FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

A Multilevel Refinement Approach to the Rooted Delay-Constrained Steiner Tree Problem

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Thomas Seidl
Matrikelnummer 0525225

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuung: ao. Univ.-Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Mitwirkung: Univ.Ass. Dipl.-Ing. Mario Ruthmair

Wien, 25.09.2011

(Unterschrift Verfasser) (Unterschrift Betreuung)

Erklärung zur Verfassung der Arbeit

Thomas Seidl
Randhartingergasse 12/26, 1100 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbstständig 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.

_____ _____

Acknowledgements

I would like to thank my advisor, Prof. Dr. Günther Raidl, for letting me work on this thesis and for his help and suggestions with creating it.

I also thank the Vienna University of Technology for the years of education I received there, and for the prolific scientific environment it provided.

My special thanks go to my mentor for this thesis, Dipl.-Ing. Mario Ruthmair. Without his countless suggestions, our numerous discussions, his help and his thorough reviews, this thesis would never have been completed.

Lastly, I want to sincerely thank my parents, who supported me in every possible way throughout my education and without whom none of this would have been possible.

Abstract

The Rooted Delay-Constrained Steiner Tree Problem (RDCSTP) is a variant of the well-known Steiner Tree Problem on a graph in which the paths to all terminal nodes are restricted by a certain maximum delay. The problem mostly appears in the context of network routing for multicasts, i.e., sending packages from a fixed source to a subset of other participants in the network. Since the RDCSTP belongs to the class of $\mathcal{NP}$-hard problems it is in general not possible to solve large instances exactly in a reasonable amount of time. Therefore, the focus mostly lies on developing good heuristics that can still solve large instances comparatively fast to near optimality.

In this thesis a Multilevel Refinement heuristic – which has already been successfully applied to other problems like the Graph Partitioning Problem – is implemented as an improvement heuristic for the RDCSTP. In the general approach of this metaheuristic the problem's complexity is first iteratively reduced while still maintaining its general characteristics. The problem is thereby simplified and can at the top level finally easily be solved. Then, the solution on this highest level is refined until a solution for the original problem is obtained.

The algorithm introduced here implements the Multilevel Refinement approach as an improvement heuristic, iteratively changing an existing solution. However, it is designed in a way that also allows it to be used to construct an initial solution. Another distinctiveness is that, due to the additional delay constraints, supplementary data structures have to be used to avoid creating invalid solutions on higher levels as much as possible. In the refinement phase an additional improvement algorithm, the Key Path Improvement, is executed on each level, drastically increasing result quality.

Experimental tests are carried out, evaluating the performance of the algorithm on large instances and comparing it to other algorithms in the literature. The obtained results are promising and indicate that the Multilevel Refinement metaheuristic is indeed a competitive approach for the RDCSTP.

Kurzfassung

Das *Rooted Delay-Constrained Steiner Tree Problem* (RDCSTP) ist eine Variante des bekannten Steinerbaum-Problems auf einem Graphen in welcher die Pfade zu allen Zielknoten durch eine bestimmte maximale Verzögerung beschränkt sind. Das Problem tritt hauptsächlich im Bereich des Netzwerk-Routings beim Multicast auf, das heißt wenn Pakete von einer einzelnen Quelle zu einer bestimmten Untermenge der anderen Netzwerk-Teilnehmer gesendet werden sollen. Da das RDCSTP, wie das ursprüngliche Steiner-Problem, zur Klasse der $\mathcal{NP}$-schwierigen Probleme gehört, ist es allgemein nicht möglich die exakte Lösung einer großen Probleminstanz in vertretbarer Zeit zu finden. Der Fokus der Forschung liegt daher großteils auf der Entwicklung guter Heuristiken, die auch bei großen Probleminstanzen in der Lage sind in vergleichbar kurzer Zeit zu möglichst guten Lösungen zu kommen.

In dieser Arbeit wird hierfür die *Multilevel-Refinement-Heuristik* – die bereits erfolgreich auf etliche andere Probleme, wie das *Graph Partitioning Problem*, angewandt wurde – als Verbesserungsheuristik für das RDCSTP entwickelt. Grundsätzlich werden bei dieser Metaheuristik in einem ersten Schritt Knoten sukzessive zusammengefasst um den Graphen auf höheren "Levels", mit weniger Knoten, darzustellen. Das so vereinfachte Problem kann dann auf der höchsten Abstraktionsebene in simpler Weise gelöst werden. Dann wird diese Lösung schrittweise wieder soweit verfeinert, bis eine Lösung für das ursprüngliche Problem erreicht wird.

Der hier vorgestellte Algorithmus für das RDCSTP implementiert diesen Multilevel-Ansatz als Verbesserungsheuristik, die eine existierende Lösung iterativ verändert. Er wurde allerdings in einer Weise entworfen, die es ihm ebenso erlaubt eine Anfangslösung selbst zu generieren. Eine weitere Besonderheit ist, dass wegen der zusätzlichen Verzögerungs-Einschränkung weitere Datenstrukturen benötigt werden, um auf höheren Levels möglichst gültige Lösungen zu erzeugen. Außerdem wird während der Verfeinerung der Lösung auf jedem Level eine weitere Verbesserungsheuristik angewandt, das *Key Path Improvement*, welches die Lösungsqualität drastisch verbessert.

Umfangreiche experimentelle Tests wurden durchgeführt um die Leistungsfähigkeit des Algorithmus bei großen Instanzen zu messen, und ihn mit anderen Algorithmen aus der Literatur zu vergleichen. Die hierbei erhaltenen Ergebnisse sind durchwegs sehr positiv und weisen somit darauf hin, dass der verfolgte Multilevel-Ansatz tatsächlich eine konkurrenzfähige Heuristik für das RDCSTP darstellt.

Contents

CHAPTER 1

Introduction

1.1 The Rooted Delay-Constrained Steiner Tree Problem

One of the most frequent algorithmic problems encountered in network routing is to connect participants with each other as efficiently as possible (for some definition of "efficient") and a large multitude of literature is available for these problems. However, such simplicistic views often fail to take other concerns into account which might still be of great significance for the practical problem.

For example, consider the repeated multicast of information from a fixed source to a subset of the participants in a network. Fixed connections should be established to faciliate such transmissions, which should of course be done with as little cost as possible.

To represent this problem in an abstract way, we model the network as a connected graph, with nodes representing all network participants and the edges representing the possible connections between these participants. Costs are defined on all edges to provide a measure of how efficiently these connections can be used, or of how desirable it is to use them. The problem then consists of creating a tree of minimum cost in this graph which contains all of the destinations of the multicast, and of course the source.

This problem is already well-known as the *Steiner Tree Problem* in literature [1, 2]. There, the nodes that have to be included in the tree are called *terminals*, while all other nodes are called *Steiner nodes*. The problem was proven to be $\mathcal{NP}$-hard in [3].

However, as a representation of the original problem this comes short of grasping a vital aspect of some multicasts, namely network delays. Especially when streaming audio or video, maybe even in the context of video conferencing, being able to cheaply enable a connection is often only important as long as the delay between the source and the destinations stays within certain acceptable boundaries. The problem model therefore has to be expanded to takes this additional criterion into account.

In addition to the previous definitions, we now define network delays for all possible connections, i.e., all edges of the graph. Also, a certain threshold is given, a maximum delay bound which no delay between the source node and a terminal may exceed in a valid solution. This

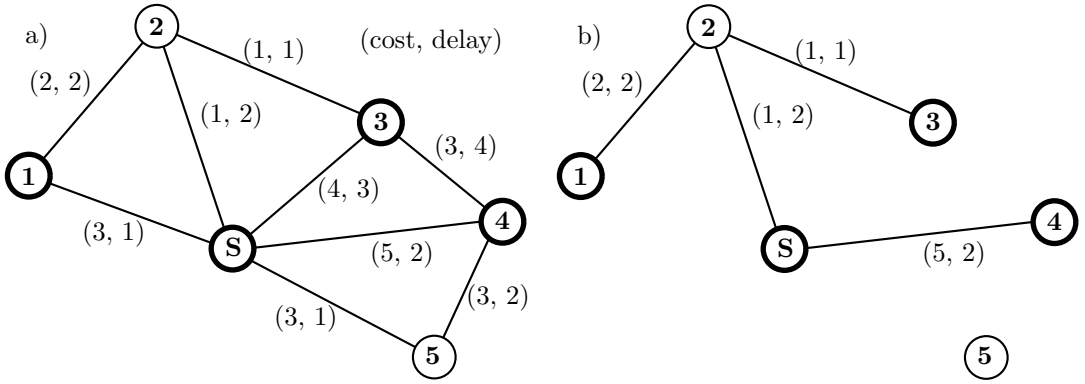

Figure 1.1: An example for an instance of the RDCSTP. a) The given graph with edge costs and delays. The nodes with thick borders are the terminals, s denotes the source node. b) The optimal solution tree for the delay bound $B = 4$.

variant is known as the *Rooted Delay-Constrained Steiner Tree Problem* (RDCSTP), or *Multicast Routing Problem With Delays*. It, too, is $\mathcal{NP}$-hard [4].

Figure 1.1 shows an example for an instance of the RDCSTP. The Steiner node 2 is used to decrease the overall costs of the tree while node 5 is excluded since using it to connect node 4 would be more expensive than the direct edge. Using edge $\{3, 4\}$ to connect node 4 would decrease the tree cost but violate the delay bound.

Since in practical usage one might often encounter large instances of the problem which can – due to the $\mathcal{NP}$-hardness of the problem – not be solved exactly in a reasonable amount of time, heuristic algorithms for the RDCSTP have for the most part been the focus of research in this area. While improving existing exact algorithms to increase their range of use is also a worthwhile effort, we still decided to research the suitability of an existing metaheuristic for the RDCSTP in this thesis. Furthermore, a variant of the problem arises in circuit design, where large problem instances might also occur frequently.

1.2 The Multilevel Refinement heuristic

The *Multilevel Refinement heuristic* is a meta-heuristic that has already been applied successfully to other graph problems, like the graph partitioning problem [5, 6]. Its basic idea is to reduce the problem complexity by successively reducing the size of the graph while still maintaining enough information about the original graph for the constructed solution to be useful for the original problem.

An application of the Multilevel Refinement heuristic on a problem consists of three phases: coarsening, solving the coarsened problem and refinement. In the coarsening phase the problem is successively simplified to create new "levels" of it. This has to be done in a way that ensures the higher levels of the problem still represent the original problem in most characteristics. In the case of Multilevel Refinement on graphs the operation usually chosen here is to merge edges or nodes to form the higher levels, thus still preserving the rough structure of the graph, with

node clusters on lower levels being represented by single nodes on higher ones. There are lots of variants here, though, and completely other strategies for coarsening a graph could also be employed.

The coarsening is executed until a certain abort criterion is met. For instance the condition could be that no more coarsening is possible, or that the problem complexity is below a certain threshold at which it can easily be solved exactly. At this point, the second phase of the algorithm is executed: the problem on the highest level is solved, which should now be easily possible.

In the concluding third phase, the refinement phase, this solution for the highest level is then iteratively extended to provide solutions for lower levels of the problem. The way this is accomplished is highly problem-specific, and the suitability of the Multilevel Refinement heuristic for a given problem largely depends on whether this step can be easily executed. Usually the changes between levels of the problem can analogously be applied to the solution, thus yielding solutions for lower levels. In graph problems when merging nodes or edges during coarsening, refinement is usually possible by replacing the merged nodes or edges in the solution by the corresponding nodes and edges from the next-lower level of the graph.

The refinement is executed until a solution for the original problem is obtained. In addition to the basic refinement, one or more extra improvement heuristics can be applied to the solution as well on each level (or only on certain ones) to further improve the final solution quality.

Figure 1.2 illustrates how an application of the Multilevel Refinement metaheuristic on a graph problem might look. In this example the graph is coarsened by merging nodes to form higher levels.

A good summary of the metaheuristic can be found in [7]. Section 2 contains examples of previous successful applications of the approach in the literature.

Since the Multilevel Refinement metaheuristic has already proven successful for several graph problems, but has not yet been applied to the RDCSTP, we were interested in evaluating its appropriateness for this problem, too. One of our specific hopes was that using the Multilevel Refinement approach would bring more impact on the global scale to local improvement heuristics for larger problem instances.

Structure of the thesis

The remainder of this thesis is structured as follows: Section 2 contains an overview of existing work in the literature which is relevant to this thesis. Section 3 then explains the general algorithm we designed in this paper and what specific problems had to be solved. This is then elaborated on in Section 4 where we more closely discuss some details of the implementation of our algorithm. In Section 5 we list the results of several benchmarks and comparisons we executed on the final program. We conclude with a short summary of the thesis and an outlook on possible future work in this area in Section 6.

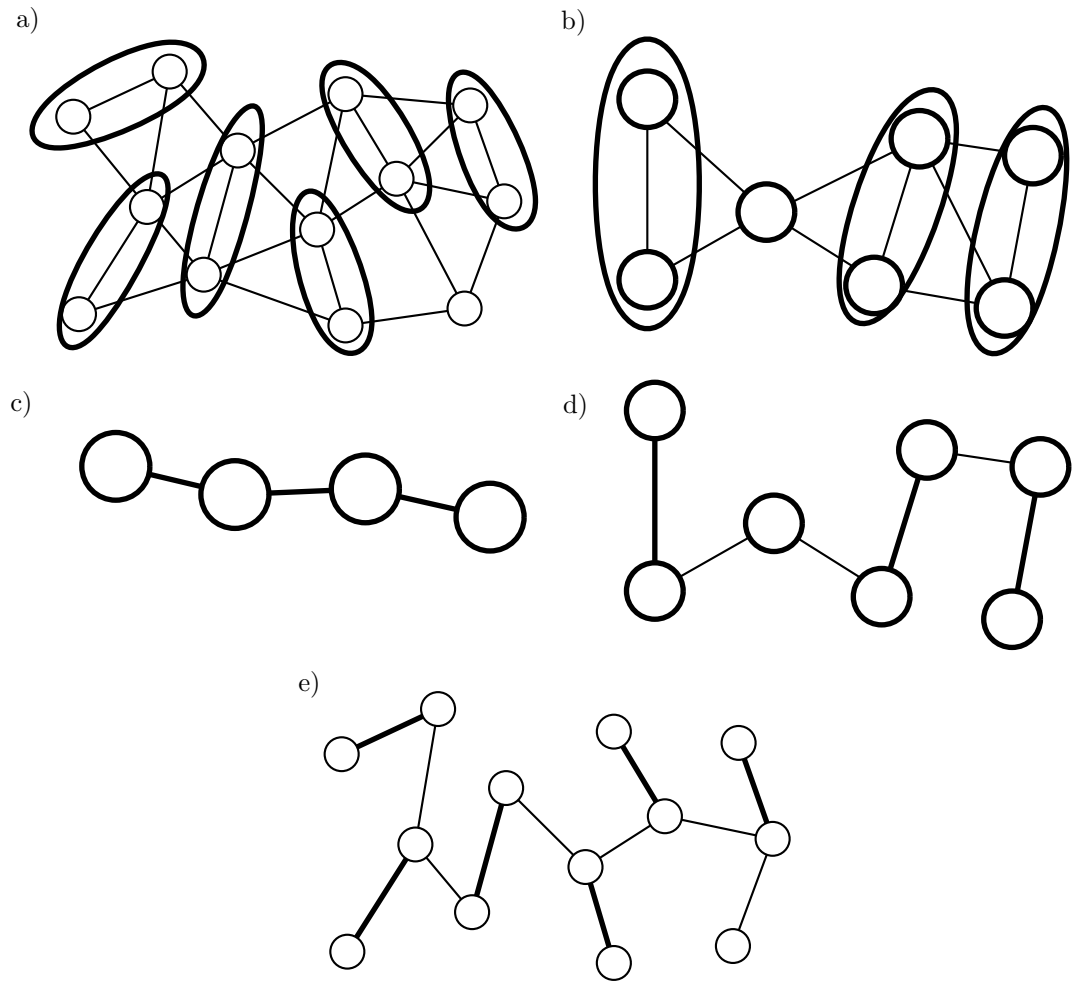

Figure 1.2: An example of an application of the Multilevel Refinement heuristic on a graph problem. a) The original problem on level 0. Nodes which will be merged for the next level are encircled. b) The corresponding problem on level 1, after the shown merging operations were executed. c) The graph on level 2. We consider the graph in this form already a valid solution tree itself and therefore stop coarsening. d) The solution tree refined to level 1. Edges that were newly added to the tree are drawn heavier. These are the edges which previously connected the afterwards merged nodes on level 1 of the graph. e) After refining again to level 0, a solution for the original problem is obtained.

CHAPTER 2

Related work

The RDCSTP (mostly under aliases such as *Delay-Constrained Multicast Routing*) is already well-known and has been the focus of research numerous times.

2.1 Preprocessing

Preprocessing techniques are an important part of the research for the RDCSTP as they allow to significantly reduce the problem size in a manner that is in no way dependent on the concrete algorithm used to solve the problem. The algorithms mentioned here were therefore also used in the final program evaluated in this thesis.

In [8], some simple cases are described in which edges can safely be removed from the graph. This includes edges that can never be part of a valid solution (due to their delay being too high) or that cannot be part of an optimal solution (e.g., when simple triangle inequalities do not hold on costs and delays for some circle of three edges). Although the paper discusses the Rooted Delay-Constrained Minimum Spanning Tree Problem – a specialized variant of the RDCSTP in which all nodes are terminals –, these techniques can equally be used for the RDCSTP itself.

Preprocessing techniques that also take the special properties of Steiner nodes into account are described in [9], for the original Steiner Tree Problem. This includes simple measures, like removing Steiner nodes that are leaves, but also some more complex checks. For inclusion in our program, these had to be extended to take the edge delays into account.

2.2 Heuristic algorithms

Due to the $\mathcal{NP}$-hardness of the problem, heuristic algorithms have been very popular for the RDCSTP and there are already numerous existing algorithms for it.

Construction heuristics

The first mention of the problem in literature was in [4], where an adapted version of the algorithm in [10] for the Steiner tree problem without delays was applied to the problem.

In [11], a minimum-delay tree is constructed as a first step. Then, the delay-bounded tree is iteratively improved to minimize costs. Paper [12] proposes an adapted version of an algorithm from [13], for the unicast routing problem. A solution is constructed by iteratively adding terminals to the tree until the whole set of terminals is included. However, this paper also takes into account asymmetric costs and delays, which are not considered here.

Improvement and meta-heuristics

A genetic algorithm is considered in [14] for a slight variant of the problem, which adds a constraint on a third edge property, the bandwidth, and allows different delay bounds per terminal.

More recently, in [15] a path-relinking approach was applied to the problem. This is a genetic algorithm in which new solutions are constructed by conceptually connecting two existing solutions in the solution space and examining all solutions along this path.

The well-known *Greedy Randomized Adaptive Search Procedure* (GRASP) heuristic was also employed several times for the RDCSTP [16, 17], generally leading to promising results. The latter of those uses a *Variable Neighborhood Descent* (VND) algorithm as the local search heuristic, which was first introduced as a stand-alone heuristic in [18].

One of the most recent works regarding the RDCSTP can be found in [19]. There, a combination of the path-relinking approach with a scatter search heuristic was developed, also showing very good results. The experimental data from this paper will therefore later be used for comparison with the algorithm developed here.

