instances of class T, its structure.

i The instance's index.

d_e^{max} The maximum delay that can be assigned to an edge from this instance.

#Sol The number of Pareto-efficient solutions that exist for this instance. If the instance could not be solved to proven optimality by any approach, this field contains a dash (–).

d^{min} The minimum delay that any feasible solution to the BOSTPD may have. This is the lowest delay bound that we need to consider during our ε-constraint method's iterations. We find this value by calculating the shortest path w.r.t. delay from r to every $t \in T$ (cf. Section 4.1.1).

d^{max} An upper bound on the maximum delay that any efficient solution to the BOSTPD may have. It is the highest delay bound that we need to consider during our ε-constraint method's iterations. Thus, in order to be guaranteed to find the complete Pareto frontier, we must only consider the integer values in $[d^{min}, d^{max}]$ as possible delay bounds. We calculate d^{max} by solving the regular STP for the instance (cf. Section 4.1.1).

#Solutions The number of Pareto-efficient solutions found by the current approach. If the approach found the complete Pareto frontier, the field contains an asterisk (*).

%$[d^{min}, d^{max}]$ The percentage of $[d^{min}, d^{max}]$ that the current approach explored, i.e., for which it has found optimal solutions (Note that the solutions that are found need not be efficient to count towards this percentage). If the approach found the complete Pareto frontier, the field contains an asterisk (*).

#Iterations The number of ε-constraint method iterations that the approach completed, i.e., for which it has found optimal solutions. Recall that not all of these solutions, which are optimal for the subproblem they solve, are necessarily Pareto-efficient, since a solution from another iteration might dominate them. Note that this number includes the iteration where only the regular STP is solved, before starting the actual ε-constraint method.

time The time in seconds that the approach needed to solve the problem, rounded to the nearest 100 milliseconds. If the time limit was reached, the field contains a dash (–), whereas it contains a plus sign (+) if the memory limit was reached.

5.3.1 Algorithm PC

This subsection describes the computational results for the algorithm PC, which uses the pathcut formulation and explores the objective space in a high-to-low order. All relevant data from the associated test runs can be found in Tables 5.2 to 5.6, where the instances are grouped by class, size and, if relevant, structure.

First, we can see in Table 5.2 that each approach solves all instances in R10 to proven optimality. For each instance, the algorithm runs for at most 100 milliseconds and only needs the minimum number of iterations, namely the number of Pareto-efficient solutions that exists

for the instance, to find the complete Pareto frontier. We assume that the short runtimes are due to the instances' comparatively small sizes, which is obviously a major factor in determining the difficulty of solving an instance to proven optimality.

Starting with the instances in set R20, the runtimes of the three approaches start to diverge. As Table 5.3 shows, the basic approach PCB now clearly requires more time to solve each instance than its counterparts, sometimes quite significantly so, and even fails to solve one instance to proven optimality (although it still manages to explore over 90% of the relevant delay interval $[d^{min}, d^{max}]$). The other two approaches still manage to solve every single instance to proven optimality and outperform PCB on those instances where all three approaches find the complete Pareto frontier. However, while PCP and PCR are quite similar to each other in terms of runtime, PCR is always faster. This nicely illustrates how preprocessing can speed up the solution process and how reusing information can improve on this even further.

It is at this instance set that the algorithm starts to find non-efficient solutions, as illustrated by the number of iterations required by the approaches (which is sometimes larger than **#Sol**). Interestingly, PCR performs the largest number of iterations of all approaches for some instances, but still solves it fastest.

We also note that d_e^{max} seems to have little to no effect on the algorithm's runtime. While instances with large d_e^{max} sometimes require more time to be solved, this is far from universal. Also, the most difficult instances are not the ones with the largest d_e^{max}. Clearly, other factors play a much more important role here.

The instances of set R50 are where our algorithm starts to hit the limits of its computational capabilities. Within our imposed time and memory limits, only PCR was still able to solve even a single instance (it solved three), while the others failed at every single one of them, mostly due to reaching the time limit.

In terms of the size of the explored delay interval, PCR again shows the best performance, but the other two approaches are usually not far behind. We assume that this is due to the fact that all approaches start hitting the "tough" subinstances at roughly the same point (as illustrated by the fact that their number of iteration' is mostly very similar). Here, PCR can leverage previous information to still solve a few of these, while the other approaches fail a little earlier.

Again, we observe no significant impact of d_e^{max} on the runtime.

For the instances in set TR20 (see Table 5.4), our algorithm shows a similar performance to that for the instances from set R20. Since these instance sets are constructed similarly (i.e., selecting cost and delay of each edge at random) and since their instances are of similar size, this is to be expected.

Again, PCB solves all instances but one (where it still explores over 90% of the delay interval), while the others find the complete Pareto frontier for each of them. By preprocessing each instance before each iteration, PCP easily outperforms the basic approach, often by quite a significant margin. PCR shows an even more pronounced improvement and commonly solves instances in just half the time required by PCP.

The instances in sets TC20 and TE20, for which the results are given in Tables 5.4 and 5.5, are the first ones that have Euclidean, rather than random edge cost. The pathcut formulation is clearly able to utilize this to its advantage, as our algorithm shows a significant improvement over that for instances TC20. Approaches PCP and PCR are again very similar in terms of

runtime, with PCR being faster mostly for the harder instances. Both approaches solve most instances very fast, with only two of them requiring significantly more than 10 seconds to be solved. In contrast, PCB requires several minutes to solve the hardest instances and, in one case, fails to find the complete Pareto frontier (even though it, again, comes close).

As noted above, we assume that the significant improvement in performance is due to the ILP formulation used by the algorithm. Since the edge costs are assigned based on the Euclidean distance between its two incident vertices, only few paths with low cost exist between the root and each terminal. Thus, few pathcut inequalities must be added to the model before branch-and-cut converges towards an optimal solution. This would also explain how PCB often comes very close to finding the complete Pareto frontier, but falls short by just a few percent. Due to the lower delay bound at later instances, a much larger number of these low-cost paths must be forbidden via pathcut inequalities before a feasible solution can be found. In contrast, both PCP and PCR can use preprocessing to remove a number of these arcs, again reducing the number of feasible low-cost paths that need to be explored.

For the instances in sets T40, the algorithm again starts to show its limitations. Notably, TR40 is the first set of instances where the three approaches differ significantly in the number of instances they are still able to solve to proven optimality. As can be seen in Table 5.5, PCB fails on all but two instances, whereas PCP and PCR still solve ten and eleven, respectively. Whenever all approaches fail to solve an instance, they always explore roughly the same part of the relevant delay interval before reaching the time or memory limit, which seems to support our previous interpretation that they are all running into a series of difficult to solve subinstances where the pathcut approach simply does not perform well in general. PCR again mostly outperforms PCP in terms of runtime (for instances where both approaches find the complete Pareto frontier) or explored delay interval (for instances where they do not), but is slower by almost 1000 seconds in one case.

As for the instances from set T20, the algorithm's performance changes significantly when dealing with Euclidean instances. As Table 5.6 shows, PCR solves every instance to proven optimality, PCP solves all but two and PCB is still able to solve at least half of them. For instances where all approaches find the complete Pareto frontier, they show the usual runtime behaviour where PCR is fastest, PCP is between somewhat and significantly slower and PCB is the slowest. However, PCP again outperforms PCR in a few select cases.

Of the instances from TE40, only one instance can still be solved by our pathcut algorithm (specifically, by PCR, which still needs more than two hours to do so). The approaches commonly explore a similar fraction of the delay interval before reaching the time (mostly for the instances with low d_e^{max}) or the memory limit (mostly for instances with high d_e^{max}). Interestingly, PCR seems to run out of memory more frequently than the other approaches.

For both TC40 and TE40, when an approach fails to find the complete Pareto frontier, it either fails towards the end of the delay interval due to reaching the time limit, or at the very start due to running out of memory for the branch-and-cut tree. Also, while the algorithm explores more than 90% of the delay interval for instances in TC40 where it fails due to reaching the time limit, that figure is much lower for instances in TE40 (commonly around 70%, but as low as 57%). Finally, failure due to running out of memory is significantly more common for instances with high d_e^{max} than for those with low d_e^{max}, where the algorithm more frequently reaches the

Table 5.2: Computational results for algorithm PC for instance set R10

						#Solutions			% $[d^{min}, d^{max}]$			#Iterations			time [s]		
Set	i	d_e^{max}	#Sol	d^{min}	d^{max}	PCB	PCP	PCR	PCB	PCP	PCR	PCB	PCP	PCR	PCB	PCP	PCR
R10	1	5	7	3	9	*	*	*	*	*	*	7	7	7	0.1	0.0	0.0
		10	5	4	9	*	*	*	*	*	*	5	5	5	0.1	0.0	0.0
		20	11	8	58	*	*	*	*	*	*	11	11	11	0.1	0.1	0.1
		50	8	21	81	*	*	*	*	*	*	8	8	8	0.1	0.0	0.0
	2	5	1	1	1	*	*	*	*	*	*	1	1	1	0.0	0.0	0.0
		10	2	6	8	*	*	*	*	*	*	2	2	2	0.0	0.0	0.0
		20	2	8	14	*	*	*	*	*	*	2	2	2	0.0	0.0	0.0
		50	4	19	49	*	*	*	*	*	*	4	4	4	0.0	0.0	0.0
	3	5	6	4	16	*	*	*	*	*	*	6	6	6	0.1	0.0	0.0
		10	7	6	24	*	*	*	*	*	*	7	7	7	0.1	0.0	0.0
		20	7	5	34	*	*	*	*	*	*	7	7	7	0.1	0.0	0.0
		50	5	9	71	*	*	*	*	*	*	5	5	5	0.0	0.0	0.0
	4	5	3	2	4	*	*	*	*	*	*	3	3	3	0.0	0.0	0.0
		10	5	5	9	*	*	*	*	*	*	5	5	5	0.0	0.0	0.0
		20	3	7	16	*	*	*	*	*	*	3	3	3	0.0	0.0	0.0
		50	5	22	50	*	*	*	*	*	*	5	5	5	0.1	0.0	0.0
	5	5	4	3	13	*	*	*	*	*	*	4	4	4	0.0	0.0	0.0
		10	8	5	20	*	*	*	*	*	*	8	8	8	0.1	0.0	0.0
		20	7	11	52	*	*	*	*	*	*	7	7	7	0.1	0.0	0.0
		50	10	13	117	*	*	*	*	*	*	10	10	10	0.1	0.1	0.0

time limit.

We assume that the observed runtime behaviour is explained by the instances' larger sizes when compared to those from T20. In combination with a high d_e^{max}, it leads to a comparatively larger number of low-cost paths from root to the terminals whose delay exceeds B and which must therefore be forbidden by adding appropriate pathcut inequalities. However, as opposed to T20, these do not constrain the search space significantly enough to allow for a quick identification of the optimal solution. Instead, branching must be used extensively, which results in large branch-and-cut trees and long runtimes. Since the root in TE40 instances is located on a corner of the other vertices, the required paths to all terminals are even longer than for instances in TC40, which explains the algorithm's worse performance on them.

In conclusion, PCR offers the best runtime performance of the three approaches that we analyzed in this section. Not only does it find the complete Pareto frontier for the largest number of instances, it is also able to find them most quickly in almost all cases. Choosing any other approach does not seem to be reasonable in any case.

We also observe that the pathcut algorithm performs very well on instances with Euclidean edge costs, especially if they are not too large. It also appears to be largely unaffected by an instance's d_e^{max}, even for instances in T40, where it mostly solves either all or none of the instances within a given subset.

Table 5.3: Computational results for algorithm PC for instance sets R20 and R50

						#Solutions			% $[d^{min}, d^{max}]$			#Iterations			time [s]		
Set	i	d_e^{max}	#Sol	d^{min}	d^{max}	PCB	PCP	PCR	PCB	PCP	PCR	PCB	PCP	PCR	PCB	PCP	PCR
R20	1	5	11	4	18	*	*	*	*	*	*	11	11	11	12.8	2.7	0.6
		10	12	6	27	*	*	*	*	*	*	12	12	12	32.0	8.4	2.3
		20	16	4	50	*	*	*	*	*	*	16	16	17	6.7	1.5	1.0
		50	23	13	99	*	*	*	*	*	*	23	23	23	214.2	9.2	3.1
	2	5	12	3	18	*	*	*	*	*	*	12	12	13	3.4	0.9	0.9
		10	15	4	43	*	*	*	*	*	*	18	18	16	509.7	4.1	3.7
		20	14	6	75	*	*	*	*	*	*	17	17	15	3.0	0.5	0.2
		50	21	14	188	*	*	*	*	*	*	22	22	23	14.2	2.5	1.6
	3	5	11	3	15	*	*	*	*	*	*	11	11	11	35.6	8.8	3.6
		10	18	5	41	*	*	*	*	*	*	20	20	18	70.0	8.7	8.5
		20	24	9	73	*	*	*	*	*	*	25	25	27	2718.5	150.9	79.5
		50	28	17	123	*	*	*	*	*	*	28	28	28	556.3	67.0	34.6
	4	5	9	3	13	*	*	*	*	*	*	9	9	9	7.7	0.2	0.1
		10	19	5	31	*	*	*	*	*	*	20	20	21	67.5	1.0	0.5
		20	20	8	54	*	*	*	*	*	*	20	20	20	15.9	2.6	1.1
		50	26	21	126	19	*	*	91	*	*	22	28	27	–	54.0	31.3
	5	5	8	2	16	*	*	*	*	*	*	10	10	10	1.7	0.1	0.1
		10	18	3	39	*	*	*	*	*	*	21	21	19	13.2	1.5	1.3
		20	13	9	49	*	*	*	*	*	*	14	14	13	3.6	0.4	0.2
		50	17	20	183	*	*	*	*	*	*	19	18	18	167.2	0.6	0.4
R50	1	5	18	2	38	8	8	9	70	73	76	13	14	12	–	–	–
		10	33	4	85	11	13	14	71	72	76	15	16	18	–	+	–
		20	–	4	173	17	17	19	74	74	76	25	26	34	–	–	–
		50	–	8	364	15	16	19	56	58	64	21	22	27	–	–	–
	2	5	14	3	20	8	8	*	72	72	*	11	11	16	–	–	9288.2
		10	20	3	33	8	8	9	65	65	68	11	11	11	–	–	–
		20	35	6	73	8	8	10	57	57	62	11	11	13	–	–	–
		50	–	12	270	13	13	17	66	66	70	16	16	22	–	–	–
	3	5	14	3	24	9	8	10	82	77	86	11	10	14	–	–	–
		10	19	3	40	11	13	*	82	87	*	13	15	20	–	–	6446.9
		20	35	7	104	12	12	13	77	77	79	19	19	19	–	–	–
		50	46	10	207	17	21	25	82	84	86	19	23	30	–	–	–
	4	5	22	3	46	14	14	14	84	84	84	19	19	18	–	–	+
		10	22	3	49	10	10	14	74	74	85	13	13	16	–	–	–
		20	38	4	130	15	15	16	76	76	79	21	21	21	–	–	–
		50	51	10	341	23	24	28	86	86	89	30	30	34	–	+	–
	5	5	15	3	24	11	11	*	86	86	*	13	13	18	–	–	2081.0
		10	20	3	53	9	10	12	76	82	86	16	17	16	–	–	–
		20	39	5	85	15	17	19	70	73	77	21	22	27	–	+	–
		50	57	8	234	12	12	14	68	68	72	18	18	20	–	–	–

