Technische Universität Wien
Vienna University of Technology

DIPLOMARBEIT

Lagrangian Relax-and-Cut and Hybrid Methods for the Bounded Diameter and the Hop Constrained Minimum Spanning Tree Problems

Ausgeführt am

Institut für Computergraphik und Algorithmen
der Technischen Universität Wien

unter der Anleitung von

Univ.-Prof. Dipl.-Ing. Dr.techn. Günther Raidl
und
Univ.-Ass. Mag. Dipl.-Ing. Martin Gruber

durch

Ferdinand Zaubzer
Matrikelnummer 0026821
Gadaunererstraße 10, 5630 Bad Hofgastein

Wien, Mai 2008

Abstract

The Bounded Diameter Minimum Spanning Tree problem (BDMST) and the Hop Constrained Minimum Spanning Tree problems (HCMST) are NP-hard combinatorial optimization problems which have their main application in network design. In this thesis an existing relax-and-cut approach for finding lower bounds and approximate solutions to those problems is enhanced and extended, and a hybrid algorithm based on the relax-and-cut approach as well as on an existing metaheuristic, namely an ant colony optimization (ACO), is presented.

The enhanced relax-and-cut (R&C) approach is based on an integer linear programming (ILP) formulation which relies on so called jump constraints. The number of jump constraints in this formulation is exponential by means of the instance size. Therefore, violated constraints are identified and relaxed on the fly. The enhanced R&C algorithm is a so called *non deleayed relax-and-cut* algorithm which is based on subgradient optimization. Since the number of separated jump inequalities can be large, a sophisicated management of a pool of such constraints is used. The two main extensions to this R&C approach are the initial identification of jump constraints with corresponding dual variables and a generalization of jump constraints.

The metaheuristic utilized for the hybrid algorithm is an ant colony optimization (ACO) algorithm. ACO algorithms exploit the ability of ants finding short paths between their nest and food sources by depositing pheromone. This works as a positive feedback system. The concept of the hybrid algorithm is the utilization of information obtained by the relax-and-cut approach as a heuristic component in the ACO algorithm that is *mixed* with the pheromone information of the ACO algorithm.

Computational experiments have been performed on previously published benchmark instances. The results have shown that most of the enhancements to the R&C algorithm have lead to significant improvements especially for the lower bounds compared to the original R&C algorithm.

Zusammenfassung

Das minimale Spannbaumproblem mit einer Beschränkung des Durchmessers (Bounded Diameter Minimum Spanning Tree – BDMST) ist ein NP schweres kombinatorisches Optimierungsproblem aus dem Bereich des Netzwerkdesigns. Ein verwandtes Problem ist das minimale Spannbaumproblem mit festgesetztem Wurzelknoten und einer beschränkten Anzahl von Kanten zwischen dem Wurzelknoten und jedem Blattknoten (Hop Constrained Minimum Spanning Tree – HCMST). In dieser Arbeit wird ein bestehender Relax-and-Cut (R&C) Algorithmus zur approximativen Lösung dieser Probleme verbessert und erweitert und ein hybrider Algorithmus basierend auf dem R&C Algorithmus und einer Metaheuristik vorgestellt.

Der R&C Algorithmus basiert auf einer *integer linear programming* (ILP) Formulierung mit einer exponentiellen Anzahl von linearen Bedingungen, den sogenannten Jump-Constraints. Dabei werden nicht erfüllte Bedingungen während der Laufzeit des Optimierungsalgorithmus identifiziert und relaxiert. Verbessert wurde der R&C Algorithmus durch eine aufwändige Verwaltung der identifizierten linearen Bedingungen und durch den Einsatz eines *non delayed* R&C Verfahrens. Die entwickelten Erweiterungen sind der Einsatz von generalisierten Jump-Constraints und eine initiale Erzeugung einer Menge von Jump-Constraints mit zugehörigen dualen Variablen.

Die für den hybriden Algorithmus verwendete Metaheuristik ist eine Ameisenkolonie-Optimierung (ACO). ACO Algorithmen nutzen die Fähigkeit von Ameisen kurze Wege zwischen Nahrungsquellen und ihrem Nest durch das Hinterlassen von Pheromonen am Weg zu finden. Die Funktionsweise entspricht einem positiv rückgekoppelten System. Der hybride Algorithmus verbindet die Pheromoninformationen mit Informationen vom R&C Algorithmus, sodass zur Lösungsfindung neben Pheromonwerten zusätzliche heuristische Werte einfließen.