2.3 Exact algorithms

Even though it is hard to design efficient exact algorithms for the RDCSTP and their use in practice is limited due to the quickly increasing complexity for larger instances there are already several approaches in this category. The first *Integer Linear Programming* formulation of the problem can be found in [20], with the addition of bandwidth to the problem.

More recently, [21, 22] investigated some other (also *Mixed Integer Programming*) formulations and additional restrictions. In [23], these were then combined with a *Branch-and-Cut* approach.

A Mixed Integer Programming formulation using layered graphs is discussed in [24]. The paper also introduces a technique called “adaptive layers”, in which new layers are iteratively added to an initially smaller problem formulation to tighten lower and upper bounds.

In [25], the stabilized column generation algorithm from [26] is expanded with a branch-and-bound approach and an additional pricing strategy to form a stabilized branch-and-price algorithm.

Although experimental results for the newer exact algorithms proved very promising, all of these exact approaches quickly reach their limits for complete graphs with more than about 100 nodes. They are therefore no viable option for large problem instances.

2.4 Multilevel Refinement heuristic

As already mentioned, the Multilevel Refinement approach has successfully been applied to the Graph Partitioning Problem in [5, 6]. Earlier applications of the Multilevel approach are discussed in [27], mentioning applications like Multilevel Annealing, Multilevel Monte-Carlo, and other heuristics.

In [28], the Multilevel Refinement heuristic is used with great success on the Travelling Salesman Problem, considerably improving the results of the traditional Chained Lin-Kernighan algorithm, which is used as the improvement heuristic on each level. Such an algorithm is also discussed, amongst many others, in [29].

The Graph Coloring Problem has also been tackled with the Multilevel Refinement heuristic [30]. There, both an iterated greedy algorithm and tabu search are tested as improvement heuristics for the refinement, in both cases improving the results obtained by using the algorithms without the Multilevel addition.

2.5 Other related work

The Multilevel Refinement heuristic has not yet been applied to the RDCSTP. However, in [31, 32] it is applied to the Rooted Delay-Constrained Minimum Spanning Tree Problem (RDCMSTP).

In the algorithm presented there, however, nodes are not explicitly merged together. Instead, on each level first a number of so-called "supervertices" are selected according to their "ranking score" (a value computed by the number and cost/delay values of their adjacent edges). Each of the remaining nodes has then to be connected to one of these supervertices by a direct edge. The supervertices and all edges between them then become the problem graph on the next level.

This process is continued until only the source node remains, which then contains a valid solution for the original problem. The algorithm therefore does not include an explicit refinement phase, thus also precluding the use of an improvement heuristic on each separate level. An improvement heuristic (a VND described in [33]) is only applied to the final solution.

CHAPTER 3

Algorithm

As explained in the introduction, the Multilevel Refinement metaheuristic consists of three primary steps: coarsening, solving the coarsest problem and refining. The following chapter describes in detail how each of these phases was implemented for the RDCSTP and what specific problems had to be resolved.

The algorithm was used as an improvement heuristic, in the form of an *Iterated Multilevel Refinement*. This means that the three steps were executed repeatedly, with the solution of the previous iteration being used as the starting solution for the next one, until certain criteria (e.g., a time limit or a certain number of iterations without any improvements) were met.

The iterations in this outer loop were also used to dynamically adapt some parameters of the algorithm, as described later.

3.1 Problem formulation and definitions

Mathematically, the RDCSTP is defined as follows. Given are a connected graph $G = (V, E)$ consisting of a set V of nodes and a set E of edges; a cost function

$$\mathcal{C}\colon E \to \mathbb{R}^+$$

and a delay function

$$\mathcal{D}\colon E \to \mathbb{R}^+$$

defined for all edges of the graph; a subset $S \subset V$ of terminals; a source node $s \in V$; and a delay bound $B \in \mathbb{R}^+$ [4].

For a tree $T = (V_T, E_T)$ (with $V_T \subseteq V$ and $E_T \subseteq E$) in G and two nodes $u, v \in V_T$, we define $\mathcal{P}_T(u, v)$ as the path between these nodes in T, i.e., the set of edges $e \in E_T$ needed to connect them. Likewise, we define $\mathcal{P}_G(u, v)$ as the path between u and v which has the lowest

delay, or an arbitrary one such path if there is more than one. We then define the cost and delay functions on such a path p in the following way:

$$\mathcal{C}^*(p) = \sum_{e \in p} \mathcal{C}(e)$$
$$\mathcal{D}^*(p) = \sum_{e \in p} \mathcal{D}(e)$$

A valid solution to the RDCSTP is then a tree $T = (V_T, E_T)$ for which the following conditions hold:

$$\forall v \in S \colon v \in V_T$$
$$\forall v \in V_T \colon \mathcal{C}^*(\mathcal{P}_T(s, v)) \leq B$$

An optimal solution T^* is a valid solution which has a minimal total cost

$$\mathcal{C}^*(E_T) = \sum_{e \in E_T} \mathcal{C}(e)$$

among all valid solutions. The existence of more than one optimal solution is of course possible.

3.2 General approach

Algorithm 3.1 gives an overview of the general approach of the Multilevel Refinement heuristic that we wanted to apply to the RDCSTP. As explained in Section 1, the algorithm consists of three phases: the problem is first coarsened iteratively to create higher levels of abstraction, in some problem-specific way. Once a certain criterion is met (usually when the current level of the problem can be solved easily enough) this process is stopped and a solution for this highest level problem created. The solution is then successively refined to lower levels in this problem hierarchy, again in a problem-specific way, until we finally obtain a solution for the original problem.

3.3 Coarsening phase

As explained, the purpose of the coarsening phase is to iteratively reduce the graph until the problem becomes trivial. In principle, this could be done in various ways. The most intuitive approach to this seems, however, to be to merge nodes. As this also leads to a comparatively simple layout of the general algorithm, this variant of coarsening was therefore used here. The schematic approach for this is described in Algorithm 3.2.

`listSortedEdges` will retrieve all edges of the graph on the current level. As the nodes that will be merged are determined by the order of the list returned by this function, the sorting

Algorithm 3.1: `multilevelRefinement()`

Purpose: Solves a problem heuristically using the Multilevel Refinement algorithm.
Input: A (usually combinatorial) problem P.
Output: A possible solution to P.

```
1  P_0 ← P;
2  l ← 0;
3  while P_l not trivial do                  // coarsen the problem until it can easily be solved
4  |  l ← l + 1;
5  |  P_l ← simplified form of P_{l-1};
6  end while
7  S_l ← solution for P_l;                   // solve the problem on the highest level
8  while l > 0 do                            // refine the solution again
9  |  l ← l − 1;
10 |  S_l ← solution to P_l, based on S_{l+1};
11 end while
12 return S_0;
```

Algorithm 3.2: `coarsen()`

Purpose: Coarsens the graph into a reduced form which can easily be solved, saving the necessary information to later refine it again.

```
1  P_0 ← G;
2  level ← 0;
3  while further coarsening possible do
4  |  level ← level + 1;
5  |  edges ← listSortedEdges();
6  |  merged ← ∅;
7  |  for {u,v} ∈ edges do
8  |  |  if u ∉ merged and v ∉ merged then
9  |  |  |  mergeEdge({u,v}) ;              // This does the actual merging.
10 |  |  |  if edge merged then
11 |  |  |  |  merged ← merged ∪ {u,v};
12 |  |  |  end if
13 |  |  end if
14 |  end for
15 |  P_level ← current state of G;
16 end while
17 return the levels of the graph in P_i;
```

there has to be well thought-out. Needed here is a heuristic measure on merging which edges will result in the best final solutions. As edges "contained" in a merged node are conceptually always part of the tree, these would usually be edges which might form a good solution.

From this it is clear that the primary indicators of the edge "score", by which edges will be ordered in `listSortedEdges` should be the edges' costs and delays. As these are have entirely different dimensions, multiplying them is the only reasonable way in which to combine them to form a score. Also, since cost and delay will probably have differently strong influences on the edge quality (in terms of the nodes connected by which edges should be merged), we allow for exponents for both cost and delay to balance their influence on the edge scores accordingly.

Another thing that should influence the edge score, since the algorithm should implement an Iterated Multilevel Refinement approach, is the previous solution. Nodes connected by edges that already were part of the previous solution should be more likely to be merged again. Therefore, the score of edges contained in the previous solution tree should be decreased (indicating a better score, in our case).

We also took into account the possibility that the types of nodes merged could influence the final solution quality. For instance, always preferring to merge terminals with other terminals instead of Steiner nodes, or Steiner nodes with other Steiner nodes, could conceivably improve or worsen the algorithm's results. A factor to represent this possible effect was hence also included in the formula.

Finally, we also added a random factor to the formula to faciliate larger variety in searching the solution space, and to avoid getting stuck in local optima too easily. We thus arrived at the following formula for the edge score:

$$score(e) = \mathcal{C}^{\alpha}(e) \cdot \mathcal{D}^{\beta}(e) \cdot treeBoost(e) \cdot edgeTypeBoost(e) \cdot rand()$$

$$treeBoost(e) = \begin{cases} \frac{1}{\mathsf{treeBoost}} & \text{if } e \text{ in previous solution} \\ 1 & \text{otherwise} \end{cases}$$

$$rand() = \begin{cases} 1 & \text{if } \mathsf{randBoost} = 0 \\ 2^{\mathit{GetRandom}(\mathsf{randBoost})} & \text{otherwise} \end{cases}$$

Here, randBoost and treeBoost are parameters of the algorithm whose effects on the solution quality will be studied in Section 5. *GetRandom(σ)* is a function which returns a random number, following a normal distribution with mean 0 and variance σ^2. In the formula it is used as an exponent with basis 2, so the probability of multiplying and of dividing the score by a certain value would be equal (and would decrease for increasing values).

edgeTypeBoost(e) is a function which can return an additional boosting factor according to the type of nodes edge e connects. It is explained in Algorithm 3.3. As can be seen, its concrete effect is dependent on two additional parameters, twoTermBoost and mixedEdgeBoost. These parameters, too, will be evaluated in Section 5.

First, it is counted how many of the two end nodes of the edge are terminals. Then, this will result in a boost or penalty to the edge score. A positive value of the twoTermBoost parameter means that edges connecting two terminals should have better scores. (Note that we did not use non-zero values with an absolute value less than 1 for these two parameters. For such values,

Algorithm 3.3: `edgeTypeBoost()`

Purpose: Determines a boost or penalty for an edge, depending on the types of nodes it connects.

Input: An edge between two nodes `u` and `v`.

Output: The value by which the edge score should be multiplied.

```
1  penalty ← 1;                                  // 1 does nothing, higher values are penalties.
2  numTerminals ← |S ∩ {u, v}|; // number of the edge's end nodes which are terminals
3  if twoTermBoost > 0 and numTerminals < 2 then
4  |  penalty ← penalty · twoTermBoost;
5  else if twoTermBoost < 0 and numTerminals > 0 then
6  |  penalty ← penalty · (−twoTermBoost);
7  end if
8  if mixedEdgeBoost > 0 and numTerminals ≠ 1 then
9  |  penalty ← penalty · mixedEdgeBoost;
10 else if mixedEdgeBoost < 0 and numTerminals > 0 then
11 |  penalty ← penalty · (−mixedEdgeBoost);
12 end if
13 return penalty;
```

the effects would of course be reversed.) Therefore, if the parameter has a positive value and the number of terminals is not 2, the score is multiplied by the parameter and thus worsened. A negative value of the twoTermBoost parameter conversely means that edges connecting two Steiner nodes should have better scores. Therefore, in this case the scores of edges where the number of terminals is not 0 are multiplied by the absolute value of the parameter. Likewise, the mixedEdgeBoost parameter is treated, where positive values mean better scores for edges connecting a terminal and a Steiner node, and negative values result again in a score boost for edges connecting two Steiner nodes.

On the whole, this sorting results in edges which have lower costs or delays, or which were already part of the previous solution, being inspected sooner (not accounting for the random factor) and therefore being more likely to be merged.

The algorithm for merging two nodes, referred to as *mergeEdge* here, is detailed in Algorithm 3.4. The function *isFeasible* checks whether the given edge can, in theory, be part of a feasible solution on the current level. In principle, this is the case if the minimum delay from the source to at least one of the connected nodes is less than or equal to the delay bound minus the edge's delay. The computation is significantly more complicated for higher levels, however. A detailed description of the function implementation will be given in Algorithm 4.1.

In *mergeEdge* it is first checked whether the edge between the two nodes that should be merged can still be part of a valid solution in the current graph. If this is not the case, the edge is simply removed from the graph and the function returns. Otherwise, the edge is removed and a new node `n` inserted into the graph. The new node is exactly then a terminal if at least one of the two merged nodes is a terminal.

Algorithm 3.4: `mergeEdge()`

Purpose: Merges the two nodes connected by `e` into a single one which inherits all edges to other nodes. Remembers all modifications so that they can be undone later during refinement.

Input: An edge `e`, connecting the nodes `u` and `v`.

```
1  if not isFeasible(e) then
2  |  E ← E \ {e};
3  |  return;
4  end if
5  E ← E \ {e};
6  V ← V ∪ {n};
7  if u ∈ S or v ∈ S then
8  |  S ← S ∪ {n};
9  end if
10 for i ∈ V do
11 |  delay ← ∞;
12 |  if {i,u} ∈ E then
13 |  |  if isFeasible({i,u}) then
14 |  |  |  cost ← C({i,u});
15 |  |  |  delay ← D({i,u});
16 |  |  end if
17 |  |  E ← E \ {{i,u}};
18 |  end if
19 |  if {i,v} ∈ E then
20 |  |  if isFeasible({i,v}) and D({i,v}) < delay then
21 |  |  |  cost ← C({i,v});
22 |  |  |  delay ← D({i,v});
23 |  |  end if
24 |  |  E ← E \ {{i,v}};
25 |  end if
26 |  if delay < ∞ then
27 |  |  E ← E ∪ {{i,n}};
28 |  |  C({i,n}) ← cost;
29 |  |  D({i,n}) ← delay;
30 |  end if
31 end for
32 V ← V \ {u,v};
33 S ← S \ {u,v};
```

Then all nodes are inspected which are neighbors of either one or both of the nodes to be merged. For each node among these neighbors, the edge or edges connecting it to the merged nodes will be checked for whether they could still be part of a valid solution once the nodes are merged. If this is not the case, the concerned edge is just removed. Otherwise, cost and delay of the edge are remembered before it is removed. If a node is connected to both of the nodes that will be merged, only the cost and delay of the edge with the lower delay will be remembered. Now, after the edge or edges to a neighbor are removed, an edge with the remembered cost and delay between this node and the newly inserted node `n` will be created, unless no values were remembered for that node. This is done for all neighbors of the two nodes. Afterwards, they, too, are removed from the graph.

The choice of "keeping" the edges with lower delays, regardless of their costs, was found sensible here as this ensures a wider range of possibilities on higher levels, not restricting the solution space too much. Other decision criteria, like one based again on the score formula, could be used here, too. Not explicitly discarding one edge but keeping information about both was also considered but eventually judged as not being practical, as this would result in large amounts of additional data on higher levels which would ultimately defeat the purpose of the Multilevel Refinement approach.

An example

Figure 3.1 shows an example of two nodes, u and v, being merged. All edges connected to the two merged nodes are inspected. In the example we assume that the minimum delay from the source to node 1 plus the delay of the path to v exceeds the delay bound, causing the edge $\{1, u\}$ to be removed from the graph.

The edges $\{2, u\}$ and $\{4, v\}$ pose no problems. They are inherited identically by the new node n. In case a node is connected to both merged nodes, like 3 is in the example, the edge with the lower delay is kept, as explained. A detailed example of what additional information was stored by our implementation of the algorithm will be given in Section 4.2.

3.4 Solving the highest level

The coarsening phase ends when no more nodes can possibly be merged. This means that no edges between non-source nodes are present in the graph anymore, leading to a graph like the one in Figure 3.2. It is now of course rather trivial to construct a valid solution on this highest level, which is therefore done via Algorithm 3.5.

The only problem here is that, due to the limiting effect of the coarsening to the solution space, it is possible that some terminals are not connected to the rest of the graph at all anymore. For those, a special solution has to be found. We decided to add a so-called "virtual edge" to the graph, which does not really exist in the original graph, but represents the cost and delay of the shortest delay path from the node to the source in the original graph. As the node itself most likely does not exist in the original graph, a random one of its sub-nodes is selected and its shortest delay path used.

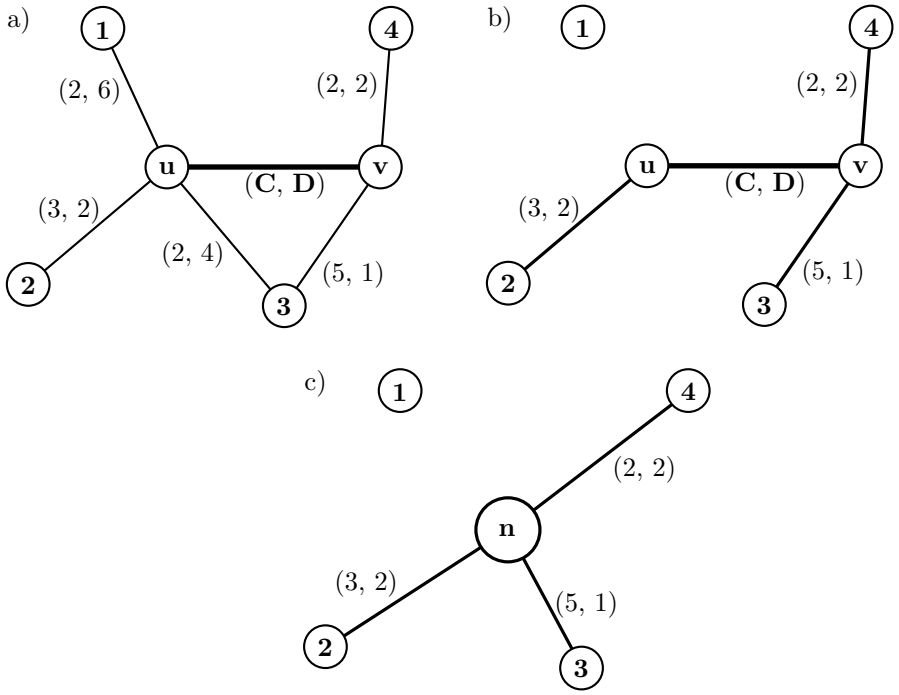

Figure 3.1: An example for merging two nodes. a) A subsection of a problem graph, with the nodes u and v being merged. b) The selected edges to neighboring nodes which will be inherited by the new merged node. c) The resulting merged node n and its edges.

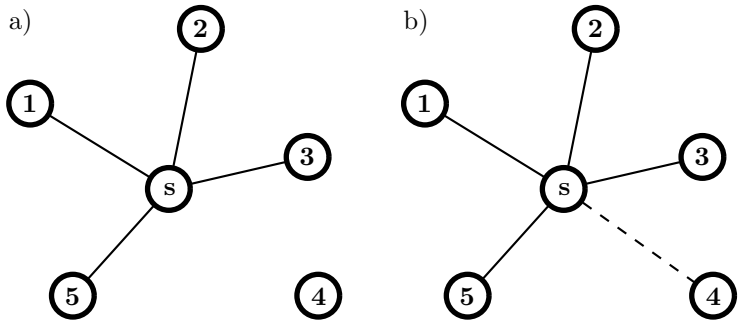

Figure 3.2: An example of what the graph might look like after the coarsening phase. a) The complete graph after coarsening. b) If a terminal is not connected to the source anymore, a virtual edge is inserted.