Table 5.4: Computational results for algorithm PC for instance sets TR20 and TC20

						#Solutions			% $[d^{min}, d^{max}]$			#Iterations			time [s]		
Set	i	d_e^{max}	#Sol	d^{min}	d^{max}	PCB	PCP	PCR	PCB	PCP	PCR	PCB	PCP	PCR	PCB	PCP	PCR
TR20	1	2	3	1	4	*	*	*	*	*	*	3	3	3	2.0	0.1	0.0
		5	7	3	9	*	*	*	*	*	*	7	7	7	7.7	0.2	0.1
		10	13	4	27	*	*	*	*	*	*	13	13	13	152.4	28.3	6.5
		100	36	38	240	21	*	*	91	*	*	23	37	36	–	76.0	62.6
	2	2	10	2	14	*	*	*	*	*	*	10	10	10	9.2	2.4	1.0
		5	12	3	21	*	*	*	*	*	*	12	12	12	16.1	3.3	1.7
		10	13	4	36	*	*	*	*	*	*	14	14	14	31.9	4.1	1.9
		100	28	23	436	*	*	*	*	*	*	30	30	30	535.5	74.4	28.4
	3	2	7	2	8	*	*	*	*	*	*	7	7	7	7.8	1.7	1.1
		5	9	3	12	*	*	*	*	*	*	9	9	9	10.4	4.1	2.4
		10	12	4	26	*	*	*	*	*	*	13	13	14	8.7	1.2	0.4
		100	17	23	216	*	*	*	*	*	*	20	20	20	23.8	1.2	0.6
	4	2	9	2	14	*	*	*	*	*	*	10	10	9	31.4	6.3	2.4
		5	14	3	25	*	*	*	*	*	*	16	16	14	256.7	107.1	35.7
		10	19	4	48	*	*	*	*	*	*	21	21	22	278.3	29.5	13.3
		100	30	23	450	*	*	*	*	*	*	34	34	33	2030.7	480.4	291.7
	5	2	9	2	13	*	*	*	*	*	*	10	10	10	29.4	1.4	0.6
		5	13	3	21	*	*	*	*	*	*	13	13	13	45.4	10.7	6.8
		10	20	4	46	*	*	*	*	*	*	21	21	20	316.3	26.4	12.1
		100	31	23	375	*	*	*	*	*	*	32	33	36	134.3	33.8	18.6
TC20	1	2	4	1	4	*	*	*	*	*	*	4	4	4	0.2	0.1	0.1
		5	7	3	10	*	*	*	*	*	*	7	7	7	0.6	0.2	0.2
		10	9	4	24	*	*	*	*	*	*	10	10	10	3.4	0.3	0.3
		100	22	38	192	*	*	*	*	*	*	23	25	24	510.9	2.5	1.2
	2	2	8	2	13	*	*	*	*	*	*	10	10	9	11.6	2.1	1.1
		5	13	3	23	*	*	*	*	*	*	15	15	16	84.7	6.2	7.5
		10	15	4	33	*	*	*	*	*	*	18	18	16	50.0	2.3	1.8
		100	20	23	353	*	*	*	*	*	*	25	25	24	43.1	11.8	5.8
	3	2	4	2	8	*	*	*	*	*	*	6	6	6	0.8	0.2	0.1
		5	6	3	15	*	*	*	*	*	*	7	7	7	3.1	0.5	0.4
		10	10	4	30	*	*	*	*	*	*	12	12	11	15.1	1.1	0.6
		100	19	23	188	*	*	*	*	*	*	23	23	22	35.3	10.1	3.2
	4	2	4	2	5	*	*	*	*	*	*	4	4	4	0.1	0.1	0.1
		5	6	3	12	*	*	*	*	*	*	9	9	8	0.5	0.2	0.1
		10	11	4	27	*	*	*	*	*	*	13	13	12	1.6	0.3	0.2
		100	13	23	167	*	*	*	*	*	*	15	15	15	7.9	0.4	0.3
	5	2	4	2	5	*	*	*	*	*	*	4	4	4	0.1	0.0	0.1
		5	8	3	14	*	*	*	*	*	*	8	8	8	0.3	0.2	0.1
		10	9	4	25	*	*	*	*	*	*	12	11	11	0.8	0.2	0.2
		100	17	23	195	*	*	*	*	*	*	18	18	18	2.2	0.3	0.3

Table 5.5: Computational results for algorithm PC for instance sets TE20 and TR40

						#Solutions			% $[d^{min}, d^{max}]$			#Iterations			time [s]		
Set	i	d_e^{max}	#Sol	d^{min}	d^{max}	PCB	PCP	PCR	PCB	PCP	PCR	PCB	PCP	PCR	PCB	PCP	PCR
TE20	1	2	4	1	5	*	*	*	*	*	*	5	5	5	100.3	75.2	37.3
		5	6	3	11	*	*	*	*	*	*	8	8	8	46.3	1.0	0.4
		10	11	4	28	*	*	*	*	*	*	13	13	13	757.7	5.9	3.2
		100	35	38	303	29	*	*	97	*	*	34	37	40	–	56.8	28.7
	2	2	8	2	13	*	*	*	*	*	*	10	10	9	9.6	2.1	1.1
		5	13	3	23	*	*	*	*	*	*	15	15	16	68.2	6.0	6.6
		10	15	4	33	*	*	*	*	*	*	18	18	16	41.7	2.3	1.7
		100	20	23	353	*	*	*	*	*	*	25	25	24	45.7	11.8	5.8
	3	2	5	2	8	*	*	*	*	*	*	6	6	6	6.8	0.9	1.0
		5	7	3	14	*	*	*	*	*	*	9	9	8	25.5	5.3	4.0
		10	9	4	24	*	*	*	*	*	*	14	14	11	6.2	2.4	0.9
		100	18	23	274	*	*	*	*	*	*	22	22	20	26.7	3.9	2.2
	4	2	5	2	10	*	*	*	*	*	*	6	6	6	6.4	0.8	0.7
		5	8	3	23	*	*	*	*	*	*	10	10	10	29.4	3.0	1.9
		10	10	4	38	*	*	*	*	*	*	13	13	13	13.5	1.1	0.6
		100	10	23	258	*	*	*	*	*	*	11	11	13	7.9	0.3	0.2
	5	2	5	2	6	*	*	*	*	*	*	5	5	5	1.6	0.3	0.3
		5	9	3	16	*	*	*	*	*	*	9	9	10	4.9	1.1	0.7
		10	11	4	26	*	*	*	*	*	*	11	11	12	2.6	1.0	0.5
		100	16	23	256	*	*	*	*	*	*	20	19	17	6.2	2.8	1.1
TR40	1	2	9	2	11	7	5	*	80	70	*	7	6	10	+	–	1022.3
		5	16	3	26	11	12	12	83	88	88	16	17	16	–	–	–
		10	19	4	49	10	9	13	83	80	89	15	14	18	–	–	–
		100	–	20	541	1	1	1	0	0	0	1	1	1	+	+	+
	2	2	10	2	17	7	*	*	88	*	*	10	12	13	–	1548.3	953.6
		5	16	3	29	12	*	*	89	*	*	14	17	19	–	4420.1	2199.9
		10	21	4	49	14	*	*	85	*	*	17	23	25	–	4016.1	1273.3
		100	35	20	377	1	*	*	0	*	*	1	40	41	+	397.4	128.7
	3	2	9	2	13	5	5	5	75	75	75	6	6	7	–	–	–
		5	15	3	27	6	6	7	68	68	72	10	10	10	–	–	+
		10	28	4	45	9	9	10	57	57	60	11	11	13	–	–	–
		100	–	20	495	1	1	1	0	0	0	1	1	1	+	+	+
	4	2	10	2	11	7	*	*	80	*	*	8	10	10	–	2812.1	3793.6
		5	13	3	23	*	*	*	*	*	*	14	14	13	7065.0	138.1	30.6
		10	24	4	48	1	*	*	2	*	*	1	24	25	+	7406.8	5592.0
		100	–	20	338	1	1	1	0	0	0	1	1	1	+	+	+
	5	2	7	2	10	*	*	*	*	*	*	8	8	8	4590.2	427.6	257.7
		5	13	3	22	11	*	*	95	*	*	13	14	15	–	274.0	58.5
		10	23	4	34	1	*	*	3	*	*	1	25	26	+	750.1	517.9
		100	–	20	364	1	1	1	0	0	0	1	1	1	+	+	+

Table 5.6: Computational results for algorithm PC for instance sets TC40 and TE40

						#Solutions			% $[d^{min}, d^{max}]$			#Iterations			time [s]		
Set	i	d_e^{max}	#Sol	d^{min}	d^{max}	PCB	PCP	PCR	PCB	PCP	PCR	PCB	PCP	PCR	PCB	PCP	PCR
TC40	1	2	10	2	14	*	*	*	*	*	*	11	11	11	495.4	145.1	214.7
		5	21	3	36	*	*	*	*	*	*	23	23	24	433.7	110.5	132.0
		10	21	4	54	17	*	*	94	*	*	23	26	25	–	4103.5	2306.3
		100	65	20	515	1	*	*	0	*	*	1	74	71	+	6535.6	2286.1
	2	2	10	2	15	*	*	*	*	*	*	11	11	11	661.1	358.0	634.4
		5	15	3	31	12	*	*	93	*	*	15	17	17	+	180.0	73.0
		10	19	4	51	16	*	*	96	*	*	21	23	23	+	150.8	85.0
		100	53	20	501	1	1	*	0	0	*	1	1	61	+	+	6314.3
	3	2	7	2	11	*	*	*	*	*	*	9	9	9	162.8	18.1	7.2
		5	15	3	25	*	*	*	*	*	*	19	19	18	1771.3	251.1	193.6
		10	18	4	42	*	*	*	*	*	*	21	21	27	2910.9	111.7	59.9
		100	48	20	299	35	*	*	92	*	*	44	59	58	–	1667.2	4552.2
	4	2	7	2	11	*	*	*	*	*	*	8	8	8	411.9	39.8	31.1
		5	12	3	20	*	*	*	*	*	*	14	14	14	3092.4	20.5	21.1
		10	19	4	45	1	*	*	2	*	*	1	23	23	+	135.4	47.9
		100	53	20	377	1	1	*	0	0	*	1	1	60	+	+	2576.8
	5	2	7	2	9	*	*	*	*	*	*	8	8	7	349.2	10.8	6.2
		5	12	3	20	*	*	*	*	*	*	13	13	12	143.8	16.9	8.7
		10	17	4	29	1	*	*	4	*	*	1	19	20	+	321.2	195.8
		100	44	20	295	1	*	*	0	*	*	1	44	45	+	1170.0	740.4
TE40	1	2	12	2	15	5	5	6	57	57	64	7	7	8	–	–	–
		5	20	3	33	11	11	10	74	74	71	17	17	14	–	–	+
		10	27	4	57	13	13	13	76	76	76	18	18	19	–	–	+
		100	–	20	490	1	1	1	0	0	0	1	1	1	+	+	+
	2	2	8	2	13	5	5	4	83	83	75	8	8	7	+	–	+
		5	20	3	26	15	15	*	83	83	*	17	17	21	–	–	8544.7
		10	24	4	41	8	8	9	55	55	63	15	15	15	–	–	+
		100	–	20	421	1	1	1	0	0	0	1	1	1	+	+	+
	3	2	10	2	13	6	6	5	75	75	67	9	9	7	–	–	–
		5	18	3	36	7	6	7	71	68	71	10	9	10	–	–	–
		10	–	4	52	1	1	1	2	2	2	1	1	1	+	+	+
		100	–	20	691	1	1	1	0	0	0	1	1	1	+	+	+
	4	2	10	2	17	5	5	5	75	75	75	7	7	7	–	–	–
		5	24	3	37	11	12	12	66	69	69	14	15	15	–	–	–
		10	26	4	47	1	1	1	2	2	2	1	1	1	+	+	+
		100	–	20	497	1	1	1	0	0	0	1	1	1	+	+	+
	5	2	9	2	15	5	5	5	79	79	79	8	8	8	–	–	–
		5	22	3	34	13	13	14	78	78	81	20	20	20	–	–	–
		10	26	4	59	1	1	1	2	2	2	1	1	1	+	+	+
		100	–	20	507	1	1	1	0	0	0	1	1	1	+	+	+

5.3.2 Algorithm LH

This subsection describes the computational results for the algorithm LH, which uses the layered graph formulation and explores the objective space in a high-to-low order. All data from the test runs of this algorithm can be found in Tables 5.7 to 5.11, where the instances are grouped just like in the previous section.

Similar to the results for the algorithm PC, Table 5.7 shows that all three approaches of this algorithm are able to solve every instance in R10 very quickly. They similarly do not find any non-efficient solutions.