Experimentelle Berechnungen mit schon bestehenden Probleminstanzen aus der Literatur wurden durchgeführt. Die Resultate zeigen, dass die Verbesserungen des R&C Algorithmus zu deutlich besseren Ergebnissen – im Speziellen zu deutlich höheren unteren Schranken – geführt haben.

Contents

1 Introduction **5**
- 1.1 Applications 6
- 1.2 Definitions 6
  - 1.2.1 Minimum Spanning Tree 6
  - 1.2.2 Bounded-Diameter Minimum Spanning Tree 6
  - 1.2.3 hop-constrained minimum spanning tree 7

2 Previous Work **9**
- 2.1 Exact Methods 9
- 2.2 Heuristic Methods 11

3 Relax-and-Cut Approach for the BDMST Problem **13**
- 3.1 Methods 13
  - 3.1.1 ILPs and Lagrangian Relaxation 13
- 3.2 The Jump-Model for the BDMST 17
  - 3.2.1 Artificially Rooted HCMST 17
  - 3.2.2 The ILP Formulation 17
  - 3.2.3 Heuristics 19

4 Enhancements for the Relax-and-Cut Approach **21**
- 4.1 Employing other Heuristics 21
  - 4.1.1 Ant Colony Optimization 21
- 4.2 Non Delayed Relax-and-Cut 22
- 4.3 Identifying Violated Jump Constraints 25
- 4.4 Managing the Constraint Pool 26
  - 4.4.1 Deleting Inactive Constraints 26
  - 4.4.2 Grouping Constraints into Subsets 27

5 Extensions and Hybrid Methods **28**
- 5.1 Generalized Jump Constraints 28
  - 5.1.1 The Jump-Formulation with Generalized Jump Constraints 29
- 5.2 Creating an Initial Pool of Constraints 30
  - 5.2.1 Methods 31

5.2.2 A Dual-Ascent Algorithm 32
5.2.3 Utilizing the Dual-Ascent Algorithm for the BDMST and HCMST Problems 33
5.3 A hybrid Algorithm Based on ACO and Relax-and-Cut . . . 36
5.3.1 Adding a Heuristic Value to the Ant Colony Optimization 37
5.3.2 Deriving a Heuristic Value from Arc Costs 37
5.3.3 Node Level Assignment with a Heuristic Component . 39
5.3.4 The New ACO Algorithm 39

6 Implementation 41
6.1 Classes and Libraries 41
6.1.1 Employed Third Party Libraries 44
6.2 New Data Structures and Algorithms 44
6.2.1 Constraint Sets and Parameters for Constraints 44
6.2.2 Creation of Node Partitions 45
6.2.3 Solution and Jump Constraint Representation 45
6.3 Usage 46

7 Computational Results 50
7.1 Results for Small BDMST Instances 51
7.2 Benefit of the Constraint Pool Management 51
7.3 Results for HCMST Instances 53
7.4 Extensions of the R&C Approach 54
7.4.1 Initial Constraint Pool 54
7.4.2 Hybrid ACO Algorithm and Generalized Jump Constraints 55

8 Conclusions 57
8.1 Future Work 58

Chapter 1

Introduction

The bounded diameter minimum spanning tree problem is an optimization problem in graph theory. It is related to the minimum spanning tree problem but it has the additional constraint of limiting the maximum number of hops (edges) between any two nodes. The problem is known to be *NP-hard* which means that currently no deterministic algorithm exists which solves the problem to optimality in polynomial time.

The hop constrained minimum spanning tree problem is a similar problem. An HCMST is a minimum spanning tree with a fixed root node and a limited height, that means the path from the root node to any other node must be limited in the number of edges.

Peter Putz has described an ILP Formulation for the BDMST in [Put07] that is mainly based on an HCMST formulation. It is solved with a relax-and-cut algorithm. In this thesis several changes and enhancements for this relax-and-cut approach are presented as well as a generalization of the approach such that it can also be used to solve HCMST problems. In addition a way of combining the relax-and-cut approach with an existing metaheuristic, namely an ant colony optimization, is presented that forms the basis for a new hybridized approach.