Algorithm 3.5: `solveCurrentLevel()`

Purpose: Creates a new solution T for the graph G on the current level.

```
1  V_T = S ∪ {s};
2  E_T = ∅;
3  for i ∈ S do
4      if {s, i} ∈ E then
5          E_T ← E_T ∪ {s, i};
6      else
7          p ← P_{P_0}(s, i);                    // shortest delay path from the original problem
8          E_T ← E_T ∪ {s, i};
9          C({s, i}) ← C*(p);
10         D({s, i}) ← D*(p);
11     end if
12 end for
13 T ← (V_T, E_T);
```

These virtual edges are saved in a special way and later removed at the end of refining (right before the last improvement phase), if they are still present in the graph, and replaced by the path they represent. In doing so, we of course also take care not to create a cycle in the solution tree.

3.5 Refinement phase

The refinement phase is the final phase of an iteration of the algorithm. All changes made during coarsening are undone, to finally arrive back at the original graph on level 0. The basic algorithm for this phase is described in Algorithm 3.6.

When undoing changes in the graph, it is important to also make the necessary adjustments to the solution to reflect those changes. For example, when a merged node is split into its two sub-nodes again, and the node is present in the tree, we also have to replace the node in the tree with the sub-nodes. Additionally we have to add the edge connecting the two nodes, as well as re-connect all adjacent edges of the merged node to its two sub-nodes (depending on the sub-node to which the respective edge was originally connected). This is demonstrated with an example in Figure 3.3.

Looking at the algorithm you will notice that after undoing the changes of each level a repair algorithm is executed. This was necessary as, even with several measures in place to keep higher level representations as accurate was possible, we could not completely avoid the possibility of creating invalid solutions on higher levels without too large performance drawback. In this step during refinement we would therefore check all terminals for their delay to the source, repairing parts of the tree where necessary.

As the final part of the refinement of each level, there is also an improvement phase with a

Algorithm 3.6: `refine()`

Purpose: Refines the coarsened graph back to its original form, also transforming and improving the solution tree while doing so.

Input: The current level l of the graph.

```
1  while l > 0 do
2  |  l ← l − 1;
3  |  changes ← changes between G_l and G_{l+1};
4  |  for change ∈ changes do
5  |  |  undo change in G and T;
6  |  end for
7  |  remove edges to Steiner nodes of degree 1 from T;
8  |  if l = 0 then
9  |  |  replace all virtual edges in T with corresponding paths;
10 |  end if
11 |  detect and repair delay bound violations;
12 |  execute improvement heuristic;
13 end while
```

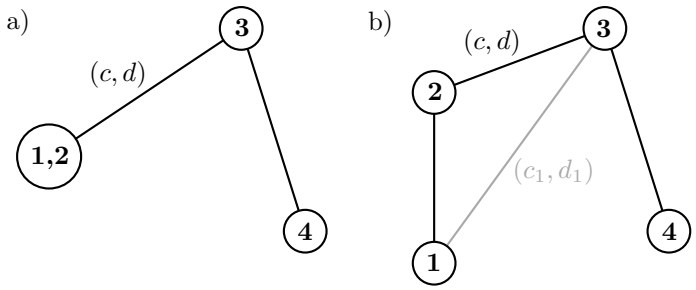

Figure 3.3: An example of a merge operation in the graph being undone for the tree. a) The initial situation on the higher level. b) The tree after the merge operation is undone. The gray edge is only part of the graph, not of the tree. Since it has got a different cost and/or delay than the edge on the higher level it is clear that the edge to the other node has to be the one represented by the higher-level edge. Since 4 is not connected to the merged node on the other level, edges to it are ignored when undoing the merge operation on the tree.

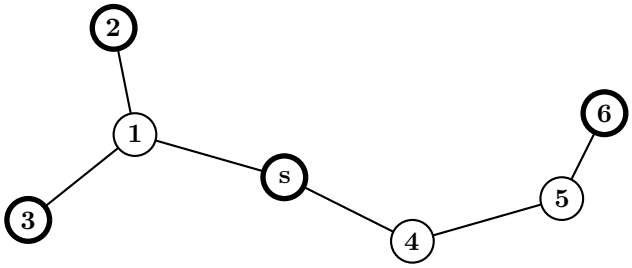

Figure 3.4: An example of key paths in a tree. All nodes other than 4 and 5 are key nodes. The key paths are the edges $\{s, 1\}$, $\{1, 2\}$ and $\{1, 3\}$, as well as the path from s to 6, including 4 and 5.

local search heuristic trying to enhance the solution on the current level. Next to the coarsening, this is the most crucial part of the algorithm, as simple refinement of the coarse solution would only rarely lead to a good overall solution directly. Therefore we implement a variant of the *Key Path Improvement* heuristic, as explained in [34, 35], to be executed after each refinement iteration.

The Key Path Improvement heuristic

The Key Path Improvement heuristic (KPI) originates from the fact that any solution of the Steiner Problem (and, therefore, also the RDCSTP) can be viewed in terms of its *key nodes*. A key node in this context is a node which is either a terminal (or the source), or a Steiner node with a degree of at least 3. *Key paths* are then all paths connecting two key nodes, without including a third one. They therefore consist of two key nodes at the end points, and an arbitrary number (possibly 0) of Steiner nodes of degree 2 in between. Figure 3.4 contains an example.

With these basic definitions in mind, the KPI now consists of first finding all key paths contained in a tree, and then for each of them removing it from the tree and reconnecting the two resulting components with each other as cheaply as possible. The delay constraint of course has to be minded here, too.

Algorithm 3.7 gives an overview of the approach. Note that it is possible that no valid connection can be found after a key path has been removed on higher levels – most notably, when a virtual edge is removed. In these cases, we just revert to the previous state.

Also note that the improvement function is called only once per level. In a typical local search heuristic the improvement would be executed iteratively until a local optimum is reached. However, experimental results showed that this would lead to worse results than obtained by the variant used here. Apparently, the performance loss due to the additional time spent in improvement outweighs the possibly better solution quality in the short term. It could also be the case that such heavily optimized solutions in general present worse starting points for subsequent iterations of the Multilevel Refinement heuristic.

The method by which cycles are detected and removed when adding the new path is outlined in Figure 3.5. Before the path is added, the nodes of the two components are marked. Then, all visited nodes in the path are marked. Once a new edge is added to the tree the algorithm

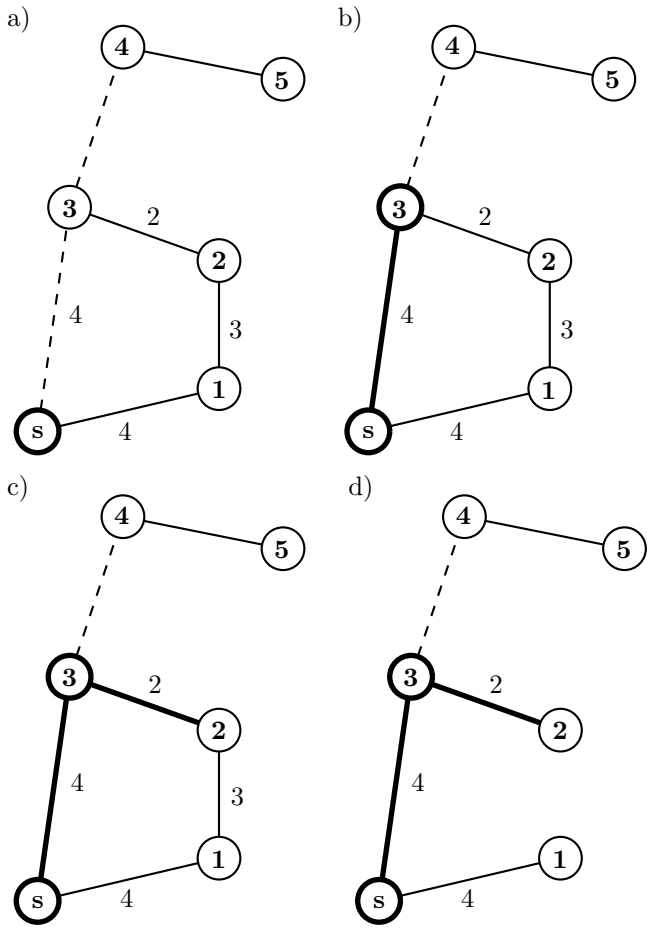

Figure 3.5: An example of detecting and removing cycles. The dashed path should be added to the tree. Visited nodes in the path are marked by a thicker circle (there is no distinction made here between terminals and Steiner nodes). Likewise, edges that were added or accepted by the algorithm are drawn heavier. The numbers next to edges are their delays, edge costs are omitted.

Algorithm 3.7: `keyPathImprovement()`

Purpose: Improves the solution tree on the current level.

```
1  keypaths ← all keypaths in T, sorted descending by cost;
2  for path ∈ keypaths do
3      // We have to be sure the path was not destroyed in a previous iteration.
4      if path is still a key path in T then
5          u ← the endpoint of path which is farther from s;
6          maxCost ← C*(path);
7          E_T ← E_T \ path;
8          comp2 ← all nodes that can be reached from u in T;
9          p2 ← getComponentSCP(comp2, maxCost);
10         // On levels above 0, there might not be a valid connection.
11         if p2 exists then
12             E_T ← E_T ∪ p2;
13             detect and remove created cycles;
14         else
15             E_T ← E_T ∪ path;
16         end if
17     end if
18 end for
```

checks whether the edge's end node (the one farther from the source) already was in the tree. If he already was in the same component as the source, a cycle has been created. The same applies when a node from the other component is reached, if it is not the first one. (This might be the case when the first node from the other component that was reached had too high delays to other nodes in the component.)

When such a cycle is detected, the algorithm then backtracks from the newly reached node along the edges it was previously connected to, adding up the delays. Once the new path to the node would increase the delay from the source to the node, or once a node already marked as visited in the path is reached, the last edge that was looked at is removed and the cycle thereby resolved.

In Figure 3.5, a) shows the initial situation, the dashed path should be added to the tree to re-connect the source to the separate component at the top. In b), when the first edge $\{s, 3\}$ is added, the algorithm detects that 3 was already part of the source component and that a cycle has therefore been created. In c), it backtracks along the edge $\{3, 2\}$ which previously connected 3. It determines that the new delay for 2 would be lower than the previous delay ($6 < 7$). Edge $\{3, 2\}$ is therefore kept in the tree and the algorithm backtracks further. Inspecting edge $\{2, 1\}$, it detects that the new route via 3 and 2 would increase the delay to node 1. Therefore, edge $\{2, 1\}$ is removed, resolving the cycle. If a cycle is created in the other component the process is analogous.

The *getComponentSCP* function used in the above explanation of the Key Path Improve-

ment algorithm denotes a function which computes the shortest path between two components while minding the delay bound. This is a slightly modified version of the so-called *Shortest Constrained Path problem*, which is known to be $\mathcal{NP}$-hard. Luckily, [36] describes a pseudo-polynomial algorithm for it which could be adapted to our purposes here. The detailed implementation will be explained in Section 4.4.

Note that we pass the removed path's cost to the function. This is done to optimize the algorithm's performance by only considering the paths that would improve the solution.

3.6 Asymptotic runtime

Before executing practical tests for obtaining empirical performance data, we are also interested in the theoretical asymptotic runtime of a single iteration of the algorithm. We call the number of nodes $N = |V|$ and the number of edges $M = |E|$. For (nearly) complete graphs, $M = \mathcal{O}\left(N^2\right)$ will hold, which is the assumption we make in the following analysis. However, in many practical problem instances, the number of edges per node can be more seen as constant, $M = \mathcal{O}\left(N\right)$, or increasing logarithmically.

The overall runtime of one iteration is of course the sum of the time needed for coarsening, solving the problem on the highest level, and refining. We label these times as $T_O = T_C + T_S + T_R$.

For coarsening (cf. Algorithm 3.2), the number of nodes will approximately be halfed on every level. Therefore, the maximum level $L = \mathcal{O}\left(\log N\right)$. In each level, we first retrieve all edges in a sorted list, which has runtime $\mathcal{O}\left(M_i \log M_i\right)$ (for sorting), where M_i is the number of edges on level i.

Then we go through the list, merging $\mathcal{O}\left(N_i\right)$ times, where each of these merges takes (for a complete graph) $\mathcal{O}\left(N_i\right)$, with $N_i \approx N2^{-i}$ being the number of nodes on level i. Since $M_i = \mathcal{O}\left(N_i^2\right)$, the whole runtime of each level of the coarsening phase will be $\mathcal{O}\left(N_i^2 \log N_i + N_i^2\right) = \mathcal{O}\left(N_i^2 \log N_i\right)$. The level i goes from 0 to $L = \mathcal{O}\left(\log N\right)$, resulting in the following term for the overall runtime of the coarsening phase T_C:

$$\begin{aligned}
\sum_{i=0}^{\log N} N^2 2^{-2i} \log\left(N^2 2^{-2i}\right) = \\
N^2 \sum_{i=0}^{\log N} \left(\frac{1}{4}\right)^i \left(2\log N - 2i\log 2\right) = \\
2N^2 \log N \left(1 + \frac{1}{4} + \frac{1}{16} + \cdots\right) - 2N^2 \log 2 \left(\frac{1}{4} + \frac{1}{8} + \cdots\right) = \\
2N^2 \log \frac{N}{2} \mathcal{O}\left(1\right) = \mathcal{O}\left(N^2 \log N\right) \\
T_C = \mathcal{O}\left(N^2 \log N\right)
\end{aligned}$$

For the runtime of solving the problem on the highest level (cf. Algorithm 3.5), the worst case upper bound would be when nearly all nodes are still present on the highest level, and nearly

all terminals need to be connected to the source via long virtual edges. The runtime would then be $T_S = \mathcal{O}\left(N^2\right)$. As even this crude approximation is below the runtime of the coarsening phase, we do not need to search for a better upper bound.

More interesting here is the refinement phase (cf. Algorithm 3.6). There are of course the same number of levels as in the coarsening phase, i.e., $\mathcal{O}\left(\log N\right)$. As we also know from the coarsening phase, the simple refinement (without improvement) takes $\mathcal{O}\left(N^2\right)$ overall (since there is no sorting involved). Calculating the runtime of the improvement phase (cf. Algorithm 3.7) is more complicated. Obtaining all key paths takes $\mathcal{O}\left(N_i\right)$, as there are $\mathcal{O}\left(N_i\right)$ key paths in the tree. The most expensive operation in the improvement heuristic is finding the shortest constrained path (SCP) between two nodes, the exact variant of which takes $\mathcal{O}\left(BM_i\right)$, where B is the delay bound. As this is done for each of the $\mathcal{O}\left(N_i\right)$ key paths, we get:

$$
\begin{aligned}
N_i \approx N2^{-i}, \quad M_i \approx N_i^2 \\
\sum_{i=0}^{\log N} BN^3 2^{-3i} = \\
BN^3 \sum_{i=0}^{\log N} \left(\frac{1}{8}\right)^i = \\
BN^3 \frac{1 - \left(\frac{1}{8}\right)^{\log N+1}}{1 - \frac{1}{8}} = \\
BN^3 \left(\frac{8}{7} - \frac{8}{7 \cdot 8^{\log N+1}}\right) = \mathcal{O}\left(BN^3\right) \\
T_R = \mathcal{O}\left(BN^3\right)
\end{aligned}
$$

Therefore, under the premises stated at the start of the section, we see that most of the runtime in the asymptotic case will be spent in the refinement phase, resulting in an overall runtime of

$$T_O = \mathcal{O}\left(N^2 \log N + N^2 + BN^3\right) = \mathcal{O}\left(BN^3\right) .$$

This runtime is at least a vast improvement compared to exact algorithms for an $\mathcal{NP}$-hard problem, as it is only polynomial, not exponential, in the number of nodes. However, the fact that the runtime will depend on the delay bound B is worrying, as simple scaling of all delays by some factor should normally not influence the algorithm. This is a problem of the exact method for finding an SCP between two components, which loops over all possible delays and finds the shortest path for each of them.

To mitigate these problems a different, heuristic algorithm was implemented for finding SCPs. This was a simple implementation of Dijkstra's algorithm [37], adapted to include delay bounds and to be suitable for computing the SCP between whole components. It will be described in detail in Section 4.4. The runtime for this algorithm is the same as for Dijkstra's algorithm itself (implemented with d-ary heaps), namely $\mathcal{O}\left(M \log N\right)$ [38]. Using this algorithm instead of the exact variant therefore leads to an overall runtime of

$$T'_O = \mathcal{O}\left(N^2 \log N + N^2 + M \log N\right) = \mathcal{O}\left(N^2 \log N\right) \ .$$

Here, the runtime of the refinement phase no longer dominates the overall runtime, which is now also determined by the coarsening phase's runtime.

CHAPTER 4

Implementation

This chapter discusses some of the details of our implementation, too specific to be mentioned in the discussion of the general algorithm. Especially, particular data structures that were introduced, detailed implementations of some algorithms and parameters introduced to the algorithm are explained.

The algorithm was implemented in a program using an existing C++ framework for the RDCSTP, provided by [39]. This framework already included the preprocessing described in [8] for the Rooted Delay-Constrained Minimum Spanning Tree Problem. Also included were some of the preprocessing techniques described in [9].

An existing construction heuristic was then used to create the initial solution for improvements. Although it would have easily been possible to use the Multilevel algorithm both for constructing and subsequent improvement, having an existing solution to compare against was considered favorable.

The construction heuristic used had a simple approach, iteratively adding shortest constrained paths to all terminals, as first introduced by [4]. Despite of the simple approach, this still turned out to find an optimal result in several cases for small instances.

4.1 Additional data structures

In the form described in Section 3, the algorithm would store too little information during coarsening to be able to reliably find valid solutions on higher levels. Especially, copying unmodified edges to higher levels while removing those between merged nodes from the tree would result in much too low delays for paths on higher levels, letting too many solutions seem valid.

The `dmax` property

There are actually two different problems with this naive approach. The first problem is that reaching a node on a higher level within the delay bound is not enough. It only means that you reach the nearest of the contained nodes within the delay bound, not all of them. We therefore

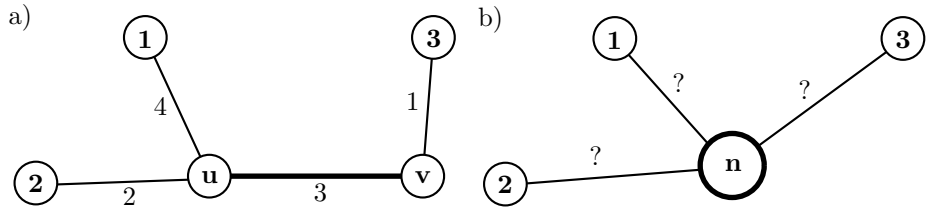

Figure 4.1: An example illustrating the additionalDelays data structure. If the edge $\{u, v\}$ in a) is merged – what should the delays look like in the resulting coarsened graph in b)? (The edges are only labelled with their delays in this example, as the edge costs are irrelevant here.)

needed a measure of the extra delay within a node we had to consider when connecting it to the solution tree.