The instances from R20 can also be solved to proven optimality by each of the approaches (see Table 5.8). All three show very similar performance over all these instances, with LHB and LHP being almost identical in terms or runtime in every case. LHR is often able to outperform the two, but requires slightly more time for one instance.

An instance's d_e^{max} value now plays a much more pronounced role in determining the runtime required for findings its Pareto frontier. Notably, instances with $d_e^{max} = 50$ are always the hardest to solve. We assume that this is due to the fact that a high d_e^{max} leads to root-terminal paths with high delay, some of which might also have a comparably low cost. This leads to a large delay interval with high d^{max}, which not only increases the number of iterations that we will likely have to perform, but also increases the delay bound B of the first iterations of the ε-constraint method. Since the size of a layered graph (and thus, the size of the corresponding ILP) depends on B almost as much as it depends on $|V|$, it stands to reason that increasing this parameter makes the problem more difficult so solve.

The algorithm's high dependence on the layered graph's size becomes even more apparent for instances from R50. While all approaches can still solve at least half of the instances to proven optimality and LHR manages to solve all but three, they require a significant amount of time to do so. This is especially true of those instances with high d_e^{max}, which require hundreds to thousands of seconds to be solved. LHB and LHP again show very similar performance, which indicates that preprocessing is not very effective when used in conjunction with the layered graph formulation on random instances. In contrast, LHR shows a consistent performance gain compared to the other two approaches, sometimes being able to find the complete Pareto frontier thousands of seconds faster.

Failures to solve an instance to proven optimality happen due to both reaching the time and memory limit, with the time limit being the more frequently occurring of the two. Also, if an approach fails to solve an instance, it mostly fails in the very first iteration. This is likely due to the fact that the first iteration of the ε-constraint method is the most difficult one, since it has the largest delay bound and therefore the largest size. This in turn leads to a large ILP, which become more difficult to solve as their size increases.

Instances from the sets T20, for with the results can be found in Tables 5.9 and 5.10, again show a runtime behaviour similar to the one for instances R20. With the exception of two instances that are only solved by LHR, all instances are solved by all approaches, which again show very similar performance. LHB and LHP mostly need approximately equal time to solve each instance, with both sometimes outperforming the other depending on the specific instance (which, again, indicates that preprocessing is of little help here). LHR mostly finds the Pareto frontier faster than the other two approaches, especially for those instances with high d_e^{max}.

The runtimes for solving instances from TC20 and TE20 are comparable to those of equivalent instances from TR20, which indicates that the underlying structure of an instance's graph does not have a large impact on the runtime required for solving it. In contrast, an instance's d_e^{max} strongly influences the difficulty of solving an instance, with those instances with $d_e^{max} = 100$ consistently being the most difficult to solve among their corresponding subset. This is consistent with our previous observations, which indicated that d_e^{max} is strongly correlated with d^{max}. This, in turn, determines the delay bound of the first iteration's instance, which directly influences the size of the layered graph, its corresponding ILP and, finally, the time required for solving it.

The results for instances in TR40, which are given in Table 5.10, further support our previous findings. LHB and LHP show similar performance on most instances (with LHB mostly outperforming LHP, further suggesting that preprocessing does not work well in conjunction with the layered graph formulation), whereas LHR is noticeably faster than the other two approaches in every case.

The runtimes of the algorithm are, again, highly dependent on an instance's d_e^{max}. Notably, in all but one case, the three approaches fail to solve the instances with $d_e^{max} = 100$ due to running out of memory, likely for the aforementioned reasons.

The algorithm performs even worse on the Euclidean instances of T40 than it does on its random ones. As shown in Table 5.11, LHB and LHR are each only able to solve nine of the instances in TC40 and two of those in TE40, whereas LHR manages to solve all instances except those with $d_e^{max} = 100$ (and one where it is 10). The observed runtimes again increase rapidly with increasing d_e^{max}, resulting in the inability to solve many of the instances to proven optimality. Failures to do so are mostly due to running out of memory during the first iteration, although the time limit is still reached for some instances.

In conclusion, LHR consistently offers the best runtime performance of the three approaches for algorithm LH. While our tests have shown that preprocessing does not significantly improve the performance of our algorithm, the reuse of information still offers an improvement.

The performance of the algorithm is highly dependent on the parameter d_e^{max}, since it influences the size of the layered graphs and their corresponding ILPs that need to be solved during the individual iterations of the ε-constraint method. It contrast to algorithm PC, it works worse on instances with Euclidean edge costs than it does on random ones.

Table 5.7: Computational results for algorithm LH for instance set R10

						#Solutions			% $[d^{min}, d^{max}]$			#Iterations			time [s]		
Set	i	d_e^{max}	#Sol	d^{min}	d^{max}	LHB	LHP	LHR	LHB	LHP	LHR	LHB	LHP	LHR	LHB	LHP	LHR
R10	1	5	7	3	9	*	*	*	*	*	*	7	7	7	0.0	0.0	0.0
		10	5	4	9	*	*	*	*	*	*	5	5	5	0.0	0.0	0.0
		20	11	8	58	*	*	*	*	*	*	11	11	11	0.6	0.6	0.5
		50	8	21	81	*	*	*	*	*	*	8	8	8	0.2	0.2	0.2
	2	5	1	1	1	*	*	*	*	*	*	1	1	1	0.0	0.0	0.0
		10	2	6	8	*	*	*	*	*	*	2	2	2	0.0	0.0	0.0
		20	2	8	14	*	*	*	*	*	*	2	2	2	0.0	0.0	0.0
		50	4	19	49	*	*	*	*	*	*	4	4	4	0.0	0.0	0.0
	3	5	6	4	16	*	*	*	*	*	*	6	6	6	0.1	0.1	0.1
		10	7	6	24	*	*	*	*	*	*	7	7	7	0.1	0.1	0.1
		20	7	5	34	*	*	*	*	*	*	7	7	7	0.1	0.1	0.1
		50	5	9	71	*	*	*	*	*	*	5	5	5	0.5	0.5	0.4
	4	5	3	2	4	*	*	*	*	*	*	3	3	3	0.0	0.0	0.0
		10	5	5	9	*	*	*	*	*	*	5	5	5	0.0	0.0	0.0
		20	3	7	16	*	*	*	*	*	*	3	3	3	0.0	0.0	0.0
		50	5	22	50	*	*	*	*	*	*	5	5	5	0.1	0.1	0.1
	5	5	4	3	13	*	*	*	*	*	*	4	4	4	0.0	0.0	0.0
		10	8	5	20	*	*	*	*	*	*	8	8	8	0.1	0.1	0.1
		20	7	11	52	*	*	*	*	*	*	7	7	7	0.1	0.2	0.1
		50	10	13	117	*	*	*	*	*	*	10	10	10	1.4	1.4	1.4

Table 5.8: Computational results for algorithm LH for instance sets R20 and R50

						#Solutions			% $[d^{min}, d^{max}]$			#Iterations			time [s]		
Set	i	d_e^{max}	#Sol	d^{min}	d^{max}	LHB	LHP	LHR	LHB	LHP	LHR	LHB	LHP	LHR	LHB	LHP	LHR
R20	1	5	11	4	18	*	*	*	*	*	*	11	11	11	1.0	1.0	0.8
		10	12	6	27	*	*	*	*	*	*	13	13	13	6.4	6.4	2.1
		20	16	4	50	*	*	*	*	*	*	16	16	16	3.8	3.6	3.0
		50	23	13	99	*	*	*	*	*	*	23	23	23	24.2	23.9	20.1
	2	5	12	3	18	*	*	*	*	*	*	13	13	13	1.1	1.2	0.9
		10	15	4	43	*	*	*	*	*	*	16	16	17	14.6	14.5	4.4
		20	14	6	75	*	*	*	*	*	*	15	15	15	5.7	5.4	4.4
		50	21	14	188	*	*	*	*	*	*	23	23	23	112.3	108.9	40.2
	3	5	11	3	15	*	*	*	*	*	*	11	11	11	1.3	1.2	0.9
		10	18	5	41	*	*	*	*	*	*	20	20	19	8.0	7.8	3.6
		20	24	9	73	*	*	*	*	*	*	25	25	26	28.9	29.1	21.1
		50	28	17	123	*	*	*	*	*	*	28	28	28	47.5	46.1	36.1
	4	5	9	3	13	*	*	*	*	*	*	9	9	9	0.3	0.3	0.3
		10	19	5	31	*	*	*	*	*	*	19	19	20	3.2	3.3	2.8
		20	20	8	54	*	*	*	*	*	*	20	20	20	6.7	7.1	6.2
		50	26	21	126	*	*	*	*	*	*	26	26	27	36.0	35.9	35.9
	5	5	8	2	16	*	*	*	*	*	*	8	8	8	0.3	0.3	0.3
		10	18	3	39	*	*	*	*	*	*	20	20	19	3.3	3.4	2.9
		20	13	9	49	*	*	*	*	*	*	14	14	13	2.4	2.4	2.1
		50	17	20	183	*	*	*	*	*	*	18	18	19	17.9	17.2	19.6
R50	1	5	18	2	38	0	0	*	3	3	*	1	1	22	+	+	391.9
		10	33	4	85	0	0	*	1	1	*	1	1	40	–	–	4933.3
		20	–	4	173	0	0	2	1	1	20	1	1	4	–	–	–
		50	–	8	364	1	1	1	0	0	0	1	1	1	+	+	+
	2	5	14	3	20	*	*	*	*	*	*	15	15	15	41.9	44.1	15.6
		10	20	3	33	*	*	*	*	*	*	22	22	22	97.0	110.2	68.4
		20	35	6	73	*	*	*	*	*	*	38	38	37	2061.5	1832.8	587.5
		50	–	12	270	0	0	3	0	0	51	1	1	6	–	–	–
	3	5	14	3	24	*	*	*	*	*	*	15	15	14	41.3	42.4	16.3
		10	19	3	40	*	*	*	*	*	*	19	19	20	136.3	148.0	56.3
		20	35	7	104	1	1	*	2	2	*	2	2	41	–	–	2168.1
		50	46	10	207	0	1	*	1	7	*	1	2	53	–	–	4953.9
	4	5	22	3	46	*	*	*	*	*	*	24	24	24	2260.8	3244.0	74.4
		10	22	3	49	*	*	*	*	*	*	24	24	23	389.6	425.1	79.8
		20	38	4	130	0	0	*	1	1	*	1	1	45	–	–	3821.7
		50	51	10	341	1	1	*	0	0	*	1	1	55	+	+	6030.9
	5	5	15	3	24	*	*	*	*	*	*	16	16	15	55.0	52.0	20.9
		10	20	3	53	*	*	*	*	*	*	26	26	25	5107.1	5332.7	280.6
		20	39	5	85	*	*	*	*	*	*	42	42	43	3237.8	3990.7	576.1
		50	57	8	234	0	0	*	0	0	*	1	1	65	–	–	7073.2

Table 5.9: Computational results for algorithm LH for instance sets TR20 and TC20

						#Solutions			% $[d^{min}, d^{max}]$			#Iterations			time [s]		
Set	i	d_e^{max}	#Sol	d^{min}	d^{max}	LHB	LHP	LHR	LHB	LHP	LHR	LHB	LHP	LHR	LHB	LHP	LHR
TR20	1	2	3	1	4	*	*	*	*	*	*	3	3	3	0.1	0.0	0.0
		5	7	3	9	*	*	*	*	*	*	7	7	7	0.2	0.2	0.2
		10	13	4	27	*	*	*	*	*	*	13	13	13	1.6	1.6	1.7
		100	36	38	240	*	*	*	*	*	*	36	36	36	104.2	138.5	125.5
	2	2	10	2	14	*	*	*	*	*	*	10	10	10	0.9	0.9	0.8
		5	12	3	21	*	*	*	*	*	*	12	12	12	2.2	2.2	1.6
		10	13	4	36	*	*	*	*	*	*	15	15	16	5.9	5.2	4.5
		100	28	23	436	*	*	*	*	*	*	30	30	29	1203.9	1122.5	642.7
	3	2	7	2	8	*	*	*	*	*	*	7	7	7	0.3	0.3	0.2
		5	9	3	12	*	*	*	*	*	*	9	9	9	0.5	0.5	0.5
		10	12	4	26	*	*	*	*	*	*	13	13	13	2.8	2.5	2.0
		100	17	23	216	*	*	*	*	*	*	19	19	20	137.9	130.8	138.2
	4	2	9	2	14	*	*	*	*	*	*	10	10	10	2.6	2.6	1.1
		5	14	3	25	*	*	*	*	*	*	14	14	16	7.7	8.1	4.3
		10	19	4	48	*	*	*	*	*	*	20	20	21	39.4	42.9	10.8
		100	30	23	450	*	*	*	*	*	*	33	33	35	2164.4	1888.4	1354.9
	5	2	9	2	13	*	*	*	*	*	*	11	11	11	2.4	2.4	1.0
		5	13	3	21	*	*	*	*	*	*	13	13	13	2.7	2.6	1.8
		10	20	4	46	*	*	*	*	*	*	21	21	21	15.3	15.6	7.2
		100	31	23	375	*	*	*	*	*	*	35	35	34	1223.4	1364.1	585.2
TC20	1	2	4	1	4	*	*	*	*	*	*	4	4	4	0.0	0.0	0.0
		5	7	3	10	*	*	*	*	*	*	7	7	7	0.2	0.2	0.2
		10	9	4	24	*	*	*	*	*	*	10	10	10	2.1	2.1	1.0
		100	22	38	192	*	*	*	*	*	*	25	25	24	54.4	46.6	47.6
	2	2	8	2	13	*	*	*	*	*	*	11	11	10	5.3	5.6	1.4
		5	13	3	23	*	*	*	*	*	*	14	14	15	17.7	17.4	4.4
		10	15	4	33	*	*	*	*	*	*	18	18	18	16.5	15.9	10.7
		100	20	23	353	*	*	*	*	*	*	25	25	25	4536.5	5728.7	993.9
	3	2	4	2	8	*	*	*	*	*	*	6	6	6	1.5	1.4	0.2
		5	6	3	15	*	*	*	*	*	*	7	7	7	4.1	4.0	0.5
		10	10	4	30	*	*	*	*	*	*	14	14	11	177.1	161.2	1.4
		100	19	23	188	*	*	*	*	*	*	24	24	21	1376.9	1408.2	75.9
	4	2	4	2	5	*	*	*	*	*	*	4	4	4	0.0	0.0	0.0
		5	6	3	12	*	*	*	*	*	*	8	8	8	0.4	0.4	0.2
		10	11	4	27	*	*	*	*	*	*	12	12	13	15.5	15.1	1.1
		100	13	23	167	*	*	*	*	*	*	15	15	16	56.1	58.7	39.4
	5	2	4	2	5	*	*	*	*	*	*	4	4	4	0.1	0.0	0.0
		5	8	3	14	*	*	*	*	*	*	8	8	8	2.3	2.2	0.3
		10	9	4	25	*	*	*	*	*	*	11	11	11	16.0	17.1	0.8
		100	17	23	195	*	*	*	*	*	*	18	18	18	715.9	750.2	56.9

Table 5.10: Computational results for algorithm LH for instance sets TE20 and TR40

						#Solutions			% $[d^{min}, d^{max}]$			#Iterations			time [s]		
Set	i	d_e^{max}	#Sol	d^{min}	d^{max}	LHB	LHP	LHR	LHB	LHP	LHR	LHB	LHP	LHR	LHB	LHP	LHR
TE20	1	2	4	1	5	*	*	*	*	*	*	5	5	5	0.8	0.8	0.1
		5	6	3	11	*	*	*	*	*	*	8	8	8	9.1	8.1	0.5
		10	11	4	28	*	*	*	*	*	*	13	13	12	83.0	70.9	2.8
		100	35	38	303	0	0	*	0	0	*	1	1	38	–	–	219.0
	2	2	8	2	13	*	*	*	*	*	*	11	11	10	5.3	5.1	1.4
		5	13	3	23	*	*	*	*	*	*	14	14	15	17.8	17.8	4.9
		10	15	4	33	*	*	*	*	*	*	18	18	18	18.2	18.2	7.8
		100	20	23	353	*	*	*	*	*	*	25	25	25	5102.1	5215.6	937.5
	3	2	5	2	8	*	*	*	*	*	*	6	6	6	1.5	1.5	0.3
		5	7	3	14	*	*	*	*	*	*	9	9	9	9.6	9.4	1.7
		10	9	4	24	*	*	*	*	*	*	13	13	11	76.5	71.6	3.6
		100	18	23	274	0	0	*	0	0	*	1	1	23	–	–	485.4
	4	2	5	2	10	*	*	*	*	*	*	7	7	7	5.6	5.8	0.3
		5	8	3	23	*	*	*	*	*	*	10	10	10	29.2	30.9	1.9
		10	10	4	38	*	*	*	*	*	*	14	14	12	151.5	149.3	7.5
		100	10	23	258	*	*	*	*	*	*	13	13	12	1006.2	1076.8	128.7
	5	2	5	2	6	*	*	*	*	*	*	5	5	5	0.2	0.2	0.2
		5	9	3	16	*	*	*	*	*	*	10	10	9	3.5	4.0	1.5
		10	11	4	26	*	*	*	*	*	*	12	12	11	11.1	10.2	2.5
		100	16	23	256	*	*	*	*	*	*	19	19	19	546.9	473.8	183.2
TR40	1	2	9	2	11	*	*	*	*	*	*	10	10	9	6.1	5.9	2.5
		5	16	3	26	*	*	*	*	*	*	19	19	18	148.9	149.6	24.3
		10	19	4	49	*	*	*	*	*	*	25	25	24	568.5	631.2	151.2
		100	–	20	541	1	1	1	0	0	0	1	1	1	+	+	+
	2	2	10	2	17	*	*	*	*	*	*	12	12	12	92.7	86.3	9.3
		5	16	3	29	*	*	*	*	*	*	21	21	16	333.1	354.0	19.3
		10	21	4	49	*	*	*	*	*	*	24	24	23	980.8	971.3	56.2
		100	35	20	377	1	1	*	0	0	*	1	1	42	+	+	3614.2
	3	2	9	2	13	*	*	*	*	*	*	9	9	9	11.4	11.4	3.1
		5	15	3	27	*	*	*	*	*	*	16	16	16	62.0	58.8	23.2
		10	28	4	45	*	*	*	*	*	*	31	31	30	1288.3	1286.4	210.3
		100	–	20	495	1	1	1	0	0	0	1	1	1	+	+	+
	4	2	10	2	11	*	*	*	*	*	*	10	10	10	5.6	5.9	3.0
		5	13	3	23	*	*	*	*	*	*	14	14	14	53.7	55.6	11.8
		10	24	4	48	*	*	*	*	*	*	24	24	25	202.1	220.6	104.7
		100	–	20	338	1	1	1	0	0	0	1	1	1	+	+	+
	5	2	7	2	10	*	*	*	*	*	*	8	8	8	1.6	1.5	1.1
		5	13	3	22	*	*	*	*	*	*	13	13	15	16.0	16.1	8.6
		10	23	4	34	*	*	*	*	*	*	26	26	25	67.4	76.1	31.7
		100	–	20	364	1	1	1	0	0	0	1	1	1	+	+	+

Table 5.11: Computational results for algorithm LH for instance sets TC40 and TE40

						#Solutions			% $[d^{min}, d^{max}]$			#Iterations			time [s]		
Set	i	d_e^{max}	#Sol	d^{min}	d^{max}	LHB	LHP	LHR	LHB	LHP	LHR	LHB	LHP	LHR	LHB	LHP	LHR
TC40	1	2	10	2	14	*	*	*	*	*	*	11	11	11	1238.8	1289.3	9.7
		5	21	3	36	0	0	*	3	3	*	1	1	23	+	+	37.9
		10	21	4	54	0	0	*	2	2	*	1	1	25	+	+	192.3
		100	65	20	515	1	1	1	0	0	0	1	1	1	+	+	+
	2	2	10	2	15	0	0	*	7	7	*	1	1	11	–	–	7.7
		5	15	3	31	0	0	*	3	3	*	1	1	16	+	+	26.3
		10	19	4	51	0	0	*	2	2	*	1	1	21	+	+	102.4
		100	53	20	501	1	1	1	0	0	0	1	1	1	+	+	+
	3	2	7	2	11	*	*	*	*	*	*	9	9	9	47.1	48.8	3.3
		5	15	3	25	*	*	*	*	*	*	18	18	18	2352.6	2286.9	27.4
		10	18	4	42	*	*	*	*	*	*	23	23	25	9734.8	9378.6	188.4
		100	48	20	299	0	0	3	0	0	27	1	1	8	–	–	–
	4	2	7	2	11	*	*	*	*	*	*	8	8	8	271.1	269.7	1.3
		5	12	3	20	*	*	*	*	*	*	13	13	14	1712.9	1819.6	7.0
		10	19	4	45	0	0	*	2	2	*	1	1	22	+	+	33.9
		100	53	20	377	1	1	1	0	0	0	1	1	1	+	+	+
	5	2	7	2	9	*	*	*	*	*	*	7	7	7	32.9	35.1	1.1
		5	12	3	20	*	*	*	*	*	*	12	12	13	652.6	622.3	10.5
		10	17	4	29	*	*	*	*	*	*	19	19	18	1364.9	1365.4	26.9
		100	44	20	295	1	1	1	0	0	0	1	1	1	+	+	+
TE40	1	2	12	2	15	1	1	*	29	29	*	4	4	13	–	–	574.1
		5	20	3	33	0	0	*	3	3	*	1	1	26	+	+	1712.0
		10	27	4	57	0	0	*	2	2	*	1	1	29	–	–	4442.6
		100	–	20	490	1	1	1	0	0	0	1	1	1	+	+	+
	2	2	8	2	13	0	0	*	8	8	*	1	1	9	+	+	13.5
		5	20	3	26	0	0	*	4	4	*	1	1	21	+	+	720.8
		10	24	4	41	0	0	*	3	3	*	1	1	30	+	+	3562.6
		100	–	20	421	1	1	1	0	0	0	1	1	1	+	+	+
	3	2	10	2	13	0	0	*	8	8	*	1	1	11	+	+	121.1
		5	18	3	36	0	0	*	3	3	*	1	1	19	+	+	1811.0
		10	–	4	52	0	0	1	2	2	2	1	1	1	+	+	+
		100	–	20	691	1	1	1	0	0	0	1	1	1	+	+	+
	4	2	10	2	17	*	*	*	*	*	*	13	13	12	5873.0	5534.4	118.0
		5	24	3	37	0	0	*	3	3	*	1	1	25	+	+	4743.2
		10	26	4	47	0	0	*	2	2	*	1	1	31	+	+	4433.5
		100	–	20	497	1	1	1	0	0	0	1	1	1	+	+	+
	5	2	9	2	15	*	*	*	*	*	*	12	12	11	4611.9	3854.2	24.2
		5	22	3	34	0	0	*	3	3	*	1	1	25	+	+	626.5
		10	26	4	59	0	0	*	2	2	*	1	1	31	+	+	2403.6
		100	–	20	507	1	1	1	0	0	0	1	1	1	+	+	+

5.3.3 Algorithm LL

This subsection describes the computational results for the algorithm LL, which uses the layered graph formulation and explores the objective space in a low-to-high order. Tables 5.12 to 5.16 contain all data from our test runs, again grouped like for the other algorithms.

As for the other algorithms, all instances from R10 are quickly solved to proven optimality by each approach (see Table 5.12). Since the algorithm explores the interval $[d^{min}, d^{max}]$ in low-to-high delay bound order, it must solve the RDCSTP for each integer value in the interval (with the exception of d^{max}, which is covered by the initial solution for the STP). Thus, to find the complete Pareto frontier, algorithm LL always requires exactly $d^{max} - d^{min} + 1$ iterations, which is exactly what happens for these instances. We observe no significant difference in runtime performance between the approaches, probably due to the relative ease with which the instances can be solved. In contrast, the instances' d_e^{max} values do seem to noticeably affect the time required for finding their respective Pareto frontiers. This is likely related to the fact that, as established earlier, d_e^{max} correlates with d^{max}, which in turn affects both the number of iterations that must be solved and the size of the resulting ILPs that are used for solving the RDCSTP instances, especially those towards the end of the loop (which are those with high delay bound, as we are going from d^{min} to d^{max}).

Table 5.13 shows that the algorithm is still able to solve every instance from R20 to proven optimality with any of the approaches. While LLB and LLP show very similar runtime performance for most instances (with significant differences between the two only emerging for the instances with $d_e^{max} = 50$, where they take turns outperforming each other), LLR is able to solve most instances faster than any of the other two approaches, especially those with high d_e^{max} (with one notable exception where it requires the longest time among all approaches). Again, a high d_e^{max} value seems to correlate with high difficulty, as all approaches show noticeably worse performance for instances where that value is large.

Our findings for instances from R50 are analogous to those from the previous paragraph, with the main difference being that the algorithm is no longer able to solve every instance to proven optimality. Nine instances are not solved by any of the approaches (with the fraction of the explored delay interval ranging from just barely over 10% to more than 70% and being relatively similar between the three approaches for most instances), while an additional two are only solved by LLR. For those instances where all approaches find the complete Pareto frontier, LLB and LLP again show very similar performance, which is consistent with our previous hypothesis stated in Section 5.3.2 that preprocessing does not seem to significantly affect the algorithm's runtime when used in conjunction with the layered graph formulation. We again observe that instances with high d_e^{max} value are noticeable harder to solve than those with a low one, likely due to its aforementioned effects on the size of the ILPs that need to be solved during the individual iterations of the ε-constraint method.

The results for the instances from the sets T20, which are given in Tables 5.14 and 5.15, are again consistent with our previous observations. The effects of a high d_e^{max} value on the runtime can be seen even more clearly, since instances with d_e^{max} up to 100 exist in these sets. As for the previous instance sets, LLB and LLP show very similar performance, while LLR is usually faster than both of them. Of the aforementioned difficult instances with $d_e^{max} = 100$, only four can be solved by every approach, while LLR is able to solve an additional four. In accordance

with our findings for algorithm LH, we observe no significant difference in runtime performance between the Euclidean and random instances, which further indicates that instance parameters like the size of its relevant delay interval $[d^{min}, d^{max}]$ and its number of vertices have greater influence over the required runtime than the underlying graph's structure.

The findings for instance sets T20 are again corroborated by the results for TR40, which are shown in Table 5.15. All approaches solve every instance with $d_e^{max} \leq 10$ to proven optimality, with LLR being the fastest approach. Instances again become increasingly difficult to solve with higher d_e^{max} and those with $d_e^{max} = 100$ cannot be solved by any of our approaches. Interestingly, the algorithm usually runs out of memory before reaching the time limit, whereas for previous instances, it mostly failed due to the latter.

Similar to our results for algorithm LH, Table 5.16 shows that the instances from set TE40, where the edge costs are Euclidean and the root vertex is on a corner of the other vertices, are harder to solve with the layered graph formulation than random ones once a certain instance size is reached. We assume that this is because of these instances' relatively large d^{max} values, which are likely due to the increased distance between the root and the other vertices.

The relative performance of the three approaches is again consistent with our previous findings, as is our observation that instances with larger d_e^{max} are, in general, more difficult to solve.

In conclusion, the reuse approach LLR again seems to be the best choice among the three in terms of its ability to quickly identify the complete Pareto frontier. All other findings, specifically concerning the significant impact that d_e^{max} seems to have on the algorithm's performance, are consistent with those for algorithm LH (see Section 5.3.2).