To solve this problem, we introduced a `dmax` property for all nodes in the graph. This property is only used for nodes on higher levels, which already contain several other nodes due to merging, and defaults to 0 otherwise. It stores the maximum delay on the path between any two terminals contained in the node. Since Steiner nodes do not have to be connected to in the tree if they would be leaves, they are not taken into account here.

However, on the whole these were just approximations of the real inner structure of the node. In general, a single value cannot represent all possible configurations in which a node can be added to a tree and much more complex data structures would have been needed. We therefore opted for keeping this as a heuristic measure and accepting that solutions for higher levels could in reality sometimes slightly violate the delay bound. Along with other performance-related changes, discussed in Section 4.4, this resulted in the necessity of executing a repair algorithm during refinement.

A value for this property is computed whenever a new merged node is created, based on the `dmax` values of the merged nodes. A practical example will be given later in this chapter.

This property is then always used when determining whether a connection from the source to a node forms a valid part of a solution. In practice, this means that the delay bound for an individual node is permanently decreased by the value of its `dmax` property.

The additionalDelays data structure

The second problem with the naive approach is that the delays of edges contained in merged nodes would be ignored when computing the delay of a path leading across such merged nodes. For illustration consider the exemplary merge operation in Figure 4.1. Once the nodes u and v are replaced by n and all edges reconnected, the question remains what delays to set for the new edges to n. If we just use the same delays as before, the delay of $\{u, v\}$ is completely disregarded, leading to wrong results when, e.g., connecting 1 and 3 via n. This would make it very likely that invalid solutions are created on higher levels, as the delays of paths are considerably underestimated.

On the other hand, if we add the delay (or half the delay) of $\{u, v\}$ to all surrounding edges, the delay to n itself will be wrong. Also, when connecting 1 to 2 via n, ignoring the delay of

Algorithm 4.1: `isFeasible()`

Purpose: Determines whether the given edge could be part of a valid solution on the current level.
Input: An edge `e`, connecting the nodes `u` and `v`.
Output: **true**, if the edge can be part of a valid solution; **false** otherwise.

```
1 pathU ← P_G(u, s);
2 predU ← node next to u in pathU;
3 pathV ← P_G(v, s);
4 predV ← node next to v in pathV;
5 delayU ← D*(pathV) + additionalDelays[v][predV][u] + u.dmax;
6 delayV ← D*(pathU) + additionalDelays[u][predU][v] + v.dmax;
7 return (min(delayU, delayV) + D(e)) ≤ B;
```

$\{u, v\}$ is really the right thing to do.

It is therefore obvious that the correct delays for the new edges would differ depending on the context in which we retrieve them. Since this cannot be done by simply setting some edge delay, we introduced the additionalDelays global data structure. This is a three-dimensional array which stores for each node and each pair of its neighbors the additional delay that will have to be added to the edge delays when connecting the pair of neighbors via the node. In our example, we would have three new entries:

```
additionalDelays[n][1][2] = 0;
additionalDelays[n][1][3] = 3;
additionalDelays[n][2][3] = 3;
```

This assumes that we are on level 1 – otherwise we would have to take into account existing entries for u and v, as will be illustrated later in this chapter. Note also that additionalDelays is of course symmetric in the second and third indices, so we would really have to add six entries. However, for the sake of simplicity we assume here, and in the rest of the paper, that setting additionalDelays`[i][j][k]` will automatically also set additionalDelays`[i][k][j]` to the same value.

This information in additionalDelays is subsequently used in all places in the algorithm where the delay of a path is computed. As an example, Algorithm 4.1 contains the detailed implementation of the *isFeasible* function introduced in Section 3.3.

As explained there, the function analyzes whether the delay from s to either of the end nodes via the checked edge lies below the delay bound. However, as can be seen this necessitates additional checking of the additionalDelays data structure and the nodes' `dmax` values. For example, the delay from s to node u, `delayU`, is the sum of the delay from s to v; plus the additional delay within v, when connecting the previous node in the lowest-delay path to s with u; plus the `dmax` value of u, as the delay bound would have to be reduced by that value. The computation for `delayV` is analogous, the function *min* simply returns the minimal value of

all its arguments. If this minimum is lower than or equal to the delay bound, the edge could still be part of a valid solution.

Memory size problems

As one can easily see, the memory size of the additionalDelays data structure necessarily is in the dimension of $\mathcal{O}\left(N^3\right)$. In practice, this becomes even worse as the additional nodes added during coarsening result in an overall factor of $(2N)^3 = 8N^3$. A naive implementation of the data structure therefore was not practical for larger instances, quickly running out of memory for instances of about 1000 nodes.

A first step to mitigate this problem was to "re-use" nodes when merging – instead of removing both merged nodes and adding a new one, we just "promoted" one of the merged nodes to the next level, adapting all its related information accordingly. When merging a terminal and a Steiner node, we thereby always kept the terminal, so this information would automatically always be correct. This re-using of nodes had no influence on the general algorithm, but made the implementation of several parts significantly easier. (For example, we also could now easily come up with a "contained" node for Algorithm 3.5 when solving the highest level.) It also helped to conserve a lot of memory for the additionalDelays data structure. However, on the other hand it necessitated "versioning" of its data, as entries could now vary according to the current level. Simply copying and storing the data structure for each level would of course almost eliminate the little decrease in memory size this approach afforded us.

In the end, we therefore resolved this problem by not using a complete three-dimensional array for all nodes, but an array containing two-dimensional arrays containing only entries for the neighbors of all nodes. Since, after preprocessing, even in complete graphs nodes were not connected to most other nodes (especially for larger instances), this resulted in a huge decrease in memory size. The versioning problem for different levels was further mitigated by using lists for each entry in the three-dimensional array, versioning each entry on its own and thereby only creating additional entries where really necessary. On the whole, this resulted in a large improvement of memory size that even allowed us to solve instances as large as 5000 nodes.

Of course, the new layout of the additionalDelays data structure required additional measures when accessing the data structure. Algorithm 4.2 shows the function used for accessing the data structure. It uses a new data structure, additionalDelaysLookup, to find the real indices used for accessing the additionalDelays data structure. Also, to exploit the symmetry of the data structure, we only store entries for half of the table and therefore need to swap the indices if supplied in the "wrong" order. Only then the list of entries for these three nodes can be obtained. This list is then searched for the relevant entry for the current level. (New entries for higher levels are always prepended to the list.)

The function `front` here returns the first element of the list, where each element has the two properties `level` and `delay`. The function `pop` removes the list's first element. If the list does not contain a relevant entry, 0 is returned. Otherwise the entry's `delay` value is returned.

Algorithm 4.2: `getAdditionalDelay()`

Purpose: Finds the additional delay to be taken into account when connecting two nodes via a third node.
Input: Three nodes `v`, `i` and `j`.
Output: The additional delay in `v` when connecting `i` and `j` via `v`.

```
1  i ← additionalDelaysLookup[u][i];
2  j ← additionalDelaysLookup[u][j];
3  if i < j then
4  |  swap i and j;
5  end if
6  list ← additionalDelays[v][i][j];
7  while list not empty and front(list).level > l do
8  |  pop(list);
9  end while
10 if list is empty then
11 |  return 0;
12 end if
13 return front(list).delay;
```

The changelog data structure

Also not mentioned in the algorithm is the changelog data structure which was used to capture the changes that were made during coarsening. While, in principle, storing each level during coarsening and then dynamically computing the differences between them during refinement would be possible, logging just all operations and then undoing them in the reverse order saves both time and memory size in practice, while also keeping that part of the algorithm considerably simpler.

As not removing nodes from the graph (as long as you make sure to not treat them as terminals anymore) during coarsening makes little difference, we only added or deleted the relevant edges. Therefore, the changelog data structure just consisted of an array containing a list of changes for each level, where each change would simply store a flag for the type of operation, adding or deleting, and the data of the edge in question.

4.2 A detailed merge example

With these new insights into the inner workings of our implementation, we can now also revisit the merge example from Section 3.3 and discuss the additional information we would need to save there. Most basically, we would of course store each adding or removal of an edge in changelog. But apart from that, we would also need to update the additionalDelays data structure and the merged node's `dmax` value accordingly.

Consider the example in Figure 4.2. When executing the merge operation depicted there,

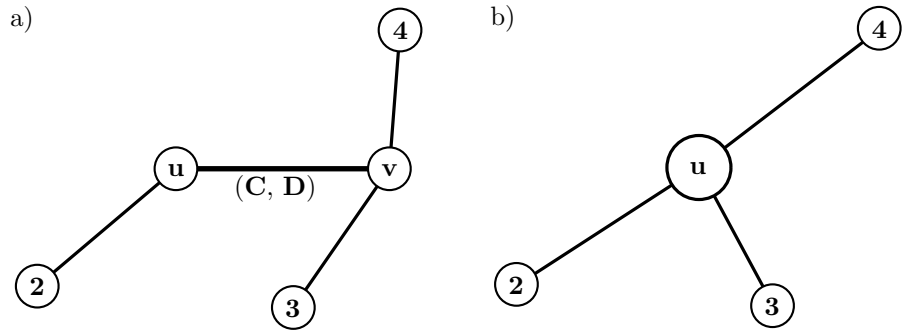

Figure 4.2: An example for merging two nodes. a) A subsection of a problem graph, with the nodes u and v being merged and infeasible or duplicated edges already removed. b) The resulting merged node (which is the old node u with altered data) and its edges.

the following code would be used to update all the necessary data in additionalDelays as well as `u.dmax`. (Note that the `dmax` values of all nodes are automatically stored at the beginning of each level, so we do not need to take care of that here.)

```
if (isTerminal(u) && isTerminal(v)) {
    u.dmax = max(u.dmax, v.dmax, D + min(u.dmax, v.dmax));
}

i2 = additionalDelaysLookup[u][2];
i3 = additionalDelaysLookup[u][3];
i4 = additionalDelaysLookup[u][4];

additionalDelays[u][i2][i3] = D + getAdditionalDelay(u, 2, v)
        + getAdditionalDelay(v, u, 3);
additionalDelays[u][i2][i4] = D + getAdditionalDelay(u, 2, v)
        + getAdditionalDelay(v, u, 4);
additionalDelays[u][i3][i4] = getAdditionalDelay(v, 3, 4);
```

The new `dmax` value of the merged node is computed in a way that ensures creating mostly valid solutions, while avoiding to limit the solution space too much. Note that the formula to use here for reliably preventing the property value from being too small in certain instances would be `D + u.dmax + v.dmax`. However, as using this formula would lead to coarsening stopping much earlier, due to the high resulting `dmax` values, this compromise between correctness and flexibility was adopted.

For computing the new additionalDelays entries from the existing ones, there are simply two cases. Either the two neighboring nodes are connected to the new node with edges that belonged to the same previous node (u or v) – in which case we just copy the corresponding entry for the previous node, as it is done for the delay from 3 to 4 in the example. (If both nodes would have

been connected to u, we would not have had to do anything.) Or the edges belonged to different nodes (as they did in the two other cases in the example), so we have to add both the delay of the merged edge and the two entries for the respective additional delays in u and v to obtain the correct new additionalDelays entry.

4.3 Parameters

The algorithm was originally built with a number of parameters, to be able to flexibly adapt some of its aspects and see what settings work best. The following parameters were introduced:

randBoost
: As already mentioned in Section 3.3, this parameter controls the influence of a random factor when choosing the edges to merge during coarsening. When set to 0, the complete algorithm is deterministic, with the random factor removed. Values greater than 0 are used as the random factor's variance, leading to more randomized edge selection with increasing randBoost.

treeBoost
: This parameter, too, was already mentioned while discussing edge selection during coarsening. It controls the influence of the previous solution on edge selection, as the scores of all edges that are present in the current tree are divided by this parameter. Therefore, the previous solution will have no influence when this is set to 1, values greater than 1 will make the selection of edges from the previous solution more likely.

twoTermBoost
: This parameter is one of the two parameters that are used to boost edges based on the types of nodes they connect in *edgeTypeBoost(*e*)*. When positive, the scores of all edges that do not connect two terminals are multiplied by this value, resulting in edges between two terminals being merged earlier (unless the parameter would be lower than or equal to 1). When negative, on the other hand, edges not connecting two Steiner nodes are multiplied by the absolute value of the parameter. When set to 0, this parameter is not used.

mixedEdgeBoost
: Similar to twoTermBoost, a positive value for this parameter means that all edges connecting either two terminals or two Steiner nodes are multiplied by the value. This means that edges connecting a terminal and a Steiner node would be merged sooner. The effect of negative values is exactly the same as for twoTermBoost, and the parameter is also not used if set to 0.

scpMode
: As described in Section 3.6, we implemented two different algorithms for solving the SCP problem occuring in the KPI heuristic, one exact and one heuristic algorithm. The two variants are discussed in detail in Section 4.4. This parameter controls which of these two functions is used.

minKPCost

This parameter stands for the "minimum key path cost" and can be used to restrict the amount of key paths being looked at in the KPI runs. Concretely, all key paths with a cost lower than this treshold are just carried over and never replaced. The parameter is specified in relation to the highest cost among all key paths. Therefore, a value of 0 means that all key paths will be looked at, while with a setting of 1, only the most expensive key path might be replaced. Values greater than 1 would effectively disable the improvement phase of the algorithm.

In Section 5.1 these parameters were then evaluated to see which settings would provide the best results, and what dynamics could be observed when varying instance size, delay bounds or other problem characteristics. Note that an examination of the exponents in the score formula from Section 3.3 was dropped due to time constraints. They were in our tests therefore always set to a value of 1.

4.4 Shortest Constrained Path algorithms

We now discuss the implementation details for the two mentioned algorithms for the Shortest Constrained Path problem.

Exact algorithm

As mentioned in Section 3.5, the pseudo-polynomial exact algorithm for solving the SCP problem for two components employs a dynamic programming approach described by Gouveia et al. in [36], for finding the shortest constrained path between two nodes. Its adapted version is shown in detail in Algorithm 4.3. Here, the modified cost function

$$\mathcal{C}'(e) = \begin{cases} 0 & \text{if } e \in T \\ \mathcal{C}(e) & \text{otherwise} \end{cases}$$

is used, which ignores the cost of all edges that are already contained in the tree. The function `getNeighbors` returns all nodes that are connected to the given node in the tree.

The basic idea is to use Dijkstra's algorithm [37] for finding the shortest path between two nodes, but restricting the paths to those up to a certain delay. Starting with this delay bound at 0 and gradually increasing it to the delay bound given by the problem we automatically get for each node the cheapest way of reaching it for each delay bound. Therefore, we additionally only have to remember the cheapest way we reached any node in the other component (starting from the source node and disregarding the cost of all edges contained in the tree) and eventually backtrack the path to the source from there.

Note that the check for a valid connection on line 23 of the algorithm only checks for the height in the component, not taking the possible additional delay in the reached node into account. This was done to avoid the much more expensive complete check, which would need to completely traverse the whole component each time, but naturally leads to invalid solutions

Algorithm 4.3: `getComponentSCP()`

Purpose: Finds the delay-constrained shortest path between two separate components of T in G.

Input: `comp2`, the component that does not contain s; and `maxCost`, lower than which the connection's cost should be.

Output: The cheapest valid path connecting the two components, if one exists.

```
1  // cost, pre and preDelay are 2-dimensional tables containing ∞ for all entries
2  // minC is a one-dimensional table containing maxCost for all entries
3  // reachable is an array of (empty) node sets
4  for i ∈ comp2 do
5  |  height[i] ← maximum delay to any other node in comp2;
6  end for
7  bound ← B − min_i(height[i]);
8  vc ← maxCost;
9  insert s into reachable[0];
10 minC[s] ← 0;
11 for b ∈ {0, 1, ..., bound − 1} do
12 |  for i ∈ reachable[b] do
13 |  |  if cost[i][b] < vc then
14 |  |  |  minC[i] ← min(minC[i], cost[i][b]);
15 |  |  |  for j ∈ getNeighbors(i) do
16 |  |  |  |  b_2 ← b+D({i, j})+getAdditionalDelay(i, pre[i][b], j);
17 |  |  |  |  if b_2 ≤ bound then
18 |  |  |  |  |  c_2 ← cost[i][b] + C'({i, j});
19 |  |  |  |  |  if c_2 < min(minC[j], cost[j][b_2]) and c_2 < vc then
20 |  |  |  |  |  |  insert j into reachable[b_2];
21 |  |  |  |  |  |  cost[j][b_2] ← c_2;
22 |  |  |  |  |  |  pre[j][b_2] ← i; preDelay[j][b_2] ← b;
23 |  |  |  |  |  |  if j ∈ comp2 and b_2 ≤ (B − height[j]) then
24 |  |  |  |  |  |  |  v ← j; vc ← c_2;
25 |  |  |  |  |  |  end if
26 |  |  |  |  |  end if
27 |  |  |  |  end if
28 |  |  |  end for
29 |  |  end if
30 |  end for
31 end for
32 if vc < maxCost then
33 |  return path from s to v (determined by pre and preDelay);
34 end if
```

being sometimes created on levels above 0. Therefore, this was another reason for including a repair mechanism in the refinement phase.

Heuristic algorithm

The heuristic variant of the algorithm, shown in Algorithm 4.4, takes a simpler approach which is just a slight adaption of Dijkstra's algorithm for finding the shortest path between two nodes [37]. As can be seen it first marks all nodes as reachable with cost ∞ and only the source node s to be reachable with cost 0. Like in the exact algorithm, the modified edge cost function is used to ignore the costs of edges that are already part of the solution tree.

Then, similar to the normal variant of Dijkstra's algorithm, the costs of all neighbors of the source node are updated according to the cost of the edges that connect them, which is done iteratively for all reachable nodes, by ascending costs. The only difference to Dijkstra's original algorithm is that we make sure that a node can be reached within the delay bound before updating its cost. Also, we have again the same success criterion as in the exact algorithm, of reaching the other component of the tree while fulfilling the relaxed condition on the path delay plus the delay height from the reached node. In this case, however, we can now instantly return from the function as it is guaranteed that we will not be able to find a cheaper path to the component with this algorithm.

Algorithm 4.4: `getComponentSCPHeuristic()`

Purpose: Finds a short delay-constrained path between two separate components of T in G.

Input: `comp2`, the component that does not contain s; and `maxCost`, lower than which the connection's cost should be.

Output: A cheap valid path connecting the two components, if one exists.

```
1  for i ∈ V do
2      cost[i] ← maxCost;
3      delay[i] ← ∞;
4      pre[i] ← i;
5      if i ∈ comp2 then
6          height[i] ← maximum delay to any other node in comp2;
7      end if
8  end for
9  cost[s] ← 0;
10 delay[s] ← 0;
11 bound ← B − min_i(height[i]);
12 queue ← {s};
13 while queue ≠ ∅ do
14     i ← node with lowest cost entry from queue;
15     if i ∈ comp2 and delay[i] ≤ (B − height[i]) then
16         return path from s to i (determined by pre);
17     end if
18     remove i from queue;
19     for j ∈ getNeighbors(i) do
20         d ← delay[i] + D({i, j}) + getAdditionalDelay(i, pre[i], j);
21         if d ≤ bound and cost[i] + C'({i, j}) < cost[j] then
22             cost[j] ← cost[i] + C'({i, j});
23             delay[j] ← d;
24             pre[j] ← i;
25             if j ∉ queue then
26                 insert j into queue;
27             end if
28         end if
29     end for
30 end while
```

CHAPTER 5

Benchmarks and comparison

To evaluate the finished program, numerous tests were carried out, varying the given instance sizes and other characteristics. For each test, a single core of a Xeon E5540 processor with 2.53 GHz was used, where each core had up to 3 GB RAM at its disposal. The test instances were randomly created complete graphs with 100, 500, 1000 and 5000 nodes. There were 30 instances for each of these sizes. The costs and delays both varied between 1 and 99, inclusive, and were independently obtained from a uniform distribution. Some of the instances can be found at [40].

For determining the set of terminals, a variable R was introduced, specifying the fraction of nodes that are terminals. For an instance with N nodes, the first $N \cdot R$ nodes in the graph were then marked as terminals.

5.1 Evaluating parameters

In the first test runs carried out, we evaluated the parameters described in Section 4.3 to find the best settings for subsequent tests. The following symbols are used in the tables representing results in this chapter:

R	Fraction of nodes which are terminals
N	Number of nodes (specifying a certain set of instances)
B	Delay bound used
*	Marks the column for the tested parameter
imp.	Relative improvement compared to the solution generated by the construction heuristic (in percent)
σ	Mean standard deviation of the imp. values
n	Number of iterations of the algorithm which could be executed

The result values “imp.”, “σ” and “n” are the (arithmetic) means over all 30 test instances for the specified test group.

$N = 500$		$R = 0.1$			$R = 0.3$			$R = 0.5$			$R = 0.7$			$R = 0.9$		
B	*	imp.	σ	n	imp.	σ	n	imp.	σ	n	imp.	σ	n	imp.	σ	n
6	0.0	11.53	0.00	3083.4	15.97	0.00	1258.3	16.98	0.02	839.4	17.86	0.01	656.4	18.19	0.02	553.5
	0.5	14.57	0.52	2883.4	16.09	0.51	1224.1	16.98	0.52	820.3	17.98	0.45	644.3	18.35	0.42	545.3
	1.0	14.80	0.46	2953.2	16.13	0.51	1226.4	16.94	0.47	816.5	17.98	0.54	636.5	**18.42**	0.44	539.4
	2.5	**15.10**	0.52	3138.6	**16.32**	0.56	1241.5	**17.04**	0.44	810.1	**17.98**	0.47	625.9	18.32	0.46	528.6
16	0.0	15.96	0.03	756.4	16.69	0.03	385.8	17.04	0.02	264.9	17.26	0.02	206.3	17.94	0.03	172.8
	0.5	**16.07**	0.88	694.5	**16.82**	0.90	369.4	17.11	0.91	258.0	17.66	0.88	203.2	17.93	0.93	171.1
	1.0	16.07	0.91	681.9	16.69	0.95	365.2	**17.15**	0.93	254.3	**17.74**	0.91	200.7	**18.05**	0.86	169.7
	2.5	15.87	1.03	670.8	16.51	0.99	356.4	16.94	0.92	251.2	17.41	0.82	198.1	17.98	0.86	168.8
32	0.0	15.74	0.00	494.8	16.29	0.06	255.6	**17.21**	0.02	171.5	17.29	0.00	133.2	**17.69**	0.06	109.8
	0.5	**15.89**	1.07	455.0	**16.39**	1.09	243.5	17.12	0.97	166.8	17.50	0.91	129.2	17.56	0.91	107.4
	1.0	15.78	1.16	442.8	16.11	0.96	239.0	16.95	0.92	163.4	**17.51**	0.95	127.4	17.46	0.87	106.3
	2.5	15.45	1.21	425.2	15.66	1.04	229.9	16.61	1.12	159.5	16.94	1.00	124.5	17.01	0.96	104.2
50	0.0	**14.34**	0.06	399.9	15.32	0.02	205.8	15.79	0.00	136.9	**16.14**	0.04	105.1	**16.36**	0.04	86.0
	0.5	14.28	1.04	367.1	**15.68**	0.96	197.5	**15.87**	0.94	134.0	16.01	0.90	102.3	16.05	0.81	85.2
	1.0	14.16	1.06	356.4	15.61	1.02	192.9	15.63	0.96	131.6	15.78	0.98	101.4	15.94	0.91	83.7
	2.5	13.58	1.20	341.2	14.81	1.04	185.8	15.01	1.18	126.9	15.12	0.99	99.0	15.26	1.00	81.1

Table 5.1: Results for different values of the randBoost parameter and 500 node instances.

$N = 1000$		$R = 0.1$			$R = 0.3$			$R = 0.5$			$R = 0.7$			$R = 0.9$		
B	*	imp.	σ	n	imp.	σ	n	imp.	σ	n	imp.	σ	n	imp.	σ	n
6	0.0	12.30	0.00	1229.6	16.21	0.08	576.2	17.63	0.07	387.6	19.01	0.03	305.6	**20.22**	0.02	269.2
	0.5	**12.71**	0.84	1177.6	16.23	0.66	565.3	**17.81**	0.60	394.2	**19.07**	0.55	307.4	20.08	0.52	266.0
	1.0	12.67	0.82	1175.1	**16.23**	0.65	563.6	17.77	0.61	386.4	18.99	0.62	305.7	20.10	0.56	270.8
	2.5	12.66	0.85	1188.1	16.12	0.67	561.0	17.71	0.58	382.7	18.92	0.55	299.8	19.95	0.52	265.5
16	0.0	**14.91**	0.04	385.5	**17.39**	0.07	186.0	17.71	0.03	106.0	18.46	0.05	81.5	18.65	0.08	73.0
	0.5	14.40	1.07	369.4	17.15	0.88	179.9	17.89	0.80	110.4	**18.57**	0.79	83.9	**18.85**	0.74	69.9
	1.0	14.44	1.12	360.0	16.97	0.88	164.3	**17.95**	0.86	108.7	18.52	0.74	83.2	18.76	0.75	69.3
	2.5	13.50	1.14	339.0	16.27	0.90	157.6	17.38	0.90	105.2	18.07	0.75	80.6	18.46	0.77	72.2
32	0.0	**14.59**	0.01	223.0	**15.00**	0.05	95.9	**15.82**	0.08	61.5	**16.11**	0.04	46.1	**16.29**	0.08	39.3
	0.5	14.49	1.07	213.3	14.98	0.99	95.8	15.51	0.84	61.3	15.88	0.75	46.7	16.21	0.82	38.2
	1.0	14.25	1.10	205.5	14.68	1.03	93.8	15.22	0.88	60.5	15.73	0.89	48.3	15.91	0.81	37.3
	2.5	13.18	1.19	195.0	13.70	1.11	90.5	14.37	0.94	59.3	14.62	0.90	44.8	14.98	0.87	38.2

Table 5.2: Results for different values of the randBoost parameter and 1000 node instances.

In the tests for each parameter all other parameters were set to default values (influenced by preliminary test runs). These were:

randBoost	0.2
treeBoost	∞
twoTermBoost	0
mixedEdgeBoost	0
scpMode	0
minKPCost	0

The time limits used were 180 seconds for 500 node instances and 720 seconds for 1000 node instances. We executed ten test runs for each instance and each tested setting.

$N = 500$		$R = 0.1$			$R = 0.3$			$R = 0.5$			$R = 0.7$			$R = 0.9$		
B	*	imp.	σ	n	imp.	σ	n	imp.	σ	n	imp.	σ	n	imp.	σ	n
6	0	13.77	0.61	2628.0	15.22	0.64	1137.2	15.96	0.53	777.5	17.00	0.49	623.5	17.34	0.47	537.0
	1	13.87	0.48	2672.6	15.54	0.46	1147.5	16.57	0.49	786.9	17.50	0.47	633.5	18.02	0.42	546.2
	4	14.03	0.56	2764.7	15.85	0.55	1183.0	16.83	0.53	809.1	17.70	0.48	640.8	18.36	0.43	545.7
	1048576	**14.44**	0.53	2848.5	**16.04**	0.51	1214.8	**16.92**	0.46	822.6	**17.97**	0.45	644.7	**18.43**	0.42	546.4
16	0	14.88	1.18	624.5	15.18	0.98	355.8	15.38	0.94	256.7	15.41	0.97	200.5	15.74	0.92	172.6
	1	15.32	1.00	670.3	16.08	0.95	363.1	16.16	0.92	258.3	16.62	0.92	205.6	17.10	0.90	176.1
	4	15.70	0.87	676.5	16.61	0.85	369.1	16.99	0.87	259.3	17.35	0.84	206.0	17.67	0.82	174.7
	1048576	**16.05**	0.83	690.6	**16.83**	0.82	372.0	**17.12**	0.92	260.7	**17.54**	0.91	205.8	**17.80**	0.84	174.9
32	0	14.14	1.34	407.1	13.90	1.19	240.1	14.13	1.10	170.0	14.48	1.09	133.6	14.23	1.05	112.8
	1	14.89	0.97	450.0	15.59	1.09	246.2	15.89	0.86	171.6	16.51	0.87	135.1	16.75	0.87	113.4
	4	15.37	1.13	451.4	15.91	0.99	246.1	16.60	0.92	170.0	17.31	0.93	133.0	17.47	0.94	111.8
	1048576	**16.00**	1.04	456.6	**16.41**	0.90	248.1	**17.03**	0.91	171.0	**17.67**	0.93	133.2	**17.68**	0.83	111.3
50	0	11.54	1.35	345.3	12.20	1.25	195.7	11.92	1.18	135.6	11.88	1.10	107.0	12.22	0.99	89.7
	1	12.61	1.21	367.3	13.83	1.14	199.6	14.20	0.98	138.5	14.50	0.99	107.8	14.81	0.85	90.2
	4	13.47	1.10	367.2	15.12	1.04	200.5	15.33	0.99	138.4	15.59	0.85	107.5	16.03	0.88	89.2
	1048576	**14.18**	0.91	373.8	**15.68**	0.98	203.6	**15.94**	0.95	138.8	**16.02**	0.87	106.7	**16.15**	0.83	88.1

Table 5.3: Results for different values of the treeBoost parameter and 500 node instances.

$N = 1000$		$R = 0.1$			$R = 0.3$			$R = 0.5$			$R = 0.7$			$R = 0.9$		
B	*	imp.	σ	n	imp.	σ	n	imp.	σ	n	imp.	σ	n	imp.	σ	n
6	0	10.78	1.08	1077.2	14.20	0.75	507.5	16.00	0.64	359.8	17.48	0.64	289.9	18.63	0.55	267.6
	1	11.40	0.89	1081.0	15.08	0.68	520.7	17.05	0.58	364.6	18.30	0.51	297.3	19.36	0.50	266.5
	4	11.87	0.87	1131.5	15.70	0.67	560.7	17.48	0.52	396.4	18.84	0.55	311.6	19.96	0.51	272.5
	1048576	**12.59**	0.82	1182.2	**16.31**	0.70	638.5	**17.82**	0.58	445.6	**19.04**	0.52	353.7	**20.13**	0.46	304.3
16	0	13.12	1.11	335.3	15.16	0.94	152.0	15.70	0.91	102.3	16.13	0.87	80.3	16.20	0.78	68.7
	1	13.83	1.04	360.7	16.24	0.88	152.2	17.06	0.82	102.1	17.84	0.70	78.4	18.12	0.72	69.9
	4	14.11	1.00	356.0	16.79	0.93	166.3	17.73	0.77	110.5	18.49	0.73	84.5	18.79	0.69	70.6
	1048576	**14.59**	1.00	430.9	**17.14**	0.84	195.3	**18.16**	0.76	129.6	**18.75**	0.76	97.6	**18.93**	0.71	81.2
32	0	11.67	1.36	193.6	12.10	1.06	88.1	12.61	0.92	59.1	12.88	0.86	45.6	13.06	0.89	38.3
	1	12.48	1.18	197.2	13.48	1.00	88.6	14.40	0.87	58.7	14.99	0.80	46.6	15.54	0.83	38.3
	4	13.64	1.11	213.5	14.49	0.94	96.6	15.33	0.83	64.0	15.70	0.77	47.9	16.31	0.77	39.7
	1048576	**14.70**	0.98	265.6	**15.10**	0.84	119.1	**15.78**	0.84	77.7	**16.07**	0.75	56.7	**16.46**	0.77	45.3

Table 5.4: Results for different values of the treeBoost parameter and 1000 node instances.

randBoost

Table 5.1 and Table 5.2 contain the results obtained for various values of the randBoost parameter. As can be seen, a small random factor generally improves the algorithm's results, the benefit of exploring more possible solutions seems to easily outweigh the benefit of reliably finding a local optimum (in which a deterministic algorithm will then of course be stuck).

The table shows also that the best results are obtained by higher random factors for smaller instances than for larger ones. This, however, can be explained with the much larger number of iterations that could be executed on these instances, which would naturally favor higher randomness. A general principle for more randomness in smaller instances does not seem to hold when examining the differences between different terminal node ratios at the same instance size.

treeBoost

In Table 5.3 and Table 5.4, the results for different values of the treeBoost parameter are listed. Contrary to our expectations, the highest value for the parameter (equivalent to the edges of the

$N = 500$		$R = 0.1$			$R = 0.3$			$R = 0.5$			$R = 0.7$			$R = 0.9$		
B	*	imp.	σ	n	imp.	σ	n	imp.	σ	n	imp.	σ	n	imp.	σ	n
6	-2	14.49	0.50	2871.3	16.13	0.54	1224.2	16.95	0.48	824.8	**18.00**	0.47	647.3	18.39	0.43	545.6
	0	14.44	0.53	2860.0	16.02	0.56	1214.7	16.91	0.44	822.1	17.96	0.46	645.8	**18.48**	0.45	546.3
	2	14.49	0.56	2948.5	**16.15**	0.57	1263.4	17.04	0.46	852.4	17.94	0.43	661.5	18.41	0.43	552.0
	3	**14.57**	0.46	2994.0	16.15	0.52	1296.6	**17.04**	0.49	870.8	18.00	0.47	672.6	18.35	0.40	555.5
16	-2	**16.27**	0.92	705.5	16.98	0.91	384.0	17.12	0.92	267.8	17.55	0.89	208.7	17.81	0.79	174.6
	0	16.06	0.89	695.0	16.89	0.96	373.2	17.14	0.86	260.5	**17.61**	0.88	205.6	17.82	0.80	174.1
	2	16.05	0.82	713.6	16.92	0.90	388.5	17.14	0.94	270.4	17.54	0.84	210.7	17.86	0.87	175.9
	3	16.14	0.95	716.7	**16.98**	0.87	394.0	**17.14**	0.89	274.0	17.51	0.82	212.2	**17.87**	0.92	176.0
32	-2	**16.19**	1.02	464.2	16.54	0.95	257.0	**17.12**	0.85	174.6	17.48	0.80	134.1	**17.71**	0.88	110.8
	0	15.91	1.01	459.2	16.44	1.06	249.1	17.07	0.93	170.3	**17.66**	0.92	132.3	17.66	0.81	117.4
	2	16.06	1.02	475.5	**16.57**	0.89	262.3	17.03	0.84	179.4	17.56	0.84	137.3	17.66	0.84	112.6
	3	16.15	0.98	475.0	16.36	1.02	264.3	17.01	0.89	180.0	17.53	0.84	138.2	17.58	0.83	112.2
50	-2	**14.31**	1.01	378.9	**15.98**	0.96	209.4	15.83	0.93	141.6	15.92	0.86	108.2	16.17	0.83	87.5
	0	14.14	1.05	372.6	15.69	0.96	202.4	15.87	0.97	138.2	**15.99**	0.93	106.1	**16.25**	0.92	92.8
	2	14.30	1.03	388.9	15.89	0.87	213.6	**16.12**	0.94	145.0	15.96	0.87	109.9	16.13	0.80	89.1
	3	14.30	1.10	384.9	15.98	0.99	214.8	16.12	0.99	146.7	15.98	0.86	110.8	16.22	0.87	88.6

$N = 1000$		$R = 0.1$			$R = 0.3$			$R = 0.5$			$R = 0.7$			$R = 0.9$		
B	*	imp.	σ	n	imp.	σ	n	imp.	σ	n	imp.	σ	n	imp.	σ	n
6	-2	**12.80**	0.73	1275.3	**16.37**	0.65	622.3	17.78	0.63	420.5	18.98	0.49	359.8	20.17	0.47	303.6
	0	12.61	0.80	1274.6	16.13	0.69	485.4	17.82	0.55	405.0	18.94	0.50	314.6	20.14	0.52	271.2
	2	12.72	0.76	1224.2	16.24	0.63	596.4	17.79	0.59	402.8	19.01	0.50	315.5	**20.21**	0.53	267.2
	3	12.73	0.80	1240.9	16.30	0.67	604.2	**17.86**	0.58	415.1	**19.02**	0.52	331.2	20.12	0.52	278.1
16	-2	**15.19**	0.98	409.6	**17.41**	0.84	196.0	**18.05**	0.77	134.6	18.56	0.75	99.7	**18.96**	0.70	79.8
	0	14.66	0.98	407.4	17.07	0.81	163.5	18.01	0.71	115.0	**18.74**	0.73	87.9	18.83	0.72	72.6
	2	14.84	1.04	387.7	17.30	0.88	179.7	17.85	0.70	117.7	18.49	0.73	88.1	18.73	0.70	72.2
	3	14.89	0.92	390.9	17.31	0.86	182.5	17.92	0.74	119.8	18.34	0.73	91.3	18.68	0.69	75.9
32	-2	**14.96**	0.95	245.7	15.24	0.94	118.1	15.86	0.79	80.3	16.04	0.77	57.7	**16.46**	0.67	45.6
	0	14.46	1.03	207.9	15.02	0.93	98.4	15.77	0.86	66.0	16.01	0.75	48.7	16.35	0.75	39.3
	2	14.75	0.97	222.6	**15.34**	0.91	102.2	15.79	0.84	65.5	16.02	0.77	48.6	16.35	0.75	38.9
	3	14.89	1.01	224.5	15.26	0.83	102.3	**15.92**	0.85	70.3	**16.10**	0.77	50.6	16.22	0.74	39.0

Table 5.5: Results for different values of the twoTermBoost parameter.

previous solution being always merged first) leads to the best results in almost all cases. For this parameter, the additional diversity introduced by lower parameter values does not seem to outweigh the benefits of exploring solutions "near" the current one.

twoTermBoost

The results for the twoTermBoost parameter can be seen in Table 5.5. While it shows a significant influence of the parameter on the solution quality, the concrete influence does not seem to follow any kind of pattern, with negative values, positive values and 0 resulting in the best results for several test groups each.

mixedEdgeBoost

Listed in Table 5.6 are the results for different values of the mixedEdgeBoost parameter. The outcome was similar to that for twoTermBoost – while the setting clearly had some level of impact on solution quality, the concrete influence was as unpredictable as before, making optimization rather difficult.