Table 5.12: Computational results for algorithm LL for instance set R10

						#Solutions			% $[d^{min}, d^{max}]$			#Iterations			time [s]		
Set	i	d_e^{max}	#Sol	d^{min}	d^{max}	LLB	LLP	LLR	LLB	LLP	LLR	LLB	LLP	LLR	LLB	LLP	LLR
R10	1	5	7	3	9	*	*	*	*	*	*	7	7	7	0.0	0.0	0.0
		10	5	4	9	*	*	*	*	*	*	6	6	6	0.0	0.0	0.0
		20	11	8	58	*	*	*	*	*	*	51	51	51	4.2	4.0	4.0
		50	8	21	81	*	*	*	*	*	*	61	61	61	1.5	1.5	1.4
	2	5	1	1	1	*	*	*	*	*	*	1	1	1	0.0	0.0	0.0
		10	2	6	8	*	*	*	*	*	*	3	3	3	0.0	0.0	0.0
		20	2	8	14	*	*	*	*	*	*	7	7	7	0.0	0.0	0.0
		50	4	19	49	*	*	*	*	*	*	31	31	31	0.2	0.2	0.2
	3	5	6	4	16	*	*	*	*	*	*	13	13	13	0.2	0.2	0.1
		10	7	6	24	*	*	*	*	*	*	19	19	19	0.3	0.2	0.2
		20	7	5	34	*	*	*	*	*	*	30	30	30	0.4	0.4	0.4
		50	5	9	71	*	*	*	*	*	*	63	63	63	4.4	4.5	4.2
	4	5	3	2	4	*	*	*	*	*	*	3	3	3	0.0	0.0	0.0
		10	5	5	9	*	*	*	*	*	*	5	5	5	0.0	0.0	0.0
		20	3	7	16	*	*	*	*	*	*	10	10	10	0.0	0.0	0.0
		50	5	22	50	*	*	*	*	*	*	29	29	29	0.2	0.2	0.2
	5	5	4	3	13	*	*	*	*	*	*	11	11	11	0.1	0.1	0.1
		10	8	5	20	*	*	*	*	*	*	16	16	16	0.4	0.4	0.3
		20	7	11	52	*	*	*	*	*	*	42	42	42	1.4	1.4	1.4
		50	10	13	117	*	*	*	*	*	*	105	105	105	25.2	25.3	24.7

Table 5.13: Computational results for algorithm LL for instance sets R20 and R50

						#Solutions			% $[d^{min}, d^{max}]$			#Iterations			time [s]		
Set	i	d_e^{max}	#Sol	d^{min}	d^{max}	LLB	LLP	LLR	LLB	LLP	LLR	LLB	LLP	LLR	LLB	LLP	LLR
R20	1	5	11	4	18	*	*	*	*	*	*	15	15	15	1.7	1.7	1.4
		10	12	6	27	*	*	*	*	*	*	22	22	22	17.6	17.0	8.1
		20	16	4	50	*	*	*	*	*	*	47	47	47	20.6	20.6	17.3
		50	23	13	99	*	*	*	*	*	*	87	87	87	153.1	185.2	148.2
	2	5	12	3	18	*	*	*	*	*	*	16	16	16	1.6	1.7	1.4
		10	15	4	43	*	*	*	*	*	*	40	40	40	67.6	67.3	48.8
		20	14	6	75	*	*	*	*	*	*	70	70	70	77.2	77.3	54.5
		50	21	14	188	*	*	*	*	*	*	175	175	175	1961.8	1617.3	1054.7
	3	5	11	3	15	*	*	*	*	*	*	13	13	13	1.8	1.7	1.3
		10	18	5	41	*	*	*	*	*	*	37	37	37	25.1	25.3	10.8
		20	24	9	73	*	*	*	*	*	*	65	65	65	166.4	166.2	67.6
		50	28	17	123	*	*	*	*	*	*	107	107	107	296.5	331.1	208.8
	4	5	9	3	13	*	*	*	*	*	*	11	11	11	0.5	0.5	0.5
		10	19	5	31	*	*	*	*	*	*	27	27	27	5.1	4.9	4.3
		20	20	8	54	*	*	*	*	*	*	47	47	47	24.9	25.7	22.0
		50	26	21	126	*	*	*	*	*	*	106	106	106	346.8	390.3	454.6
	5	5	8	2	16	*	*	*	*	*	*	15	15	15	1.0	1.1	1.0
		10	18	3	39	*	*	*	*	*	*	37	37	37	9.8	10.1	8.9
		20	13	9	49	*	*	*	*	*	*	41	41	41	12.2	12.3	11.0
		50	17	20	183	*	*	*	*	*	*	164	164	164	416.1	493.8	335.9
R50	1	5	18	2	38	14	13	16	65	54	73	25	24	28	–	–	–
		10	33	4	85	21	23	26	29	30	39	25	26	37	–	–	–
		20	–	4	173	27	27	30	24	24	28	41	43	51	–	–	–
		50	–	8	364	32	32	37	11	11	13	41	41	49	–	–	–
	2	5	14	3	20	*	*	*	*	*	*	18	18	18	57.3	56.0	32.1
		10	20	3	33	*	*	*	*	*	*	31	31	31	267.8	255.4	138.0
		20	35	6	73	*	*	*	*	*	*	68	68	68	5543.8	4880.9	2169.7
		50	–	12	270	48	49	59	31	31	44	81	82	120	–	–	–
	3	5	14	3	24	*	*	*	*	*	*	22	22	22	110.5	130.7	55.7
		10	19	3	40	*	*	*	*	*	*	38	38	38	714.5	650.2	331.9
		20	35	7	104	33	33	33	69	69	69	70	70	93	–	–	–
		50	46	10	207	43	43	44	47	47	55	94	96	117	–	–	–
	4	5	22	3	46	*	*	*	*	*	*	44	44	44	8654.3	9300.5	2004.6
		10	22	3	49	*	*	*	*	*	*	47	47	47	1457.0	1498.5	524.3
		20	38	4	130	30	30	33	46	46	53	59	59	74	–	–	–
		50	51	10	341	42	43	45	35	36	42	119	125	142	–	–	–
	5	5	15	3	24	*	*	*	*	*	*	22	22	22	148.2	132.9	60.9
		10	20	3	53	19	19	*	90	90	*	49	48	51	–	–	4570.4
		20	39	5	85	38	38	*	88	69	*	72	71	81	–	–	8507.5
		50	57	8	234	48	48	52	37	37	56	95	93	128	–	–	–

Table 5.14: Computational results for algorithm LL for instance sets TR20 and TC20

						#Solutions			% $[d^{min}, d^{max}]$			#Iterations			time [s]		
Set	i	d_e^{max}	#Sol	d^{min}	d^{max}	LLB	LLP	LLR	LLB	LLP	LLR	LLB	LLP	LLR	LLB	LLP	LLR
TR20	1	2	3	1	4	*	*	*	*	*	*	4	4	4	0.1	0.0	0.1
		5	7	3	9	*	*	*	*	*	*	7	7	7	0.2	0.2	0.2
		10	13	4	27	*	*	*	*	*	*	24	24	24	5.7	5.5	6.0
		100	36	38	240	*	*	*	*	*	*	203	203	203	2579.1	2875.0	2213.8
	2	2	10	2	14	*	*	*	*	*	*	13	13	13	1.4	1.4	1.0
		5	12	3	21	*	*	*	*	*	*	19	19	19	5.2	4.8	3.7
		10	13	4	36	*	*	*	*	*	*	33	33	33	17.7	20.1	12.6
		100	28	23	436	24	24	27	41	41	72	229	222	354	–	–	–
	3	2	7	2	8	*	*	*	*	*	*	7	7	7	0.3	0.3	0.3
		5	9	3	12	*	*	*	*	*	*	10	10	10	0.6	0.6	0.5
		10	12	4	26	*	*	*	*	*	*	23	23	23	6.8	6.1	5.0
		100	17	23	216	*	*	*	*	*	*	194	194	194	2683.1	2386.0	1938.3
	4	2	9	2	14	*	*	*	*	*	*	13	13	13	3.2	3.2	2.5
		5	14	3	25	*	*	*	*	*	*	23	23	23	25.5	25.5	9.1
		10	19	4	48	*	*	*	*	*	*	45	45	45	142.2	154.4	45.2
		100	30	23	450	28	26	29	67	60	70	291	258	317	–	–	–
	5	2	9	2	13	*	*	*	*	*	*	12	12	12	2.7	2.9	1.7
		5	13	3	21	*	*	*	*	*	*	19	19	19	5.3	5.0	3.8
		10	20	4	46	*	*	*	*	*	*	43	43	43	66.0	65.2	36.2
		100	31	23	375	29	30	30	55	76	76	258	278	318	–	–	–
TC20	1	2	4	1	4	*	*	*	*	*	*	4	4	4	0.0	0.0	0.0
		5	7	3	10	*	*	*	*	*	*	8	8	8	0.3	0.3	0.3
		10	9	4	24	*	*	*	*	*	*	21	21	21	5.8	5.8	5.0
		100	22	38	192	*	*	*	*	*	*	155	155	155	506.8	799.4	534.8
	2	2	8	2	13	*	*	*	*	*	*	12	12	12	6.0	6.0	2.6
		5	13	3	23	*	*	*	*	*	*	21	21	21	38.9	39.0	17.3
		10	15	4	33	*	*	*	*	*	*	30	30	30	25.8	29.7	16.1
		100	20	23	353	17	17	18	55	55	62	196	196	236	–	–	–
	3	2	4	2	8	*	*	*	*	*	*	7	7	7	1.5	1.6	1.4
		5	6	3	15	*	*	*	*	*	*	13	13	13	11.9	12.8	10.3
		10	10	4	30	*	*	*	*	*	*	27	27	27	455.1	490.2	559.2
		100	19	23	188	*	*	*	89	89	*	158	158	166	–	–	7244.9
	4	2	4	2	5	*	*	*	*	*	*	4	4	4	0.0	0.0	0.0
		5	6	3	12	*	*	*	*	*	*	10	10	10	0.7	0.7	0.6
		10	11	4	27	*	*	*	*	*	*	24	24	24	47.0	46.8	31.3
		100	13	23	167	*	*	*	*	*	*	145	145	145	1252.5	1187.6	942.6
	5	2	4	2	5	*	*	*	*	*	*	4	4	4	0.1	0.0	0.0
		5	8	3	14	*	*	*	*	*	*	12	12	12	3.7	4.2	3.1
		10	9	4	25	*	*	*	*	*	*	22	22	22	43.6	43.0	34.7
		100	17	23	195	*	*	*	82	82	*	168	166	173	–	–	7806.6

Table 5.15: Computational results for algorithm LL for instance sets TE20 and TR40

						#Solutions			% $[d^{min}, d^{max}]$			#Iterations			time [s]		
Set	i	d_e^{max}	#Sol	d^{min}	d^{max}	LLB	LLP	LLR	LLB	LLP	LLR	LLB	LLP	LLR	LLB	LLP	LLR
TE20	1	2	4	1	5	*	*	*	*	*	*	5	5	5	0.8	0.8	0.6
		5	6	3	11	*	*	*	*	*	*	9	9	9	9.2	9.2	3.8
		10	11	4	28	*	*	*	*	*	*	25	25	25	324.3	345.2	177.2
		100	35	38	303	34	34	34	59	59	59	173	174	184	–	–	–
	2	2	8	2	13	*	*	*	*	*	*	12	12	12	6.0	6.0	2.6
		5	13	3	23	*	*	*	*	*	*	21	21	21	40.2	40.1	15.9
		10	15	4	33	*	*	*	*	*	*	30	30	30	27.6	29.3	16.4
		100	20	23	353	17	17	18	55	55	62	201	196	235	–	–	–
	3	2	5	2	8	*	*	*	*	*	*	7	7	7	1.2	1.3	1.0
		5	7	3	14	*	*	*	*	*	*	12	12	12	15.7	16.1	9.4
		10	9	4	24	*	*	*	*	*	*	21	21	21	115.7	111.8	75.1
		100	18	23	274	15	15	15	53	53	53	150	150	158	–	–	–
	4	2	5	2	10	*	*	*	*	*	*	9	9	9	8.8	8.9	2.8
		5	8	3	23	*	*	*	*	*	*	21	21	21	132.1	127.3	37.7
		10	10	4	38	*	*	*	*	*	*	35	35	35	465.9	472.9	187.7
		100	10	23	258	*	*	*	44	44	*	208	207	236	–	–	9635.1
	5	2	5	2	6	*	*	*	*	*	*	5	5	5	0.3	0.2	0.2
		5	9	3	16	*	*	*	*	*	*	14	14	14	5.7	6.5	3.7
		10	11	4	26	*	*	*	*	*	*	23	23	23	23.0	23.4	8.7
		100	16	23	256	14	14	*	64	71	*	219	228	234	–	–	7656.5
TR40	1	2	9	2	11	*	*	*	*	*	*	10	10	10	6.0	6.0	3.4
		5	16	3	26	*	*	*	*	*	*	24	24	24	235.3	236.7	118.7
		10	19	4	49	*	*	*	*	*	*	46	46	46	2722.4	2537.5	492.2
		100	–	20	541	1	1	1	0	0	0	1	1	1	+	+	+
	2	2	10	2	17	*	*	*	*	*	*	16	16	16	101.5	98.5	64.8
		5	16	3	29	*	*	*	*	*	*	27	27	27	510.6	477.3	264.1
		10	21	4	49	*	*	*	*	*	*	46	46	46	2560.9	2688.0	1152.4
		100	35	20	377	1	1	1	1	1	1	1	1	1	+	+	+
	3	2	9	2	13	*	*	*	*	*	*	12	12	12	20.1	21.6	7.9
		5	15	3	27	*	*	*	*	*	*	25	25	25	212.0	229.4	116.7
		10	28	4	45	*	*	*	*	*	*	42	42	42	3215.1	3889.4	806.8
		100	–	20	495	1	1	1	0	0	0	1	1	1	+	+	+
	4	2	10	2	11	*	*	*	*	*	*	10	10	10	5.1	5.2	3.4
		5	13	3	23	*	*	*	*	*	*	21	21	21	103.0	100.3	44.5
		10	24	4	48	*	*	*	*	*	*	45	45	45	1128.8	1156.9	321.1
		100	–	20	338	1	1	1	1	1	1	1	1	1	+	+	+
	5	2	7	2	10	*	*	*	*	*	*	9	9	9	2.0	2.1	1.5
		5	13	3	22	*	*	*	*	*	*	20	20	20	42.7	42.2	23.2
		10	23	4	34	*	*	*	*	*	*	31	31	31	90.7	86.3	52.4
		100	–	20	364	1	1	1	1	1	1	1	1	1	+	+	+