$N = 500$		$R = 0.1$			$R = 0.3$			$R = 0.5$			$R = 0.7$			$R = 0.9$		
B	*	imp.	σ	n	imp.	σ	n	imp.	σ	n	imp.	σ	n	imp.	σ	n
6	-2	**14.57**	0.54	2854.2	**16.09**	0.48	1222.5	**16.97**	0.49	825.1	17.95	0.50	647.2	18.42	0.42	547.0
	0	14.52	0.50	2845.7	16.01	0.50	1215.8	16.94	0.46	821.8	17.92	0.46	645.4	18.38	0.43	545.6
	2	14.38	0.52	2812.0	15.97	0.50	1185.9	16.83	0.46	808.6	17.92	0.43	636.5	**18.43**	0.44	545.4
	3	14.27	0.45	2855.3	15.86	0.55	1187.0	16.86	0.51	799.3	**18.01**	0.47	630.5	18.41	0.40	544.3
16	-2	**16.29**	0.86	699.9	**16.90**	0.95	382.5	16.99	0.82	265.8	17.54	0.83	207.7	17.89	0.87	173.3
	0	16.03	0.90	692.0	16.77	0.88	372.7	**17.17**	0.91	259.8	17.54	0.80	205.7	17.94	0.87	173.9
	2	15.86	0.87	681.8	16.45	0.95	371.1	16.95	0.89	258.3	**17.71**	0.77	206.6	17.99	0.80	177.1
	3	15.71	0.92	713.2	16.33	0.89	367.3	16.88	0.92	255.5	17.60	0.86	204.2	**18.00**	0.89	176.0
32	-2	**16.15**	0.95	462.5	**16.61**	1.00	255.0	17.02	0.89	175.3	17.56	0.89	134.8	17.65	0.89	111.5
	0	15.98	1.02	456.3	16.48	0.89	247.5	17.02	0.94	170.4	17.58	0.90	132.8	**17.68**	0.82	110.7
	2	15.80	1.02	451.9	16.21	1.03	250.2	**17.17**	0.93	170.7	**17.60**	0.76	134.0	17.62	0.83	114.2
	3	15.67	1.01	471.1	16.11	0.93	247.3	17.01	0.88	168.4	17.52	0.79	131.5	17.62	0.82	112.9
50	-2	**14.43**	1.09	377.0	**16.03**	0.91	208.7	**15.90**	1.01	142.0	15.83	0.84	108.3	**16.23**	0.83	87.9
	0	14.20	1.05	373.3	15.64	0.95	201.6	15.79	0.93	138.4	**15.93**	0.78	106.4	16.19	0.80	87.4
	2	14.04	1.07	369.9	15.52	1.03	203.3	15.85	0.91	137.9	15.89	0.85	107.9	16.22	0.88	90.9
	3	14.19	1.00	384.8	15.57	1.09	202.8	15.64	0.95	136.5	15.74	0.88	104.5	16.04	0.84	89.6

$N = 1000$		$R = 0.1$			$R = 0.3$			$R = 0.5$			$R = 0.7$			$R = 0.9$		
B	*	imp.	σ	n	imp.	σ	n	imp.	σ	n	imp.	σ	n	imp.	σ	n
6	-2	**12.73**	0.77	1187.3	**16.28**	0.67	571.8	17.70	0.53	393.9	18.88	0.54	307.1	20.12	0.53	264.9
	0	12.52	0.73	1164.1	16.12	0.64	558.0	17.78	0.59	384.2	18.98	0.55	299.5	20.11	0.48	259.8
	2	12.39	0.80	1155.5	16.27	0.67	563.9	**17.86**	0.50	387.8	**19.13**	0.50	333.9	**20.18**	0.49	291.3
	3	12.59	0.83	1204.2	16.19	0.68	576.8	17.81	0.61	391.9	19.09	0.54	306.3	20.12	0.48	277.7
16	-2	**14.93**	0.83	379.7	**17.26**	0.83	177.1	**17.97**	0.74	114.3	18.55	0.75	86.1	18.80	0.63	71.0
	0	14.61	0.96	368.6	17.09	0.86	168.0	17.88	0.78	110.2	**18.79**	0.75	84.2	18.88	0.73	69.8
	2	13.85	1.14	329.4	16.64	0.91	172.8	17.93	0.77	118.9	18.64	0.73	91.4	**18.95**	0.73	77.2
	3	14.12	1.03	387.8	16.53	0.88	154.9	17.83	0.78	108.3	18.60	0.69	87.0	18.86	0.75	74.0
32	-2	**14.86**	1.07	219.3	**15.18**	0.89	99.4	**15.74**	0.89	64.4	15.88	0.78	47.5	16.32	0.75	38.5
	0	14.49	1.00	213.8	14.97	0.85	95.3	15.58	0.85	61.6	**16.00**	0.83	45.7	16.44	0.73	38.1
	2	14.09	1.08	215.6	14.67	0.93	100.4	15.51	0.87	69.3	15.89	0.73	52.4	**16.44**	0.74	43.7
	3	14.21	1.09	236.1	14.48	0.98	101.5	15.19	0.78	60.3	15.71	0.74	48.7	16.27	0.78	41.7

Table 5.6: Results for different values of the mixedEdgeBoost parameter.

scpMode

Table 5.7 shows a comparison of the results for the two possible settings of the scpMode parameter – exact or heuristic. While we expected the runtime cost for the exact solution to explode for larger instances, leading to the heuristic variant delivering better results for those, this could not even be observed for 1000 node instances. While these large instances showed an increase in the number of iterations of up to 40% for the heuristic variant, this was clearly predominated by the better results obtained with the exact algorithm. Overall, there were no instances where the heuristic variant led to even slightly better results than the exact one.

Of course, as discussed in Section 3.6, this result is also dependent on the delays being relatively small integers. By distributing the delays over a much larger range, or by allowing real values for delays, the relative performance of both settings could probably be changed. However, for real-world applications it is rarely the case that delays have to be allowed to be so fine-grained as to make such a wide range of possible delay values necessary.

$N = 500$		$R = 0.1$			$R = 0.3$			$R = 0.5$			$R = 0.7$			$R = 0.9$		
B	*	imp.	σ	n	imp.	σ	n	imp.	σ	n	imp.	σ	n	imp.	σ	n
6	*exact*	**14.51**	0.52	2899.3	**16.01**	0.52	1227.4	**16.91**	0.48	827.6	**17.99**	0.48	650.4	**18.46**	0.46	549.7
	heuristic	11.19	0.75	3942.5	11.50	0.75	1833.5	11.69	0.67	1261.4	12.55	0.67	990.4	12.89	0.69	842.3
16	*exact*	**16.07**	0.82	715.8	**16.83**	0.89	382.8	**17.07**	0.85	266.7	**17.62**	0.84	209.7	**17.82**	0.82	177.2
	heuristic	12.16	1.16	961.2	10.67	1.20	577.7	10.41	1.28	413.5	11.01	1.25	327.7	11.47	1.14	277.5
32	*exact*	**15.96**	0.95	475.9	**16.42**	0.94	255.7	**17.09**	0.97	175.5	**17.67**	0.85	136.4	**17.61**	0.86	113.9
	heuristic	12.90	1.35	648.3	12.04	1.30	396.3	12.18	1.14	280.8	12.82	1.13	219.3	12.95	1.07	182.9
50	*exact*	**14.29**	1.00	389.2	**15.71**	1.01	209.5	**15.80**	0.87	141.6	**16.00**	0.88	109.1	**16.24**	0.84	90.2
	heuristic	11.50	1.37	521.7	12.18	1.19	320.5	12.18	1.09	227.3	12.31	1.03	174.7	12.50	1.06	143.7

$N = 1000$		$R = 0.1$			$R = 0.3$			$R = 0.5$			$R = 0.7$			$R = 0.9$		
B	*	imp.	σ	n	imp.	σ	n	imp.	σ	n	imp.	σ	n	imp.	σ	n
6	*exact*	**12.54**	0.80	1205.0	**16.23**	0.63	584.1	**17.83**	0.56	406.9	**19.00**	0.56	324.9	**20.18**	0.54	281.6
	heuristic	6.87	1.00	1573.6	8.08	0.88	803.6	8.70	0.78	562.3	9.92	0.83	425.2	11.47	0.70	361.5
16	*exact*	**14.62**	1.04	383.2	**17.08**	0.82	176.2	**18.00**	0.74	115.6	**18.67**	0.76	89.0	**18.84**	0.74	73.8
	heuristic	8.31	1.36	557.7	9.97	1.13	267.2	10.71	1.03	173.7	11.53	0.94	124.3	12.00	0.99	101.5
32	*exact*	**14.46**	0.95	230.1	**15.10**	0.97	104.0	**15.73**	0.87	67.1	**16.06**	0.75	50.3	**16.33**	0.72	41.7
	heuristic	10.80	1.19	339.1	10.65	1.24	161.8	10.90	1.11	97.1	11.28	0.98	72.4	11.80	0.92	59.1

Table 5.7: Results for different values of the scpMode parameter.

$N = 500$		$R = 0.1$			$R = 0.3$			$R = 0.5$			$R = 0.7$			$R = 0.9$		
B	*	imp.	σ	n	imp.	σ	n	imp.	σ	n	imp.	σ	n	imp.	σ	n
6	0.0	**14.48**	0.53	2898.6	**16.07**	0.49	1219.9	**16.99**	0.48	823.7	**17.97**	0.50	640.9	**18.39**	0.44	542.7
	0.3	12.02	0.74	3759.4	12.34	0.72	2732.7	12.74	0.73	2179.8	13.24	0.56	1747.2	13.43	0.48	1322.7
	0.8	2.22	0.61	6258.1	0.10	0.13	6306.7	0.07	0.04	6606.5	0.02	0.02	6744.0	0.00	0.01	5549.0
16	0.0	**16.01**	0.86	717.0	**16.88**	0.94	379.1	**17.14**	0.95	267.1	**17.54**	0.83	203.3	**17.80**	0.78	172.2
	0.3	10.24	1.52	897.7	7.18	1.36	713.5	5.40	1.47	643.7	3.99	1.43	626.3	2.32	0.94	626.7
	0.8	0.15	0.13	1172.1	0.00	0.00	1133.0	0.00	0.00	1121.9	0.00	0.00	1124.8	0.00	0.00	1141.2
32	0.0	**15.93**	0.99	474.9	**16.40**	0.93	249.5	**16.99**	0.93	170.8	**17.45**	0.83	131.2	**17.65**	0.85	109.4
	0.3	9.53	1.81	617.4	5.28	1.77	540.4	2.38	1.37	525.7	1.18	0.63	527.0	0.80	0.58	525.0
	0.8	0.00	0.00	780.7	0.00	0.00	761.7	0.00	0.00	748.9	0.00	0.00	738.7	0.00	0.00	732.5
50	0.0	**14.21**	0.99	381.2	**15.76**	1.02	203.4	**15.76**	0.97	136.1	**15.98**	0.89	105.2	**16.28**	0.91	86.0
	0.3	6.41	1.76	505.4	2.08	1.37	462.2	0.40	0.48	445.9	0.11	0.20	438.1	0.04	0.11	432.5
	0.8	0.00	0.00	622.9	0.00	0.00	604.6	0.00	0.00	595.4	0.00	0.00	583.3	0.00	0.00	577.3

$N = 1000$		$R = 0.1$			$R = 0.3$			$R = 0.5$			$R = 0.7$			$R = 0.9$		
B	*	imp.	σ	n	imp.	σ	n	imp.	σ	n	imp.	σ	n	imp.	σ	n
6	0.0	**12.68**	0.84	1199.0	**16.32**	0.67	576.4	**17.76**	0.55	398.3	**19.02**	0.54	318.3	**20.15**	0.48	280.5
	0.3	6.13	1.28	1596.5	7.15	0.95	1421.3	6.57	0.74	1111.4	5.97	0.66	929.3	5.11	0.63	954.1
	0.8	0.00	0.00	2973.0	0.00	0.00	3184.4	0.00	0.00	3326.6	0.00	0.00	3687.0	0.00	0.00	4088.5
16	0.0	**14.50**	1.04	376.0	**17.16**	0.80	172.7	**17.98**	0.81	114.0	**18.63**	0.70	87.1	**18.84**	0.73	69.6
	0.3	3.45	1.59	599.0	3.17	1.24	491.8	2.77	1.16	494.0	1.46	0.58	542.7	0.40	0.44	637.5
	0.8	0.00	0.00	971.5	0.00	0.00	963.1	0.00	0.00	953.1	0.00	0.00	951.1	0.00	0.00	951.3
32	0.0	**14.53**	1.00	223.6	**15.15**	0.84	101.7	**15.63**	0.85	65.7	**16.07**	0.73	50.2	**16.28**	0.82	35.8
	0.3	4.03	1.73	357.2	1.08	0.66	342.7	0.27	0.44	345.5	0.06	0.09	423.9	0.00	0.00	440.0
	0.8	0.00	0.00	605.1	0.00	0.00	593.0	0.00	0.00	583.5	0.00	0.00	576.7	0.00	0.00	572.4

Table 5.8: Results for different values of the minKPCost parameter.

	$R = 0.1$		$R = 0.3$		$R = 0.5$		$R = 0.7$		$R = 0.9$	
B	MLR	KPI	MLR	KPI	MLR	KPI	MLR	KPI	MLR	KPI
6	11.88	6.23	15.09	7.94	15.93	8.22	17.08	9.69	17.68	10.26
16	14.17	7.44	15.43	8.37	16.23	9.43	17.08	9.52	17.53	9.72
32	14.69	6.75	15.88	8.83	16.97	8.60	17.01	9.42	17.75	9.42
50	12.99	6.60	15.87	8.49	16.56	8.75	15.98	9.25	16.77	9.25

Table 5.9: Comparison between the Key Path Improvement heuristic (KPI) and the complete Multilevel Refinement heuristic (MLR) presented in this thesis. Shown in each case are the average relative improvements compared to the initially constructed solution.

minKPCost

Statistics for the last tested parameter, minKPCost, are listed in Table 5.8. Similar to the results for the scpMode parameter, the conceived performance improvements for higher values (i.e., inspecting less key paths) in no case outweighed the smaller improvements in solution quality. Moreover, the difference was even more pronounced for this parameter, with higher settings being faster by more than 1000% in some instances, but making the algorithm almost useless in improving the initial solution. This also illustrates the importance of the improvement heuristic for the overall solution quality.

Due to the impressive performance of the KPI heuristic, it was also tested how it would perform alone, without the Multilevel framework, as a simple local search heuristic. The results are listed in Table 5.9 (for 500 node instances). As can be seen, while the complete Multilevel algorithm is obviously superior, the KPI heuristic itself already leads to significant improvements to the originally created solution, and in much shorter time. It might therefore be a good option when a solution has to be found too quickly to use the whole Multilevel algorithm. As the KPI heuristic is deterministic, a longer runtime will not improve the solution quality, though.

5.2 Automatic parameters

Based on the tests in Section 5.1, the parameters for final benchmarks and comparisons were determined. Due to the unequivocal results, scpMode was fixed to "exact", and minKPCost to 0.

For twoTermBoost and mixedEdgeBoost, the results were not meaningful enough to determine their best settings. Furthermore, it could be assumed that these two parameters would be especially closely coupled with each other. Therefore, additional benchmarks were executed for the 500 and 1000 node instances, varying both parameters at the same time. The results are illustrated in Table 5.10. As can be seen, the results were not as random as previously suspected. An area of better results is clearly visible, with a difference of almost half a percentage point in the relative improvement to the initially constructed solution. It can also be seen that the "off" setting 0 for both parameters results in rather good solutions, but not the best. The parameters were therefore fixed to twoTermBoost $= 3$ and mixedEdgeBoost $= -2$ for all further tests.

	-10	-5	-3	-2	0	2	3	5	10
-10	15.30	15.28	15.27	15.29	15.36	15.42	15.40	15.32	15.28
-5	15.28	15.29	15.32	15.35	15.43	15.38	15.46	15.39	15.32
-3	15.23	15.38	15.44	15.31	15.49	15.54	15.48	15.47	15.29
-2	15.39	15.34	15.40	15.49	15.53	15.52	**15.55**	15.43	15.43
0	15.38	15.37	**15.55**	15.51	15.47	15.47	15.52	15.46	15.42
2	15.30	15.45	15.50	15.37	15.25	15.39	15.41	15.45	15.40
3	15.48	15.41	15.41	15.33	15.22	15.24	15.28	15.47	15.43
5	15.40	15.36	15.21	15.18	15.24	15.20	15.21	15.36	15.35
10	15.38	15.20	15.15	15.24	15.19	15.07	15.20	15.19	15.27

Table 5.10: Summarized results of the tests to determine the best settings for the twoTermBoost and mixedEdgeBoost parameters. Different columns represent different values for twoTermBoost, the rows differentiate between the mixedEdgeBoost settings. The values are the arithmetic means of the relative improvement over all test instances. Tested were all 500 node instances with delay bounds of 6, 16, 32 and 50, and all 1000 node instances with delay bounds of 6, 16 and 32. The ratio of terminals took the values 0.1, 0.3, 0.5, 0.7 and 0.9.

For the last two parameters, a different solution was sought, which would both ensure that local optima were reliably found and that the algorithm would still explore other areas of the solution space afterwards. Therefore, varying parameters, which would automatically adapt to the current situation, were implemented. This approach was partly influenced by the work in [41], which describes a way to use such dynamic variations in algorithmic parameters to support both diversification and intensification when searching for solutions.

Algorithm 5.1 gives an overview of how this approach was implemented. After a certain number of consecutive iterations in which no new best solution was found, a variation of the parameters is started. When a new best solution is then reached, the parameters are again reset to their initial values and the limit for unsuccessful iterations is increased by 1 (to automatically adapt to larger time limits).

The exact way in which parameters are varied is illustrated in Figure 5.1. Three automatic modes were designed for the randBoost parameter and four for the treeBoost parameter, each of them testing a different balance between intensification and diversification. Keep in mind, though, that these automatic parameters are only used as long as several consecutive iterations do not find a new best solution.

In the $auto_1$ mode for the randBoost parameter, the parameter is first uniformly raised to 0.5 over the course of the first ten iterations. It is then further raised at twice the slope to 1.5 over the next ten iterations. In the following 20 iterations, this is mirrored to again reach a setting of 0 and start the cycle over from the beginning.

The $auto_3$ setting does the same in principle, but uses `maxNoGain` steps (instead of ten) for each phase. This results in faster variation at the beginning of the program runtime, and slower variation later on, helping to adapt to longer runtimes. Shown in the figure is the variation function when `maxNoGain` would have a value of 5 (i.e., the fifth time parameter variation is

Algorithm 5.1: `runWithAutoParameters()`

Purpose: Runs the Multilevel algorithm on G, adapting the parameters randBoost and treeBoost between iterations.

```
1  randBoost ← 0;
2  treeBoost ← ∞;
3  noGain ← 0;
4  maxNoGain ← 1;
5  while time limit not exceeded do
6  |  coarsen();
7  |  solveCurrentLevel();
8  |  refine();
9  |  if new best solution found then
10 |  |  if noGain > maxNoGain then
11 |  |  |  increment maxNoGain;
12 |  |  end if
13 |  |  noGain ← 0;
14 |  |  randBoost ← 0;
15 |  |  treeBoost ← ∞;
16 |  else
17 |  |  increment noGain;
18 |  |  if noGain > maxNoGain then
19 |  |  |  vary parameters;
20 |  |  end if
21 |  end if
22 end while
```

started during the program run).

The $auto_2$ setting for the randBoost parameter (not shown in the figure) randomly varies the parameter according to the following formula:

$$\text{randBoost} = 2^{\,GetRandom(0.5)-1}$$

Here, *GetRandom(σ)* is the same function as in Section 3.3 – i.e., it returns a random number, following a normal distribution with mean 0 and variance σ^2. The result is a randBoost parameter roughly following the distribution shown in Figure 5.2, with values of about 0.5 being most common.

The $auto_1$ setting for the treeBoost parameter starts off at 5 in the first iteration with varied parameters (coming from ∞ before). It then steadily drops by 0.2 for each further iteration, therefore reaching 0 (or, rather, a minimal value slightly above 0) in the 26^{th} iteration and then remaining at that level, completely banning edges contained in the previous solution from coarsening.

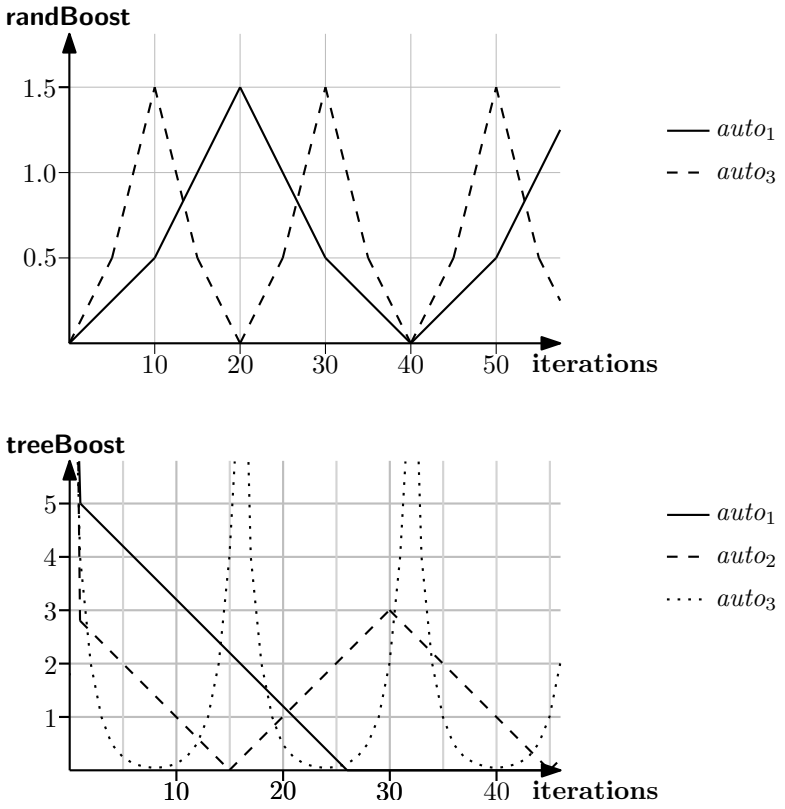

Figure 5.1: The functions according to which the randBoost and treeBoost parameters are varied when using the various (deterministic) automatic settings.

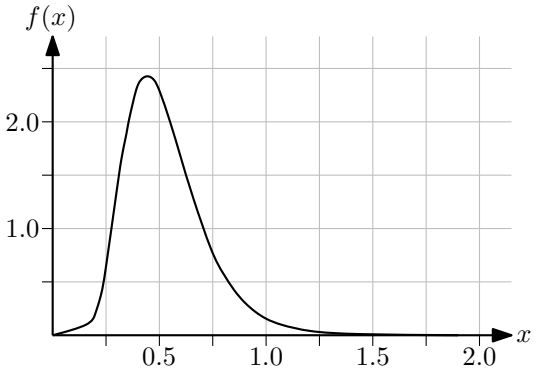

Figure 5.2: The probability density function of the $auto_2$ randBoost parameter.

The $auto_2$ setting, on the other hand, varies the parameter continously between 0 and 3, by alternately dropping to 0 and then rising to 3, always at a rate of 0.2 per iteration. It starts off at a value of 2.8.

With the $auto_3$ setting for the treeBoost parameter, it is not varied in linear, but exponential segments. Starting at a value of 4, the parameter is first halved at each iteration until it reaches the value 0.03125 ($\frac{1}{32}$), and then doubled again at each iteration until it reaches 8. This is repeated periodically.

Finally, there was also a probabilistic setting for the treeBoost parameter, namely $auto_4$ (not shown in the figure). This setting sets the parameter to ∞ with a probability of 50%, or to 1 or 0 with a probability of 25% each.

Results with automatic parameters

Table 5.11 and 5.12 show an overview of the results obtained with the automatic parameters for 500 and 1000 node instances. In these tests, both variables were varied at the same time to find possible correlations between their best settings. The values shown are the aggregations for one specific value of one parameter over all values of the other one – as no particular correlation could be found, a more detailed listing was not considered useful. All other parameters were set to the fixed values mentioned above.

As another note on the results, you will see that no standard deviations are listed for the results. The reason for this is that, due to time and resource constraints, we were not able to execute the test runs more than once for each instance, thus providing not enough data to meaningfully specify standard deviations. Therefore, the results also cannot be considered statistically sound enough to make any definite statements. However, as they were on the whole very consistent, it can be gathered that our conclusions from them will at least be plausible.

Contrary to our hopes, it was not really possible to improve much on the previously best results. In the case of the treeBoost parameter, the results were considerably worse almost throughout with the automatic parameters, compared to the treeBoost $= \infty$ setting. Therefore, the automatic settings were dropped again for this parameter, and it was fixed to ∞ for all further tests.

The results were a bit better for the randBoost parameter. Even though the randBoost $= 0.2$ still showed about as good results as the automatic settings, at least two of the three automatic settings showed continously good results. Since fixed parameters are more sensitive to variations in instance size and runtime, the $auto_1$ setting was eventually chosen for all further tests.

5.3 100 node instances

Even though instances with only 100 nodes can usually too easily be solved exactly to be a useful benchmark for heuristic algorithms, we were still curious how the algorithm would perform for such small instances. Especially, we here had the unique chance to also compare our algorithm to the exact solutions and thus see how much worse our solutions were, on average.

In the tests we used 30 different instances of the size. The delay bound and ratio of terminals were again varied, and 30 test runs executed for each instance and each delay bound and terminal

$N = 500$		$R = 0.1$		$R = 0.3$		$R = 0.5$		$R = 0.7$		$R = 0.9$	
B	*	imp.	n	imp.	n	imp.	n	imp.	n	imp.	n
6	*auto*$_1$	11.88	122.5	**15.09**	112.3	15.93	101.3	17.08	117.4	17.68	112.5
	auto$_2$	11.80	123.7	14.94	110.6	16.01	100.1	17.14	117.6	17.64	112.9
	auto$_3$	11.74	123.0	14.79	111.4	15.87	101.7	16.98	117.1	17.70	113.7
	0.2	12.50	118.5	14.96	109.9	**16.06**	99.3	**17.23**	114.6	17.65	112.4
	0.5	**12.74**	119.0	15.04	111.7	**16.06**	99.6	17.07	114.4	**17.89**	112.0
16	*auto*$_1$	14.17	129.6	15.43	117.4	16.23	117.0	17.08	129.9	17.53	119.0
	auto$_2$	14.31	128.7	15.67	118.1	16.40	116.9	16.90	130.9	17.59	118.4
	auto$_3$	14.11	129.9	15.59	118.6	16.51	117.4	16.81	130.5	**17.83**	118.5
	0.2	**14.55**	124.9	**15.89**	116.1	16.39	115.1	16.30	127.8	17.75	117.2
	0.5	14.50	123.7	15.88	116.3	**16.58**	114.8	**17.19**	128.6	17.69	116.7
32	*auto*$_1$	14.69	126.9	**15.88**	132.8	**16.97**	132.9	17.01	139.2	17.75	141.2
	auto$_2$	**14.87**	126.4	15.65	132.7	16.77	132.9	17.10	139.5	17.72	140.6
	auto$_3$	14.73	126.9	15.78	131.7	16.63	134.2	17.29	139.5	**17.86**	141.4
	0.2	14.43	121.7	15.62	129.5	16.53	132.7	17.27	137.0	17.83	139.4
	0.5	14.13	120.1	15.35	130.6	16.82	132.2	**17.38**	136.8	17.75	135.1
50	*auto*$_1$	12.99	113.8	15.87	139.7	16.56	150.1	15.98	143.5	16.77	147.4
	auto$_2$	**13.07**	111.7	15.86	140.7	16.50	148.7	16.00	143.2	16.62	147.0
	auto$_3$	12.82	111.0	15.67	142.2	**16.65**	149.4	15.96	144.2	16.54	145.5
	0.2	12.70	107.9	**15.91**	135.8	16.04	147.3	**16.20**	142.4	16.76	143.1
	0.5	12.40	106.8	15.52	136.3	16.01	146.7	16.14	142.0	**16.81**	133.1

$N = 1000$		$R = 0.1$		$R = 0.3$		$R = 0.5$		$R = 0.7$		$R = 0.9$	
B	*	imp.	n	imp.	n	imp.	n	imp.	n	imp.	n
6	*auto*$_1$	10.64	110.9	**15.13**	91.9	16.78	78.5	**18.38**	87.1	19.57	87.1
	auto$_2$	10.79	110.1	14.99	92.6	16.81	79.4	18.35	88.0	19.62	88.2
	auto$_3$	**10.99**	109.3	15.12	91.5	16.76	78.4	18.31	87.1	19.54	87.6
	0.2	10.48	108.2	**15.13**	91.8	16.75	79.9	**18.38**	87.9	**19.71**	89.4
	0.5	10.68	109.5	15.09	94.4	**16.93**	82.1	18.24	90.2	19.47	90.1
16	*auto*$_1$	12.03	36.3	15.64	27.0	16.29	19.1	16.80	15.7	16.91	14.1
	auto$_2$	11.95	37.6	**15.68**	26.7	16.24	18.0	16.88	15.6	17.04	14.4
	auto$_3$	12.17	37.2	15.62	27.1	16.32	20.2	16.72	15.9	17.00	14.6
	0.2	12.58	36.6	15.41	26.8	16.37	19.9	**17.01**	15.4	**17.68**	14.2
	0.5	**12.63**	38.9	15.34	26.8	**16.43**	19.6	16.77	15.8	17.60	15.0
32	*auto*$_1$	8.25	9.5	13.18	17.1	14.06	16.6	14.77	13.4	15.65	12.8
	auto$_2$	7.02	9.5	12.90	16.0	14.20	15.0	**14.86**	13.6	15.37	12.5
	auto$_3$	7.61	10.7	**13.19**	17.3	14.24	15.8	14.82	12.9	15.63	12.3
	0.2	**8.44**	12.1	13.12	16.5	**14.44**	15.8	14.50	13.4	15.23	12.8
	0.5	6.60	10.0	12.86	16.3	14.07	12.9	14.69	13.9	**15.84**	21.7

Table 5.11: Test results for automatic and normal settings for the randBoost parameter.

$N = 500$		$R = 0.1$		$R = 0.3$		$R = 0.5$		$R = 0.7$		$R = 0.9$	
B	*	imp.	n	imp.	n	imp.	n	imp.	n	imp.	n
6	$auto_1$	12.33	114.0	14.00	107.5	15.47	98.0	16.74	115.5	17.44	112.5
	$auto_2$	**12.64**	114.0	14.63	107.1	15.47	98.1	16.83	114.2	**17.71**	111.6
	$auto_3$	12.40	112.1	14.21	105.6	15.34	96.7	16.73	115.1	17.42	113.4
	$auto_4$	12.22	115.2	14.63	107.0	16.00	98.1	17.04	115.4	17.39	111.6
	∞	12.50	118.5	**14.96**	109.9	**16.06**	99.3	**17.23**	114.6	17.65	112.4
16	$auto_1$	14.17	121.2	14.35	115.0	15.37	114.7	15.89	128.3	16.29	117.2
	$auto_2$	14.24	121.3	15.29	115.8	15.75	114.7	**16.52**	128.9	17.05	117.1
	$auto_3$	14.18	120.4	14.98	114.3	14.89	113.4	16.07	129.0	16.65	117.6
	$auto_4$	13.94	120.9	15.87	115.6	15.89	115.2	16.37	127.8	17.24	117.9
	∞	**14.55**	124.9	**15.89**	116.1	**16.39**	115.1	16.30	127.8	**17.75**	117.2
32	$auto_1$	13.65	118.4	14.85	130.5	15.27	133.4	16.35	138.8	16.56	139.8
	$auto_2$	13.82	118.1	15.22	129.4	15.54	133.3	16.76	137.5	17.42	138.6
	$auto_3$	13.14	118.7	14.55	129.7	15.27	131.3	16.19	137.6	16.97	139.3
	$auto_4$	14.36	119.1	15.22	127.7	15.91	132.1	16.93	137.5	17.60	139.4
	∞	**14.43**	121.7	**15.62**	129.5	**16.53**	132.7	**17.27**	137.0	**17.83**	139.4
50	$auto_1$	11.65	106.3	13.68	137.0	14.48	145.8	14.31	142.7	15.11	143.9
	$auto_2$	11.65	108.7	13.80	137.0	15.11	147.2	15.15	143.3	16.04	147.3
	$auto_3$	10.82	105.4	13.59	136.3	14.07	146.0	14.56	144.8	15.58	144.6
	$auto_4$	11.63	107.6	14.73	137.6	15.38	146.5	16.19	141.3	15.80	144.4
	∞	**12.70**	107.9	**15.91**	135.8	**16.04**	147.3	**16.20**	142.4	**16.76**	143.1

$N = 1000$		$R = 0.1$		$R = 0.3$		$R = 0.5$		$R = 0.7$		$R = 0.9$	
B	*	imp.	n	imp.	n	imp.	n	imp.	n	imp.	n
6	$auto_1$	9.42	103.1	13.96	92.0	16.22	80.2	17.71	90.5	19.22	92.4
	$auto_2$	9.89	103.8	14.28	89.7	16.39	79.7	17.86	87.8	19.45	89.0
	$auto_3$	9.38	102.6	14.08	90.5	16.23	78.1	17.52	89.9	18.98	91.1
	$auto_4$	10.40	105.6	14.95	91.8	16.66	80.1	18.03	90.6	19.24	90.5
	∞	**10.48**	108.2	**15.13**	91.8	**16.75**	79.9	**18.38**	87.9	**19.71**	89.4
16	$auto_1$	12.32	37.2	15.19	27.0	15.80	19.9	16.57	16.2	17.63	14.9
	$auto_2$	12.06	36.0	15.31	26.7	15.86	19.6	16.70	15.6	17.48	14.8
	$auto_3$	10.86	35.1	14.79	26.5	15.83	19.6	16.07	16.1	17.20	14.2
	$auto_4$	11.60	36.6	15.13	27.2	16.09	19.2	16.68	16.0	17.22	14.4
	∞	**12.58**	36.6	**15.41**	26.8	**16.37**	19.9	**17.01**	15.4	**17.68**	14.2
32	$auto_1$	7.45	10.6	12.90	16.7	14.10	15.6	14.34	12.6	15.15	12.6
	$auto_2$	7.33	11.3	12.94	16.6	13.89	14.8	**14.67**	13.3	14.84	12.6
	$auto_3$	6.50	9.0	12.18	15.7	**14.46**	16.9	13.87	13.8	14.84	12.4
	$auto_4$	7.06	8.9	12.88	15.4	13.59	15.5	14.37	13.2	**15.43**	13.2
	∞	**8.44**	12.1	**13.12**	16.5	14.44	15.8	14.50	13.4	15.23	12.8

Table 5.12: Test results for automatic and normal settings for the treeBoost parameter.

$N = 100$		Construction		Mean			Best	
B	R	opt.	avg.	opt.	avg.	σ	opt.	avg.
16	0.1	5	11.1%	12	1.89%	1.54	23	1.59%
	0.3	0	20.11%	3	4.37%	4.83	8	3.28%
	0.5	0	21.82%	0	5.94%	7.29	1	4.53%
	0.7	0	22.16%	0	5.04%	8.89	0	3.82%
	0.9	0	23.47%	0	5.14%	9.15	0	4.04%
	1	0	24.31%	0	5.56%	10.66	0	4.38%
30	0.1	2	19.3%	15	3.1%	2.83	23	1.1%
	0.3	0	26.57%	3	4.37%	3.76	8	3.18%
	0.5	0	26.95%	2	4.07%	3.91	8	2.99%
	0.7	0	27.55%	0	5.12%	3.81	1	4.09%
	0.9	0	29.08%	0	4.98%	5.92	0	3.74%
	1	0	28.97%	0	5.1%	5.66	0	3.94%
50	0.1	2	14.68%	11	2.27%	1.45	21	0.78%
	0.3	0	23.55%	3	3.43%	1.4	7	2.64%
	0.5	0	28.95%	0	4.54%	2.41	3	3.54%
	0.7	0	30.31%	0	4.22%	2.97	1	3.04%
	0.9	0	30.83%	0	5.06%	4.21	0	3.53%
	1	0	31.28%	0	6.08%	2.85	0	4.95%
100	0.1	2	18.36%	13	2.53%	0.92	24	0.8%
	0.3	0	27.11%	1	3.17%	1.57	11	1.6%
	0.5	0	29.59%	0	4.32%	1.04	1	3.29%
	0.7	0	31.04%	0	5.62%	1.47	0	4.32%
	0.9	0	31%	0	6.46%	1.61	1	5.25%
	1	0	31.75%	0	6.81%	1.11	0	5.93%

Table 5.13: Results of the test runs with 100 node instances. "Construction" marks the results for the initial construction heuristic. "Mean" and "Best" represent the mean and best values obtained with the complete algorithm for each problem instance. For these three data sets, "opt." lists the number of instances in which the respective result was that of the optimum solution, and "avg." lists the average relative deficit compared to the optimal solutions. "σ" for the "Mean" results shows the average standard deviations across the 30 instances.

ratio setting. The time limit for each run was set to 8 seconds, which sufficed for several hundred iterations. A summary of the results is shown in Table 5.13.

As can be seen, even the greedy initial construction heuristic accomplished to find the optimal solution for a number of problem instances, especially for low delay bounds and low number of terminals. (Remember that, since the construction heuristic is deterministic, it always yielded the same result for a given problem instance, so listing mean and best result separately would not be meaningful here.)

For the most interesting results, the mean values of the Multilevel Refinement heuristic, the results are also rather promising. On average, the solutions obtained tend to be worse than the optimal solutions by about two to seven percent. A clear trend is that the solutions tend to get relatively worse for higher delay bounds and higher number of terminals.

The last columns, listing the comparison of the best results obtained for each instance, are also interesting. It shows that for about half of the instances in which a solution was reached, the solution was not reached reliably. It can therefore be concluded that executing the algorithm on a problem multiple times has generally a good chance of finding better values than just running the algorithm once, but with a higher time limit. This means that, e.g., wrapping the algorithm in a GRASP heuristic might be a promising area of further research.

5.4 5000 node instances

To test how well the algorithm can deal with very large instances, it was also used (with the aforementioned parameter settings) on 30 instances with 5000 nodes. In these tests it really showed that the program reached its limits, needing several hours to more than a day for the desired 100 iterations per test run and also almost completely using up the available RAM.