Table 5.16: Computational results for algorithm LL for instance sets TC40 and TE40

						#Solutions			% $[d^{min}, d^{max}]$			#Iterations			time [s]		
Set	i	d_e^{max}	#Sol	d^{min}	d^{max}	LLB	LLP	LLR	LLB	LLP	LLR	LLB	LLP	LLR	LLB	LLP	LLR
TC40	1	2	10	2	14	*	*	*	*	*	*	13	13	13	2654.6	2356.5	1905.7
		5	21	3	36	15	15	15	53	53	53	19	19	20	–	–	–
		10	21	4	54	15	15	15	41	41	31	25	25	25	–	–	–
		100	65	20	515	1	1	1	0	0	0	1	1	1	+	+	+
	2	2	10	2	15	9	9	9	71	71	64	12	12	12	–	–	–
		5	15	3	31	14	14	14	52	52	52	16	16	17	–	–	–
		10	19	4	51	14	14	14	33	33	33	20	20	21	–	–	–
		100	53	20	501	1	1	1	0	0	0	1	1	1	+	+	+
	3	2	7	2	11	*	*	*	*	*	*	10	10	10	91.0	92.4	52.5
		5	15	3	25	*	*	*	*	*	*	23	23	23	4131.8	3999.7	2348.1
		10	18	4	42	1	1	*	5	5	*	1	1	39	+	+	9171.6
		100	48	20	299	31	31	33	29	29	33	81	83	98	–	–	–
	4	2	7	2	11	*	*	*	*	*	*	10	10	10	449.0	444.4	370.6
		5	12	3	20	*	*	*	*	*	*	18	18	18	2696.8	2745.5	2912.3
		10	19	4	45	1	1	1	5	5	5	1	1	1	+	+	+
		100	53	20	377	1	1	1	1	1	1	1	1	1	+	+	+
	5	2	7	2	9	*	*	*	*	*	*	8	8	8	47.7	52.6	40.0
		5	12	3	20	*	*	*	*	*	*	18	18	18	1241.7	1241.7	1078.5
		10	17	4	29	*	*	*	*	*	*	26	26	26	2516.6	2663.9	2358.3
		100	44	20	295	1	1	1	1	1	1	1	1	1	+	+	+
TE40	1	2	12	2	15	11	11	11	79	86	79	12	13	13	–	–	–
		5	20	3	33	13	13	13	48	48	45	16	17	17	–	–	–
		10	27	4	57	16	16	16	30	30	30	18	18	19	–	–	–
		100	–	20	490	1	1	1	0	0	0	1	1	1	+	+	+
	2	2	8	2	13	*	*	*	67	67	67	10	10	10	–	–	+
		5	20	3	26	12	12	12	50	50	46	13	13	12	–	–	–
		10	24	4	41	15	15	15	39	39	39	17	17	17	–	–	–
		100	–	20	421	1	1	1	0	0	0	1	1	1	+	+	+
	3	2	10	2	13	9	9	8	67	67	67	9	9	9	–	–	–
		5	18	3	36	12	12	12	38	35	38	14	13	14	–	–	–
		10	–	4	52	1	1	1	4	4	4	1	1	1	+	+	+
		100	–	20	691	1	1	1	0	0	0	1	1	1	+	+	+
	4	2	10	2	17	*	*	*	*	*	*	16	16	16	9803.3	9847.0	7260.5
		5	24	3	37	15	15	15	43	43	43	16	16	16	–	–	–
		10	26	4	47	1	1	1	5	5	5	1	1	1	+	+	+
		100	–	20	497	1	1	1	0	0	0	1	1	1	+	+	+
	5	2	9	2	15	*	*	*	*	*	*	14	14	14	3210.1	3424.2	2238.0
		5	22	3	34	16	16	17	50	47	50	17	16	17	+	–	–
		10	26	4	59	1	1	1	4	4	4	1	1	1	+	+	+
		100	–	20	507	1	1	1	0	0	0	1	1	1	+	+	+

5.4 Comparing the different models' performance

As the results given in the previous section have shown, all algorithms benefit from preprocessing and the reuse of information from previous iterations. The reuse approaches always outperform their simpler counterparts, both in terms of runtime and the size of the relevant objective space interval that can be explored (which sometimes results in that approach still being able to solve an instance to proven optimality, whereas the others fail to do so).

This section now compares the reuse approaches of the different algorithms presented in the previous section: the pathcut algorithm PCR, the layered graph algorithm working in high-to-low order (LHR) and the layered graph algorithm working in low-to-high order (LLR).

For each set of instances and each corresponding d_e^{max}, we aggregated the results of all five corresponding instances, which can be found in Table 5.17. The table contains the following columns for each algorithm:

- The number of instances for which the complete Pareto frontier was found. If all instances were solved to proven optimality, this number is printed **bold**.
- The average fraction of $[d^{min}, d^{max}]$ that the algorithm explored. If this average is 100% (i.e., the algorithm solved all instances to proven optimality), the column contains an asterisk (*) instead.
- The average runtime of the algorithm. Note that instances where the algorithm ran out of memory were counted as 10000 seconds.

First, we observe that all instances in sets R10 and R20 can be solved to proven optimality by each algorithm. For those in R10, the algorithms each find the complete Pareto frontier very quickly, with only LLR requiring more than one second on average to solve the instances with $d_e^{max} = 50$.

With the instances from set R20, the differences between the algorithms start to become more noticeable. LLR, while still being able to find the complete Pareto frontier for all instances, now requires substantially more time to do so than the other two algorithms, especially for instances with high d_e^{max}.

Algorithms PCR and LHR, on the other hand, show a very similar performance for all of R20's instance subsets. LHR is faster at finding the Pareto frontier for instances with $d_e^{max} \leq 20$, whereas PCR overtakes it in terms of runtime performance for those with $d_e^{max} = 50$.

To analyze these apparent differences in runtime performance, we first make a note of how different parameters may influence the difficulty of solving a specific instance with a particular algorithm.

First, we note that the branch-and-cut algorithm based on the layered graph formulation for solving the RDCSTP is clearly dependent on the delay bound B, since it determines the that graph's number of layers and therefore its size and the size of the corresponding ILP (which, since we are dealing with with complete graphs, is in $O(B \cdot |V|^2)$, but is usually smaller due to preprocessing; see Section 2.4) The maximum value for B that we need to consider is determined by d^{max}, since we need to solve the RDCSTP for $B = d^{max} - 1$ if we want to find the complete Pareto frontier. Finally, d^{max} is strongly correlated with d_e^{max}, since the former is defined as the

delay of a path P_t (specifically, the delay of the highest-delay root-terminal path in the optimal solution of the STP), whereas the latter determines the average delay of paths within the graph. Thus, we can expect the delay bounds of the RDCSTP instances to be higher in general for those instances with high d_e^{max}.

While this likely makes solving the ILPs based on the layered graph formulation more difficult due to their increased size (which is usually one of the major factors in determining the difficulty of solving an ILP), the pathcut formulation is not adversely affected by an increased delay bound. In fact, since a high delay bound leads to fewer pathcut inequalities being added to the formulation, the algorithm may be able to find an optimal solution even more quickly than for lower delay bounds if the remaining formulation is sufficiently strong to ensure a quick convergence towards an optimal solution.

Together, these observations provide an explanation for the observed behaviour, that PCR is slower than LHR for instances with small d_e^{max}, but faster for those with high d_e^{max}.

For an explanation as to why algorithm LLR seems to perform significantly worse than its counterpart LHR (which simply explores the objective space in the opposite direction), we again look to the parameter d_e^{max} – specifically, its effect on the size of the relevant delay interval $[d^{min}, d^{max}]$. This interval usually becomes larger as d_e^{max} increases, since d^{min} does not grow as fast as d^{max}. Since LLR needs to solve the RDCSTP for each integer value within that interval, this leads to a high number of iterations that are required for finding the complete Pareto frontier. Together with the aforementioned increasing difficulty of solving these subproblems (especially those towards the high end of the interval), this explains the observed performance. It also serves as an explanation as to why the pathcut algorithm need slightly more time to solve instances with higher d_e^{max}, since it too must explore a larger delay interval (as must every algorithm using the ε-constraint method).

The results for instance set R50 corroborate our previous findings on the effects of increasing d_e^{min} on the performance of the algorithms in that all three perform significantly worse for those instances with high d_e^{min}. Notably, this instance set is where the algorithms start to fail at solving all instances within the set. While LHR still manages to solve all but three instances, LLR can only solve ten and PCR fails to solve more than three. These results seem to suggest that the size of the input instance is a significant factor in determining the difficulty of solving it, which is to be expected for combinatorial optimization problems. Also note that while the two algorithms based on the layered graph formulation consistently become worse as d_e^{max} increases, the pathcut algorithm again becomes slightly better for those instances with $d_e^{max} = 50$, which supports our previous interpretation.

The generally bad performance of PCR on these instances can be explained by the fact that the number of potential root-terminal paths increases significantly as the instance size grows (especially since we are dealing with complete graphs). Thus, our branch-and-cut approach of identifying such paths that exceed the delay bound is no longer able to sufficiently constrain the search space, which results in the algorithm being unable to solve most instances to proven optimality. The layered graph formulations are not impacted as much.

The runtimes for the instances from set TR20 are consistent with our previous observations. All algorithms solve instances with low d_e^{max} rather fast, but require more time for solving instances as that value increases. Again, PCR is noticeably less affected by an increase in d_e^{max},

likely for the aforementioned reasons.

For the sets TC20 and TE20, which contain the first instances with Euclidean edge costs, we observe similar runtimes to those for instance set TR20. However, for instances with high d_e^{max}, the pathcut algorithm shows an even more significant improvement over the other algorithms, being more than one hundred times faster than LHR for instance set TC20 and still around fifty times faster for set TE20. As argued in the previous section, we assume that this is due to the fact that since the edge costs are Euclidean, only relatively few root-terminal paths with low cost exist and therefore, the algorithm quickly converges to an optimal solution.

Like the ones for instance set TR20, our results for set TR40 are consistent with our previous findings. The layered graph algorithms both show excellent performance on instances with small d_e^{max}, but deteriorate quickly as the value rises, in the end failing to solve most instances with $d_e^{max} = 100$. In contrast, the pathcut algorithm already has significant difficulty with solving even those instances with small d_e^{max}, likely due to the instances already large size. However, its performance deteriorates much more slowly than that of the other algorithms.

For the instances in sets TC40 and TE40, LHR again works rather well for those with low d_e^{max}, but is unable to find the Pareto frontier for those with $d_e^{max} = 100$. The runtimes of LLR follow the same trend, but are always noticeably worse than those of LHR.

The pathcut algorithm, on the other hand, is able to solve every instance in TC40 to proven optimality, likely due to leveraging the graph's structure. It is, however, again outperformed by LHR for instances with small d_e^{max}, which is consistent with our previous observations. In contrast to its good performance on instance set TE20, PCR can only solve one of the instances from TE40 to proven optimality. We assume that the instances' large sizes already outweigh any factors that might have helped PCR with solving the smaller ones in TE20.

In conclusion, algorithms PCR and LHR both show distinct strengths and weaknesses which must be considered before selecting one of them as the algorithm of choice.

The algorithm based on the layered graph formulation, LHR, shows the best overall performance, being able to solve the highest number of instances to proven optimality out of all three algorithms. Additionally, when one or both of the other algorithms also manage to solve all instances of a given subset, LHR is often the fastest to finds the complete Pareto frontier. Its performance does, however, degrade significantly with increasing d_e^{max}. As we elaborated, this is likely because the size of its ILP is strongly correlated with this value.

The pathcut-based algorithm PCR also shows excellent performance on most instances, often finding the Pareto frontier almost as quickly as algorithm LHR. Additionally, it does not suffer as severe a performance degradation as the layered graph algorithms with high d_e^{max} and is therefore able to solve these instances faster than the other two algorithms. This is especially true for instance sets TC20 and TE20, where it clearly outclasses them. However, its performance does not scale well with an increasing number of vertices. For instances with $|V| > 20$, it rarely comes close to matching the performance of LHR, except for those with high d_e^{max}.

Finally, algorithm LLR is clearly outperformed by its high-to-low counterpart LHR on nearly every instance. In the one case where it explores a slightly larger percentage of the delay interval than LHR, both are clearly outmatched by the pathcut algorithm. In general, the low-to-high approach did not show any clear advantages that make up for the higher number of ε-constraint iterations that it entails.

Table 5.17: Comparison of the different algorithms' computational performance

		#solved			avg. %			avg. time [s]		
Set	d_e^{max}	PCR	LHR	LLR	PCR	LHR	LLR	PCR	LHR	LLR
R10	5	**5**	**5**	**5**	*	*	*	0.0	0.0	0.0
	10	**5**	**5**	**5**	*	*	*	0.0	0.0	0.1
	20	**5**	**5**	**5**	*	*	*	0.0	0.1	1.2
	50	**5**	**5**	**5**	*	*	*	0.0	0.4	6.1
R20	5	**5**	**5**	**5**	*	*	*	1.1	0.6	1.1
	10	**5**	**5**	**5**	*	*	*	3.3	3.2	16.2
	20	**5**	**5**	**5**	*	*	*	16.4	7.4	34.5
	50	**5**	**5**	**5**	*	*	*	14.2	30.4	440.4
R50	5	2	**5**	4	89.2	*	94.6	8273.8	103.8	2430.7
	10	1	**5**	4	83.0	*	87.8	9289.4	1083.7	3112.9
	20	0	4	2	74.6	84.0	70.0	10000	3430.7	8135.4
	50	0	3	0	76.2	70.2	42.0	10000	7611.6	10000
TR20	2	**5**	**5**	**5**	*	*	*	1.0	0.6	1.1
	5	**5**	**5**	**5**	*	*	*	9.3	1.7	3.5
	10	**5**	**5**	**5**	*	*	*	6.8	5.2	21.0
	100	**5**	**5**	2	*	*	83.6	80.4	569.3	6830.4
TC20	2	**5**	**5**	**5**	*	*	*	0.3	0.3	0.8
	5	**5**	**5**	**5**	*	*	*	1.7	1.1	6.3
	10	**5**	**5**	**5**	*	*	*	0.6	3.0	129.3
	100	**5**	**5**	4	*	*	92.4	2.2	242.7	5305.8
TE20	2	**5**	**5**	**5**	*	*	*	8.1	0.5	1.4
	5	**5**	**5**	**5**	*	*	*	2.7	2.1	14.1
	10	**5**	**5**	**5**	*	*	*	1.4	4.8	93.0
	100	**5**	**5**	2	*	*	74.8	7.6	390.8	9458.3
TR40	2	4	**5**	**5**	95.0	*	*	3205.4	3.8	16.2
	5	3	**5**	**5**	92.0	*	*	4457.8	17.4	113.4
	10	3	**5**	**5**	89.8	*	*	5476.6	110.8	565.0
	100	1	1	0	20.0	20.0	0.6	8025.7	8722.8	10000
TC40	2	**5**	**5**	4	*	*	92.8	178.7	4.6	2473.8
	5	**5**	**5**	3	*	*	81.0	85.7	21.8	5267.8
	10	**5**	**5**	2	*	*	53.8	539.0	108.8	8306.0
	100	**5**	0	0	*	5.4	7.0	3294.0	10000	10000
TE40	2	0	**5**	2	72.0	*	82.6	10000	170.2	7899.7
	5	1	**5**	0	78.4	*	44.4	9708.9	1922.7	10000
	10	0	4	0	29.0	80.4	16.4	10000	4968.5	10000
	100	0	0	0	0.0	0.0	0.0	10000	10000	10000

CHAPTER 6

Solution approaches for the Multi-objective Steiner Tree Problem with Resources

In the previous chapters, we focused most of our attention on the BOSTPD. Because it only has two objective functions, it is both easier to understand and to solve, while still posing most of the challenges that arise when dealing with multi-objective optimization.

However, as we will show in this chapter, our solution approach can easily be generalized to the multi-objective case. Indeed, our original problem definition in Section 1.2 describes the general multi-objective problem, which we only then restricted to the bi-objective case (both for ease of understanding and improved runtime performance). In this chapter, we will now elaborate on how to solve the MOSTPR, using techniques adapted from those presented in Chapter 4.

Again, we will mostly restrict ourselves to a low-dimensional variant of the MOSTPR: the Tri-objective Steiner Tree Problem with (two) Resources. By showing how the problem can be generalized from a bi-objective to a tri-objective one, we aim to give an intuitive understanding of how it can be generalized to arbitrarily many objectives.

6.1 Multi-objective ε-constraint method

First, observe that the differently dimensioned Multi-objective Steiner Tree Problems differ only in their objective functions. Thus, the feasible region of the decision space that is described by the problems' sets of constraints is always the same (basically, requiring that the selected edges form a tree that connects all terminal vertices). The number of objective functions determines the dimension of the objective space, which in turn determines how we must explore it to find the Pareto frontier when using the ε-constraint method.

Finding the complete Pareto frontier in a two-dimensional objective space, as we have in the case of the BOSTPD, is relatively easy: We find an optimal solution w.r.t. the first objective func-

tion and iteratively move towards an optimal solution w.r.t. the second objective function from there, always making sure that the intermediate solutions are optimized as well. Intuitively, we consider every reasonable point within the Pareto-efficient interval of one objective function and find its corresponding optimal objective value w.r.t. the other objective function (cf. Sections 2.3 and 4.1).

This enumerative approach, however, does not scale well with the number of objective functions. Assuming that we are dealing with a multi-objective optimization problem (MOOP) with k objective functions, we have to find an optimal solution for every point within the $(k-1)$-dimensional subspace defined by the other objective functions' Pareto-efficient intervals. Thus, the number of subproblems that we have to solve rises exponentially with the number of objective functions.

In principle, however, the general ε-constraint procedure remains the same for the multi-objective case as it does for the bi-objective one: the multi-objective problem is transformed into a series of single-objective problems, which are then solved individually (perhaps using information from previous iterations to help the solution process, cf. Section 4.4) to yield the Pareto frontier. The fact that the original MOOP has more than two objective functions does, however, change both the iterative ε-constraint approach and the structure of the resulting single-objective subproblems.

6.1.1 ε-constraint method for the MOSTPR

In the bi-objective case, finding the interval of delay values for which Pareto-efficient solutions may exist was comparatively easy. By solving the regular STP and the shortest path problem w.r.t. delay on the input graph, we were able to determine an upper and a lower bound on the delay values we had to consider during our ε-constraint solution algorithm. By iterating through all these values from either side (optionally skipping those values for which no efficient solution can exist when starting at the highest delay bound), we were able to compute the complete Pareto frontier (cf. Section 4.1).

Unfortunately, such a simple approach does not work once we have to deal with more than two objective functions. Specifically, solving the STP no longer provides us with an upper bound for all resource demand bounds. We will provide a quick sketch on why this approach no longer works in general.

Let S^* be an optimal solution of the STP on our input graph, with c_{S^*}, $d^1_{S^*}$ and $d^2_{S^*}$ being its objective values. Clearly, solutions S^1 and S^2 with objective values $c_{S^1} = c_{S^2} = c_{S^*} + 1$, $d^1_{S^1} = d^1_{S^*} + 1$, $d^2_{S^1} = d^2_{S^*} - 1$, $d^1_{S^2} = d^1_{S^*} - 1$ and $d^2_{S^2} = d^2_{S^*} + 1$ are not dominated by S^* and therefore potentially Pareto-efficient. Thus, limiting our search to solutions with $d^1 \leq d^1_{S^*}$ and $d^2 \leq d^2_{S^*}$ might discard Pareto-efficient solutions and therefore prevent us from finding the complete Pareto frontier.

To find lower bounds on the resource demands that we need to consider in our algorithm, we can still use our aforementioned shortest path algorithms. However, while this allows us to find a lower bound for every single resource demand, starting our iterative procedure at the point corresponding to all these lower bounds does not necessarily yield a feasible solution. The reason for this that the shortest path tree w.r.t. the first resource's demand and the one w.r.t. the second resource's demand do not coincide in general. Thus, selecting the arcs corresponding

to the first tree precludes the selection of the second tree's arcs that are not already part of the solution without introducing cycles (which our problem definition forbids).

We may, of course, still use this point (the so-called *Utopia point*) as a starting point of our iterative procedure and solve the RDCSTP for all combinations of resource demand bounds larger than it, simply disregarding those where no feasible solution exists. However, since we do not yet have an upper bound on the delay bounds, we do not know when to stop our iterative procedure. To find these upper bounds, we must find the so-called *Nadir point*.

The Nadir point can be considered the opposite of the Utopia point. Whereas the latter is formed by the lower bounds of all objective functions, the former is formed by their upper bounds, i.e., its coordinates in the objective space correspond to the individual upper bounds of every objective function w.r.t. efficient solutions.

Unfortunately, finding the Nadir point is very difficult in practice [52], since we would need to already have information about set of Pareto-efficient solutions to determine the maximum reasonable value for each objective function. However, we can instead try to find an upper bound of every objective function w.r.t. feasible solutions, i.e., the maximum value that each objective can take in a feasible solution for our problem.

In our case, this upper bound corresponds to the length of the maximum longest path from the root vertex to every terminal, which presents two new challenges. Not only is finding the longest path in a graph an NP-hard problem, but the bound given by it is relatively weak, since it is unlikely that any optimal solution would actually contain any of these longest paths. This not only increases the size of the search space significantly, but also makes it harder for us to solve problem instances with the layered graph model, since its size grows quickly with increasingly high resource demand bounds.

While these challenges suggest that solving larger instances of the MOSTPR is unlikely to be possible within reasonable time, smaller instances might still be solvable with the following algorithm.

Since starting at the Nadir point and progressively decreasing the bounds allows us to skip parts of the search space, this approach is likely more suitable for solving the MOSTPR. We therefore start by setting the resource demand bounds B_1 and B_2 to the Nadir point's coordinates and solve the corresponding RDCSTP. Once we find a solution S with objective values d_S^1 and d_S^2, we add two new RDCSTP instances to our list of subproblems to be solved: one with bounds B_1 and $d_S^2 - 1$ and one with bounds $d_S^1 - 1$ and B_2. If a subproblem does not have a feasible solution, we know that we have reached an edge of the Pareto frontier. In this case, we simply take the next subproblem from our list and solve it. When the list is empty, we have found the complete Pareto frontier and stop our procedure. Depending on whether that list is implemented as a stack or as a queue, we explore the objective space in DFS or BFS order. After this algorithm terminates, we remove all non-efficient solutions from our set of candidates for Pareto-efficient solutions and return it as the algorithm's result.

6.2 Solving the resulting single-objective problems

As noted in the previous section, the structure of the single-objective subproblems changes to account for the fact that more than one objective function was transformed into constraints.

Specifically, in a feasible solution, every path between the root and a terminal vertex must obey one delay-constraint (or, more accurately, one resource-constraint) per transformed objective function.

Formally, let $G = (V, E, c, d^1, d^2, B_1, B_2)$ be an instance of the problem, where V is the set of vertices and E is the set of edges in the graph. The function $c : E \to \mathbb{N}$ assigns each edge a non-negative cost, whereas functions $d^1 : E \to \mathbb{N}$ and $d^2 : E \to \mathbb{N}$ assign them a demand for resource one and two, respectively. B_1 and B_2 are the demand bounds for these resources.

A feasible solution of the problem is a subgraph $G' = (V', E')$ that

1. is a tree,
2. connects all terminal vertices $t \in T$ and
3. obeys both delay bounds B_1 and B_2

Criterion three can be formally stated as follows: let P_t be the unique path from the root vertex r to a terminal vertex t in G'. A feasible solution must ensure that

$$\sum_{e \in P_t} d_e^i \leq B_i \qquad i \in \{1, 2\}$$

Our task is finding a feasible solution that minimizes the total edge cost

$$\sum_{e \in E'} c_e$$

Both solution approaches for the RDCSTP from Section 4.3 can be generalized to the tri-objective case.

6.2.1 Path-cut formulation

This ILP formulation works by first solving the regular STP for the instance and then searching the resulting preliminary solution for paths that exceed the delay bound B. If such a path is found, a constraint forbidding its use is added to the model and it is resolved until no more such paths are found.

The same approach can be used in the tri-objective case by extending the notion of an infeasible path. Every path that exceeds at least one of the bounds B_1 and B_2 is considered to be infeasible and therefore disallowed.

Finding such paths works exactly as in the bi-objective case, but independently for each of the transformed objective functions. For both functions, we calculate the cumulative delay from r to every terminal vertex, e.g., by using BFS. If we find a terminal whose root-path's delay exceeds any B_i, we add that path to our set of infeasible paths and forbid its further use by adding a corresponding constraint to the ILP model. Once no more such paths can be found, the solution is feasible.

6.2.2 Layered graph formulation

In the layered graph ILP model, the notion of feasible paths is encoded implicitly in the eponymous data structure. If a path is feasible, it has a corresponding variable assignment in the ILP model, whereas an infeasible one does not. Thus, since our definition of path feasibility changes with the number of objective functions, so must our data structure. Specifically, we need to introduce a second dimension of layers corresponding to the second resource demand bound that we now have to account for. While a brief explanation of how to generate multi-dimensional layered graphs is given at the end of Section 2.4, we will now provide a more elaborate explanation.

Formally, let $G = (V, A, c, d^1, d^2, B_1, B_2)$ be an instance of our problem. Similarly to the procedure described in Section 2.4, we construct a layered graph $\hat{G} = (\hat{V}, \hat{A}, c_L)$ as follows:

First, let $\hat{V} = r_L \cup \{v^{l,m} | v \in V \setminus \{r\}, 1 \leq l \leq B_1, 1 \leq m \leq B_2\}$, i.e., let the layered graph's set of vertices consist of a single copy of the root vertex r and $B_1 \cdot B_2$ copies of every other vertex in V.

Next, we add copies of the original graph's arcs to the layered graph. To this end, let $\hat{A} = \{(u^{l,m}, v^{l+d_e^1, m+d_e^2}) | e = (u, v) \in A, 0 \leq l \leq B_1 - d_e^1, 0 \leq m \leq B_2 - d_e^2\}$. Informally, this means that if an arc exists between the two vertices u and v in the original graph, arcs between the layered copies of u and v exist on the layered graph. Which specific copies are connected depends on the arc's resource demands: from every copy $u^{l,m}$ of the source vertex, we add an arc to the copy $v^{l+d_e^1, m+d_e^2}$ of the target vertex, as long as that target vertex copy exists (i.e., its layer indices are within the bounds B_1 and B_2). The variables d_e^1 and d_e^2 refer to the arc's demand for resource 1 and 2, respectively.

Intuitively, a vertex copy's layers encode how much of each resource was already spent on the path to that vertex. When moving along an arc in the layered graph, we always move towards the higher layer indices, which corresponds to the expenditure of resources according to the arc's demands.

Clearly, this approach can be generalized to an arbitrary number of resource demand constraints by simply adding additional dimensions to the layered graph. We add $B_1 \times \cdots \times B_k$ copies of every non-root vertex to the layered graph and connect them with arc copies analogously to the way we just described. Note, however, that the size of the layered graph grows exponentially with the number of resources that we need to consider. Thus, this model likely becomes too difficult to be solved in practice relatively quickly.

CHAPTER 7

Conclusion

As we noted in Chapter 1 of this thesis, the problem of finding a network that is both cheap to construct (as modeled by its edges' costs) and cheap to use (as modeled by their resource demands) arises naturally in the context of network optimization.

To facilitate modeling and solving these problems, we introduced the Multi-objective Steiner Tree Problem with Resources (see Section 1.2), which captures this aforementioned notion of optimizing a network w.r.t. both total edge cost and maximum cumulative resource consumption along each path from the root to a terminal vertex.

To solve the Bi-objective Steiner Tree Problem with Delays, the bi-objective variant of the MOSTPR, we developed a series of algorithms (see Chapter 4) that are based on the principles of the ε-constraint method and integer linear programming.