Due to these long runtimes and our limited resources we were not able to run tests in the required quantity to arrive at statistically sound results. A detailed listing of the results is therefore omitted here. They were however also very promising for these large instances, with the algorithm still achieving average improvements of ten to 15 percent compared to the construction heuristic.

5.5 Comparison to other heuristics

After optimizing the parameters, the algorithm was compared to competing algorithms for this problem. The choice for the competitor fell on the *Scatter Search and Path Relinking* algorithm with *Variable Neighborhood Descent* improvement (SSPR-VND), discussed in [19]. This algorithm was selected since it is not only one of the most recent and most successful algorithms for the RDCSTP, but has also benchmark data with well-documented instances publicly available.

As in the original paper, the time limit was set to 60 seconds CPU time for all test runs (although this was far more than needed for these small instances). To obtain meaningful results, 30 test runs were executed for each instance and delay bound. As solving the Steiner Problem without delays is a different problem, with its own well-suited algorithms, the tests for the delay bound ∞ were dropped and therefore only those for $\Delta_1 = 1.1 \times \mathit{Delay}(T_{OPT})$ and $\Delta_2 =$

$0.9 \times Delay(T_{OPT})$ executed. As used in the source, $Delay(T_{OPT})$ here stands for the maximum delay of a path to the source in the solution to the problem without delay bound.

Table 5.14 shows the results of this comparison. "Mean" is the arithmetic mean of the solution tree costs of all 30 runs for the given test instance and delay bound, "Best" is the best result obtained for it. σ denotes the standard deviation of the solution tree cost. "Nr." marks the instances, as used in the referenced paper. "Opt." lists the costs of the optimal solutions for each problem instance. The best mean costs for each instance are marked in bold, results marked with an asterisk signify the optimal solution for an instance.

Before reviewing the results it should be noted that the appropriateness of these test instances as benchmarks for RDCSTP heuristics can be disputed. As the instances are all very small – varying between 50 and 100 nodes –, all these problem instances could as well be solved with exact algorithms, especially in the 60 seconds used for testing. Benchmarks with much larger instances would be considerably better suited for comparing these heuristics, but no existing benchmark data for such instances could be found. Choosing a delay bound that is large enough to not constrain the optimal solution at all (like done with the Δ_1 delay bounds) could also be considered a distortion of the core problem.

It has also to be noted that several of the retrieved test instances seemed faulty, or maybe the listed delay bound wrong. Instance B02 seemed completely defective, while the optimal results for the Δ_2 instances were in several cases *worse* than the ones obtained by the SSPR-VND algorithm according to the paper. These tests were therefore excluded from the comparison and the SSPR-VND column marked with dashes. The unavailable result for instance B14 with the Δ_2 delay bound, however, is due to the SSPR-VND algorithm not finding any valid solution.

The optimal solutions for the Δ_1 delay bounds were taken from the reference paper, while the optima for the Δ_2 delay bounds were computed with the Layered Graph algorithm in [24]. A further demonstration of the inappropriateness of these benchmark instances was that this computation was possible in a few seconds in nearly all cases, with only one instance taking more than the one minute allowed for the tested heuristics.

The results themselves look very promising for the Multilevel Refinement heuristic. While the easiness of the problems lead to results being tied (with both algorithms reliably finding the global optimum) in the majority of cases, the algorithm described in this paper showed slightly better results on the whole, even though both algorithms "won" in several instances. A short summary of the results is given in Table 5.15.

Concretely, when summing up the relative advantage or disadvantage of the Multilevel Refinement heuristic in each test instance, the summed advantage is 2.62% across all instances (2.22% in the Δ_1 instances and an almost-tie in the others). Additionally, the Multilevel Refinement heuristic reliably finds a solution if one exists (even when disregarding the initial solution), while the SSPR-VND heuristic failed to do so in one tested problem instance.

On the whole, these test results indicate that the Multilevel Refinement approach followed in this paper is not only a competitive, but even slightly superior alternative to existing techniques for the RDCSTP. However, as indicated above, further tests have to assess the relative advantage of either algorithm in larger test instances, as these are the main targets of heuristic algorithms.

	Δ_1		Multilevel			SSPR-VND		
Nr.	B	Opt.	Mean	Best	σ	Mean	Best	σ
B01	145	82	**82***	82*	0	**82***	82*	0
B02	228	92	92*	92*	0	–	–	–
B03	248	138	**138***	138*	0	**138***	138*	0
B04	173	59	**59***	59*	0	**59***	59*	0
B05	125	61	**61***	61*	0	**61***	61*	0
B06	281	122	**122***	122*	0	**122***	122*	0
B07	212	111	**111***	111*	0	**111***	111*	0
B08	209	104	**104***	104*	0	**104***	104*	0
B09	280	220	**220***	220*	0	**220***	220*	0
B10	262	86	**86***	86*	0	**86***	86*	0
B11	235	88	88	88*	0.2	**88***	88*	0
B12	225	174	**174***	174*	0	**174***	174*	0
B13	190	165	**165***	165*	0	168.1	165*	1.67
B14	221	235	**235***	235*	0	236.6	235*	4.61
B15	308	318	319.8	318*	0.6	**318.6**	318*	0.92
B16	291	127	**127***	127*	0	**127***	127*	0
B17	219	131	**131.6**	131*	0.9	131.7	131*	1.24
B18	425	218	**218***	218*	0	**218***	218*	0

	Δ_1		Multilevel			SSPR-VND		
Nr.	B	Opt.	Mean	Best	σ	Mean	Best	σ
B01	118	83	**83***	83*	0	**83***	83*	0
B02	187	93	93*	93*	0	–	–	–
B03	203	141	141*	141*	0	–	–	–
B04	142	62	64	64	0	**62***	62*	0
B05	102	62	62.9	62*	0.3	**62***	62*	0
B06	199	124	**125**	125	0	**125**	125	0
B07	173	112	**112***	112*	0	**112***	112*	0
B08	171	107	**107***	107*	0	**107***	107*	0
B09	229	221	**221***	221*	0	**221***	221*	0
B10	215	88	88*	88*	0	–	–	–
B11	180	89	**89***	89*	0	**89***	89*	0
B12	184	175	**177**	177	0	**177**	177	0
B13	139	169	**169***	169*	0	**169***	169*	0
B14	180	237	**237***	237*	0	/	/	0
B15	194	324	**324***	324*	0	332.1	328	5.22
B16	238	129	**129***	129*	0	131.4	129*	1.08
B17	180	133	**133***	133*	0	134	134	0
B18	348	219	**219***	219*	0	**219***	219*	0

Table 5.14: Results from the test instances used in [19], compared to the results obtained there. Δ_1 specifies a delay bound of 1.1 times the maximum delay in the optimal solution to the problem without delays; Δ_2 marks delay bounds of 0.9 times that value.

Alg.	B	solv.	opt.	sum
SSPR-VND	$1.1 \cdot B_O$	**17**/17	**14** (**17**)	
	$0.9 \cdot B_O$	14/15	10 (11)	
MLR	$1.1 \cdot B_O$	**17**/17	13 (**17**)	**-2.2%**
	$0.9 \cdot B_O$	**15**/15	**12** (**13**)	**-0.4%**

Table 5.15: Summary of the comparison in Table 5.14. "solv." lists the number of instances that could be solved in each category by each algorithm. The values in "opt." show for how many instances the algorithms always found the optimal solution, with the value in parantheses being the number of instances where they found it at least once. The "sum" column lists the summed relative advantage of the Multilevel Refinement algorithm, with negative values signifying an advantage.

CHAPTER 6

Conclusions and Future Work

In this thesis we implemented an algorithm based on the Multilevel Refinement meta-heuristic for the Rooted Delay-Constrained Steiner Tree Problem. This problem, which is also known as the Multicast Routing Problem With Delays, has been proven to be $\mathcal{NP}$-hard. Our algorithm was developed in the form of an Iterated Multilevel Refinement heuristic to be used as an improvement heuristic for an existing solution. Since the previous solution was only used in a single spot in each iteration without checking its validity, the algorithm could also be used as the initial construction heuristic, with additional improvement afterwards.

In the algorithm, we first merged nodes based on the cost and delay of their connecting edges, preferring ones with lower cost and delay. Thus, we iteratively created smaller graphs representing higher levels of abstraction of the original problem. In doing so we also saved the additional delays hidden in these merged nodes when connecting other nodes to them. This was done to make the creation of mostly valid solutions on higher levels possible without restricting the solution space too much. A solution tree was then created for the trivial problem on the highest level. Finally, the combined nodes were separated again in reverse order, stepping down again in the multilevel hierarchy. On each level we also executed an improvement heuristic on the solution in this phase. The Key Path Improvement heuristic was chosen for this, leading to considerably improved results. For the $\mathcal{NP}$-hard problem of finding the shortest constrained path between two tree components, which is encountered in that improvement heuristic, we employed a well-known exact pseudo-polynomial dynamic programming approach.

Since initial test results were promising, we introduced additional parameters for the algorithm to further improve its performance. These showed to have a significant impact on the constructed solutions, and meaningful settings resulted in a further increase of solution quality, as benchmarks demonstrated. We also showed that the algorithm has an asymptotic runtime of $T = \mathcal{O}\left(BN^3\right)$, which roughly corresponded to the benchmark results.

Finally, we demonstrated in a test against another algorithm, which used a Scatter Search and Path Relinking approach, that our algorithm can compete well with comparable algorithms, showing on the whole slightly better results for the tested instances. In contrast to the other

algorithm used in the comparison the Multilevel algorithm also reliably found a solution to all tested problems.

For lack of benchmarks for larger instances, however, this comparison was not as meaningful as would be desired. Still, we also demonstrated, without comparison to another algorithm, that our algorithm is capable of successfully handling even large instances of 5000 nodes with moderate resources, considerably improving the initially constructed solutions.

In future work, some of the underlying design decisions might be questioned and put to the test. For instance, we currently only merge each node once at each level, and never merge the source node. These restrictions were incorporated to simplify the algorithm, but better results might be possible when working around these limitations. Also, the explicit and implicit parameters might be inspected more closely. For instance, a parameter restricting the number of merges on each level could be introduced and used to further improve solution quality. Also, an attempt could be made to examine the correlation between problem characteristics (instance size, delay bound, number of terminals) and the best algorithm parameters to produce more intelligent automatic parameters. Finally, improvement heuristics other than the Key Path Improvement heuristic currently used could be tested in the refinement phase. The use of the Key Path Improvement heuristic in other meta-heuristics could also yield good results.

Bibliography

[1] GILBERT, E. N., AND POLLAK, H. O. Steiner minimal trees. *SIAM Journal on Applied Mathematics 16*, 1 (1968), 1–29.

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

[3] KARP, R. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, R. Miller and J. Thatcher, Eds. Plenum Press, 1972, pp. 85–103.

[4] KOMPELLA, V. P., PASQUALE, J. C., AND POLYZOS, G. C. Multicasting for multimedia applications. In *INFOCOM '92. Eleventh Annual Joint Conference of the IEEE Computer and Communications Societies, IEEE* (May 1992), pp. 2078–2085.

[5] KARYPIS, G., AND KUMAR, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput. 20* (December 1998), 359–392.

[6] HENDRICKSON, B., AND LELAND, R. A multi-level algorithm for partitioning graphs. *SC Conference 0* (1995), 28.

[7] TENG, S. Coarsening, sampling, and smoothing: Elements of the multilevel method. *Algorithms for Parallel Processing 105* (1999), 247–276.

[8] RUTHMAIR, M., AND RAIDL, G. R. Variable Neighborhood Search and Ant Colony Optimization for the Rooted Delay-Constrained Minimum Spanning Tree Problem. In *Proceedings of the 11th International Conference on Parallel Problem Solving from Nature: Part II* (2010), R. Schaefer et al., Eds., vol. 6239 of *LNCS*, Springer, pp. 391–400.

[9] KOCH, T., AND MARTIN, A. Solving Steiner tree problems in graphs to optimality. *Networks 32*, 3 (1998), 207–232.

[10] KOU, L., MARKOWSKY, G., AND BERMAN, L. A fast algorithm for Steiner trees. *Acta Informatica 15*, 2 (1981), 141–145.

[11] ZHU, Q., AND PARSA, M. A source-based algorithm for delay-constrained minimum-cost multicasting. In *INFOCOM'95. Fourteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Bringing Information to People. Proceedings. IEEE* (1995), IEEE, pp. 377–385.

[12] ZHENGYING, W., BINGXIN, S., AND LING, Z. A delay-constrained least-cost multicast routing heuristic for dynamic multicast groups. *Electronic Commerce Research 2*, 4 (2002), 323–335.

[13] ZHENGYING, W., BINGXIN, S., AND TAO, M. Dclc routing algorithm based on selective function. *Mini-Micro Computer Systems 21*, 12 (2000), 1267–1269.

[14] ZHENGYING, W., BINGXIN, S., AND ERDUN, Z. Bandwidth-delay-constrained least-cost multicast routing based on heuristic genetic algorithm. *Computer communications 24*, 7-8 (2001), 685–692.

[15] GHABOOSI, N., AND HAGHIGHA, A. A path relinking approach for delay-constrained least-cost multicast routing problem. In *Tools with Artificial Intelligence, 2007. ICTAI 2007. 19th IEEE International Conference on* (2007), vol. 1, IEEE, pp. 383–390.

[16] SKORIN-KAPOV, N., AND KOS, M. A grasp heuristic for the delay-constrained multicast routing problem. *Telecommunication Systems 32*, 1 (2006), 55–69.

[17] XU, Y., AND QU, R. A GRASP approach for the Delay-constrained Multicast routing problem. In *Proceedings of the 4th Multidisplinary International Scheduling Conference (MISTA4)* (Dublin, Ireland, 2009), pp. 93–104.

[18] QU, R., XU, Y., AND KENDALL, G. A Variable Neighborhood Descent Search Algorithm for Delay-Constrained Least-Cost Multicast Routing. In *Proceedings of Learning and Intelligent OptimizatioN (LION3)* (2009), Springer, pp. 15–29.

[19] XU, Y., AND QU, R. A hybrid scatter search meta-heuristic for delay-constrained multicast routing problems. *Applied Intelligence* (2010), 1–13.

[20] NORONHA JR, C., AND TOBAGI, F. Optimum routing of multicast streams. In *INFOCOM'94. Networking for Global Communications., 13th Proceedings IEEE* (1994), IEEE, pp. 865–873.

[21] LEGGIERI, V. *Multicast problems in telecommunication networks*. PhD thesis, Università del Salento, Lecce, Italy, 2007.

[22] LEGGIERI, V., HAOUARI, M., AND TRIKI, C. The steiner tree problem with delays: a tight compact formulation and reduction procedures. Tech. rep., Technical report, Universita del Salento, Lecce, Italy, 2010.

[23] LEGGIERI, V., HAOUARI, M., AND TRIKI, C. An Exact Algorithm for the Steiner Tree Problem with Delays. *Electronic Notes in Discrete Mathematics 36* (2010), 223–230.

[24] RUTHMAIR, M., AND RAIDL, G. R. A Layered Graph Model and an Adaptive Layers Framework to Solve Delay-Constrained Minimum Tree Problems. In *Proceedings of the 15th Conference on Integer Programming and Combinatorial Optimization (IPCO XV)* (2011), O. Günlük and G. Woeginger, Eds., vol. 6655 of *LNCS*, Springer, pp. 376–388.

[25] LEITNER, M., RUTHMAIR, M., AND RAIDL, G. R. Stabilized Branch-and-Price for the Rooted Delay-Constrained Steiner Tree Problem. In *Network Optimization: 5th International Conference, INOC 2011* (Hamburg, Germany, June 2011), J. Pahl, T. Reiners, and S. Voß, Eds., vol. 6701 of *LNCS*, Springer, pp. 124–138.

[26] LEITNER, M., RUTHMAIR, M., AND RAIDL, G. R. Stabilized Column Generation for the Rooted Delay-Constrained Steiner Tree Problem. In *Proceedings of the VII ALIO/EURO – Workshop on Applied Combinatorial Optimization* (Porto, Portugal, May 2011), pp. 250–253.

[27] BRANDT, A. Multilevel computations: Review and recent developments. In *Multigrid methods: theory, applications, and supercomputing;[papers from the 3. Copper Mountain Conference on Multigrid Methods, held at Copper Mountain, Colo., April 5-10, 1987]* (1988), vol. 110, Dekker, p. 35.

[28] WALSHAW, C. A multilevel approach to the travelling salesman problem. *Operations Research* (2002), 862–877.

[29] JOHNSON, D., AND MCGEOCH, L. Experimental analysis of heuristics for the stsp. *The Traveling Salesman Problem and its Variations* (2004), 369–443.

[30] WALSHAW, C. A multilevel approach to the graph colouring problem. In *SE10 9LS* (2001), Citeseer.

[31] BERLAKOVICH, M. Multilevel Heuristiken für das Rooted Delay-Constrained Minimum Spanning Tree Problem. Master's thesis, Vienna University of Technology, Institute of Computer Graphics and Algorithms, Vienna, Austria, July 2010. supervised by G. Raidl and M. Ruthmair.

[32] BERLAKOVICH, M., RUTHMAIR, M., AND RAIDL, G. R. A Multilevel Heuristic for the Rooted Delay-Constrained Minimum Spanning Tree Problem. In *Extended Abstracts of the 13th International Conference on Computer Aided Systems Theory* (2011), A. Quesada-Arencibia et al., Eds., pp. 247–249.

[33] RUTHMAIR, M., AND RAIDL, G. R. A Kruskal-Based Heuristic for the Rooted Delay-Constrained Minimum Spanning Tree Problem. In *Proceedings of the 12th International Conference on Computer Aided Systems Theory* (2009), R. Moreno-Díaz, F. Pichler, and A. Quesada-Arencibia, Eds., vol. 5717 of *LNCS*, Springer, pp. 713–720.

[34] VOSS, S. Steiner's problem in graphs: heuristic methods. *Discrete Applied Mathematics 40*, 1 (1992), 45–72.

[35] LEITNER, M. *Solving Two Network Design Problems by Mixed Integer Programming and Hybrid Optimization Methods*. PhD thesis, Vienna University of Technology, Institute of Computer Graphics and Algorithms, Vienna, Austria, 2010.

[36] GOUVEIA, L., PAIAS, A., AND SHARMA, D. Modeling and Solving the Rooted Distance-Constrained Minimum Spanning Tree Problem. *Computers and Operations Research 35*, 2 (2008), 600–613.

[37] DIJKSTRA, E. A note on two problems in connexion with graphs. *Numerische mathematik 1*, 1 (1959), 269–271.

[38] AHUJA, R., MAGNANTI, T., AND ORLIN, J. *Network flows: theory, algorithms, and applications*. Ed. Prentice Hall. New York, 1993.

[39] RUTHMAIR, M. C++-Framework for Solving Delay-Constrained Tree Problems, 2011.

[40] RUTHMAIR, M. Test instances for the RDCMSTP. `https://www.ads.tuwien.ac.at/~marior/instances/random/`, 2011.

[41] RIBEIRO, C., UCHOA, E., AND WERNECK, R. A hybrid GRASP with perturbations for the Steiner problem in graphs. *INFORMS Journal on Computing 14*, 3 (2003), 228–246.