The algorithms use the ε-constraint method to decompose a bi-objective problem instance into a series of instances of the single-objective Rooted Delay-constrained Steiner Tree Problem. The subproblem instances are then encoded as integer linear programs according to two ILP formulations (the pathcut and the layered graph formulation; see Section 4.3) and solved by branch-and-cut with the ILP solver CPLEX. Preprocessing was used to reduce the size of the individual instances (see Section 4.2).

To further improve the runtime performance of our algorithms, we developed a framework for the reuse of information from previous iterations of the ε-constraint method. This includes the last iteration's solution, as well as inequalities that were added to the previous ILPs by branch-and-cut. To enable the reuse of solutions that are infeasible for the next iteration, we developed a heuristic method for transforming such solutions into new, feasible ones (see Section 4.4).

We implemented the aforementioned algorithms in C++ and tested them on a series of benchmark instances. Our tests showed that

- preprocessing and information reuse have a significant positive effect on the algorithms' runtimes,

- when using the ε-constraint method for this problem, algorithms that explore the objective space in a high-to-low delay bound order show better performance than those that do so in a low-to-high order, and
- that the pathcut formulation works best on instances with Euclidean edge costs, whereas the layered graph formulation works best on those with random edge costs and shows the best overall performance.

Finally, we provided a theoretical description on how the algorithms for the bi-objective case can be adapted to solve the multi-objective problem. This includes a description of the ε-constraint method for multiple objectives, as well as descriptions of ILP formulations for solving the RDCSTP with multiple delay (or resource) constraints, which is the subproblem arising for the individual iterations. However, we also note the significant challenges resulting from the multi-objective generalization of our bi-objective algorithms. These include the problem of finding a suitable starting point for the ε-constraint method, as well as the likely bad performance of the algorithms due to the large number of resulting subproblem instances and their difficulty.

7.1 Future work

As our test results from Chapter 5 have shown, an algorithm's performance can vary quite significantly depending on an instance's structure of lack thereof. Thus, an more detailed analysis of each algorithm's performance on more different classes of instances would certainly be interesting and possibly provide us with additional insight into why specific formulations seem to work better on certain classes of instances. Such insights might even lead to new algorithms that combine the different formulations into a single new one or select the best formulation depending on the instance's structure. Note that such an analysis should also cover instances based on sparse graphs, which we did not consider in this thesis at all.

Our tests also showed that the reuse of information throughout the iterations of the ε-constraint method can significantly improve the performance of an algorithm. Further research into what other information could be reused, as well as how to better utilize the current information during subsequent iterations (like better heuristics for repairing solutions) is certainly warranted.

Also, while we focused on using exact methods like integer linear programming to solve the subproblem instances, heuristic methods could be used instead to speed up the process of finding a solution. While this would obviously sacrifice the proven optimality of these solutions, it would likely allow us to solve larger instances where our current algorithms are no longer able to find the complete Pareto frontier.

Finally, the development and implementation of a complete algorithm for solving the multi-objective problem is an interesting goal. The latter would allow us to model and solve even more complex real-world problems than with the current bi-objective algorithms.

Abbreviations

APSPP All-pair Shortest Path problem.

BOPCSTP Bi-objective Prize-collecting Steiner Tree Problem.

BOSTPD Bi-objective Steiner Tree Problem with Delays.

COP combinatorial optimization problem.

HCMSTP Hop-constrained Minimum Spanning Tree Problem.

HCSTP Hop-constrained Steiner Tree Problem.

ILP integer linear program.

ILP integer linear programming.

LP linear program.

LP linear programming.

MIP mixed integer linear program.

MOOP multi-objective optimization problem.

MOSTPR Multi-objective Steiner Tree Problem with Resources.

MSTP Minimum Spanning Tree Problem.

RDCMSTP Rooted Delay-constrained Minimum Spanning Tree Problem.

RDCSTP Rooted Delay-constrained Steiner Tree Problem.

SSSPP Single-source Shortest Path problem.

STP Steiner Tree Problem on Graphs.

TSP Traveling Salesman Problem.

Bibliography

[1] Push-relabel maximum flow algorithm CS2. `http://web.archive.org/web/20010604073359/http://www.intertrust.com/star/goldberg/soft.html`.

[2] Norbert Ascheuer, Matteo Fischetti, and Martin Grötschel. A polyhedral study of the asymmetric traveling salesman problem with time windows. *Networks*, 36(2):69–79, 2000.

[3] Norbert Ascheuer, Matteo Fischetti, and Martin Grötschel. Solving the asymmetric travelling salesman problem with time windows by branch-and-cut. *Mathematical Programming*, 90(3):475–506, 2001.

[4] Richard Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[5] Jean-François Bérubé, Michel Gendreau, and Jean-Yves Potvin. An exact ϵ-constraint method for bi-objective combinatorial optimization problems: Application to the Traveling Salesman Problem with Profits. *European Journal of Operational Research*, 194(1):39–50, 2009.

[6] Vira Chankong and Yacov Y. Haimes. *Multiobjective decision making: theory and methodology*. North-Holland, 1983.

[7] George B. Dantzig. *Linear programming and extensions*. Princeton University Press, 1998.

[8] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[9] Stuart E. Dreyfus and Robert A. Wagner. The Steiner problem in graphs. *Networks*, 1(3):195–207, 1971.

[10] Irina Dumitrescu and Natashia Boland. Improved preprocessing, labeling and scaling algorithms for the Weight-Constrained Shortest Path Problem. *Networks*, 42(3):135–153, 2003.

[11] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.

[12] Matthias Ehrgott and Margaret M. Wiecek. Mutiobjective programming. In *Multiple criteria Decision Analysis: State of the art surveys*, pages 667–708. Springer, 2005.

[13] Peter Elias, Amiel Feinstein, and Claude E. Shannon. A note on the maximum flow through a network. *Information Theory, IRE Transactions on*, 2(4):117–119, 1956.

[14] José Figueira, Salvatore Greco, and Matthias Ehrgott. *Multiple criteria decision analysis: state of the art surveys*, volume 78. Springer, 2005.

[15] Robert W. Floyd. Algorithm 97: Shortest Path. *Communications of the ACM*, 5(6):345–, 1962.

[16] Lester R. Ford. Network Flow Theory. Report P-923, The Rand Corporation, 1956.

[17] Lester R Ford and Delbert R Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8(3):399–404, 1956.

[18] Nejla Ghaboosi and Abolfazl Toroghi Haghighat. A path relinking approach for delay-constrained least-cost multicast routing problem. In *19th IEEE International Conference on Tools with Artificial Intelligence*. IEEE, 2007.

[19] Michel X. Goemans and Young-Soo Myung. A catalog of Steiner tree formulations. *Networks*, 23(1):19–28, 1993.

[20] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, 1988.

[21] Luis Gouveia. Using the Miller-Tucker-Zemlin constraints to formulate a minimal spanning tree problem with hop constraints. *Computers & Operations Research*, 22(9):959–970, 1995.

[22] Luis Gouveia. Multicommodity flow models for spanning trees with hop constraints. *European Journal of Operational Research*, 95(1):178–190, 1996.

[23] Luis Gouveia, Markus Leitner, and Ivana Ljubić. Hop constrained Steiner trees with multiple root nodes. *European Journal of Operational Research*, 236(1):100–112, 2014.

[24] Luis Gouveia, Markus Leitner, and Ivana Ljubić. The two-level diameter constrained spanning tree problem. *Mathematical Programming*, 2014.

[25] Luis Gouveia, Ana Paias, and Dushyant Sharma. Modeling and solving the rooted distance-constrained minimum spanning tree problem. *Computers & Operations Research*, 35(2):600–613, 2008.

[26] Luis Gouveia, Luidi Simonetti, and Eduardo Uchoa. Modeling hop-constrained and diameter-constrained minimum spanning tree problems as Steiner tree problems over layered graphs. *Mathematical Programming*, 128(1-2):123–148, 2011.

[27] S. Louis Hakimi. Steiner's problem in graphs and its implications. *Networks*, 1(2):113–133, 1971.

[28] Frank K. Hwang, Dana S. Richards, and Pawel Winter. *The Steiner tree problem*. Elsevier, 1992.

[29] Donald B. Johnson. Efficient Algorithms for Shortest Paths in Sparse Networks. *Journal of the ACM*, 24(1):1–13, 1977.

[30] Brian Kallehauge, Natashia Boland, and Oli B. G. Madsen. Path inequalities for the vehicle routing problem with time windows. *Networks*, 49(4):273–293, 2007.

[31] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, STOC '84, pages 302–311, New York, NY, USA, 1984. ACM.

[32] Richard M. Karp. *Reducibility among combinatorial problems*. Springer, 1972.

[33] Leonid G. Khachiyan. Polynomial algorithms in linear programming. *USSR Computational Mathematics and Mathematical Physics*, 20(1):53–72, 1980.

[34] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[35] Thorsten Koch and Alexander Martin. Solving Steiner tree problems in graphs to optimality. *Networks*, 32(3):207–232, 1998.

[36] Vachaspathi P. Kompella, Joseph C. Pasquale, and George C. Polyzos. Multicasting for multimedia applications. In *Proceedings IEEE INFOCOM 92: The Conference on Computer Communications*. IEEE, 1992.

[37] Vachaspathi P. Kompella, Joseph C. Pasquale, and George C. Polyzos. Multicast routing for multimedia communication. *IEEE/ACM Trans. Networking*, 1(3):286–292, 1993.

[38] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–48, 1956.

[39] Ailsa H. Land and Alison G. Doig. An automatic method of solving discrete programming problems. *Econometrica: Journal of the Econometric Society*, pages 497–520, 1960.

[40] Valeria Leggieri, Mohamed Haouari, and Chefi Triki. The Steiner tree problem with delays: A compact formulation and reduction procedures. *Discrete Applied Mathematics*, 164:178–190, 2014.

[41] Markus Leitner, Ivana Ljubić, and Markus Sinnl. A computational study of exact approaches for the bi-objective prize-collecting Steiner tree problem. *INFORMS Journal on Computing*, 27(1):118–134, 2015.

[42] Markus Leitner, Mario Ruthmair, and Günther R. Raidl. Stabilizing branch-and-price for constrained tree problems. *Networks*, 61(2):150–170, 2012.

[43] A. Yu Levin. Algorithm for the shortest connection of a group of graph vertices. *Soviet Mathematics. Doklady 12.*, 200(4):773, 1971.

[44] Ivana Ljubić and Stefan Gollowitzer. Layered graph approaches to the hop constrained connected facility location problem. *INFORMS Journal on Computing*, 25(2):256–270, 2013.

[45] Edward F. Moore. *The Shortest Path Through a Maze*. Bell Telephone System. Technical publications. monograph. Bell Telephone System., 1959.

[46] George L. Nemhauser and Laurence A. Wolsey. *Integer and combinatorial optimization*, volume 18. Wiley New York, 1988.

[47] Manfred Padberg and Giovanni Rinaldi. Optimization of a 532-city symmetric traveling salesman problem by branch and cut. *Operations Research Letters*, 6(1):1–7, 1987.

[48] Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM review*, 33(1):60–100, 1991.

[49] Tobias Polzin. *Algorithms for the Steiner Problem in networks*. PhD thesis, Universität des Saarlandes, Saarbrücken, 2003.

[50] Robert C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957.

[51] Anthony Przybylski, Xavier Gandibleux, and Matthias Ehrgott. Two phase algorithms for the bi-objective assignment problem. *European Journal of Operational Research*, 185(2):509 – 533, 2008.

[52] Anthony Przybylski, Xavier Gandibleux, and Matthias Ehrgott. A two phase method for multi-objective integer programming and its application to the assignment problem with three objectives. *Discrete Optimization*, 7(3):149–165, 2010.

[53] Rong Qu, Ying Xu, and Graham Kendall. A variable neighborhood descent search algorithm for delay-constrained least-cost multicast routing. In *Learning and Intelligent Optimization*, pages 15–29. Springer Science and Business Media, 2009.

[54] Bernard Roy. Transitivité et connexité. *Comptes Rendus de l'Académie des Sciences Paris*, 249:216–218, 1959.

[55] Mario Ruthmair. *On Solving Constrained Tree Problems and an Adaptive Layers Framework*. PhD thesis, Vienna University of Technology, 2012.

[56] Mario Ruthmair and Günther R. Raidl. A layered graph model and an adaptive layers framework to solve delay-constrained minimum tree problems. In *Integer Programming and Combinatoral Optimization*, pages 376–388. Springer Science and Business Media, 2011.

[57] Stefan Ruzika and Horst W. Hamacher. A survey on multiple objective minimum spanning tree problems. In *Algorithmics of Large and Complex Networks*, pages 104–116. Springer Science and Business Media, 2009.

[58] Hussein F. Salama, Douglas S. Reeves, and Yannis Viniotis. The delay-constrained minimum spanning tree problem. In *Computers and Communications, 1997. Proceedings., Second IEEE Symposium on*, pages 699–703. IEEE, 1997.

[59] Nina Skorin-Kapov and Mladen Kos. A GRASP heuristic for the delay-constrained multicast routing problem. *Telecommunication Systems*, 32(1):55–69, 2006.

[60] Kenneth Steiglitz and Christos H. Papadimitriou. Combinatorial optimization: Algorithms and complexity. *Printice-Hall, New Jersey*, 1982.

[61] Mirko B Vujošević and Milan Stanojević. A bicriterion steiner tree problem on graph. *Yugoslav journal of operations research*, 13(1):25–33, 2003.

[62] Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM (JACM)*, 9(1):11–12, 1962.

[63] Pawel Winter. Steiner problem in networks: a survey. *Networks*, 17(2):129–167, 1987.

[64] Richard T. Wong. A dual ascent approach for Steiner tree problems on a directed graph. *Mathematical Programming*, 28(3):271–287, 1984.

[65] Qingfu Zhang and Yiu-Wing Leung. An orthogonal genetic algorithm for multimedia multicast routing. *IEEE Transactions on Evolutionary Computation*, 3(1):53–62, 1999.