Overview of the Thesis

The remainder of this thesis is organized as follows: The next two Sections present definitions and detailed descriptions of the problems as well as different applications. Chapter 2 gives an overview of existing methods for solving the problems. In Chapter 3 the utilized relax-and-cut approach and its fundamentals are described. Chapter 4 presents enhancements for this relax-and-cut approach. Extensions as well as the combination with the metaheuristic for the hybrid approach are described in Chapter 5 and Chapter 6 and 7 present details about the implementation as well as computational results.

1.1 Applications

The bounded diameter minimum spanning tree problem can be found in several applications, mainly in telecommunication network design when the number of hops from one node to any other must be limited, for example for quality reasons. If a maximum transmission delay in packet switched networks from one node to another must not be exceeded the number of hops on the path of the message should be limited. The reason is that the packet switching hops have a more significant impact on communication delay than the length of continuous transmission lines. This is also an application of the HCMST problem, if only the delay between a server and a client is of interest. Limiting the hops on the path from a client to the server increases both, availability and reliability [WA88]. Availability is the probability that a client can connect to a server and reliability is the probability that an established connection will not be interrupted.

Beside network design the BDMST problem can also be found as subproblems of other problems:

In [BK91] Bookstein and Klein describe an algorithm for the compression of correlated bit-vectors based on minimum spanning trees, where the decoding time of a bit-vector depends on its depth in the tree. Thus, an MST with a diameter bound seems reasonable.

In addition it appears as a subproblem of the vehicle routing problem (VRP) [AC90]. This problem describes a fleet of vehicles which have to serve several customers. It is closely related to the travelling salesman problem, since a solution to the VRP consists of several TSP solutions with common start and end points. If the total length of a tour is limited, MST based heuristics such as the Christofides heuristic have to be extended to use BDMSTs instead of MSTs.

1.2 Definitions

1.2.1 Minimum Spanning Tree

Given a connected graph $G(V, E)$ with positive edge costs $c_{v,w}$, a **minimum spanning tree** (MST) is a cycle free subgraph $T(V, E_T)$ that connects all vertices of G and has a minimum sum of edge costs $\sum_{(v,w)\in E_T} c_{v,w}$. Figure 1.1 (a) shows an MST.

1.2.2 Bounded-Diameter Minimum Spanning Tree

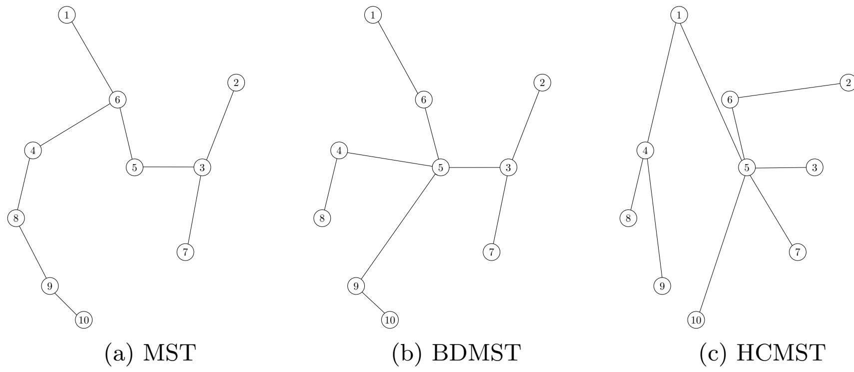
A **bounded diameter minimum spanning tree** (BDMST) with diameter D is a minimum spanning tree in which the path between any two nodes does not contain more than D edges. Figure 1.1 (b) depicts a BDMST with a diameter of 4.

The eccentricity of a node v is defined as the maximum number of edges on the path from node v to any other node in the tree. Given the eccentricity we can define the center for a BDMST with even diameter as the node with minimal eccentricity. For a BDMST with odd diameter the center consists of a pair of nodes, i.e. we have one edge forming the center of the tree. For the BDMST in Figure 1.1 the center node is 5.

The MST problem can be efficiently solved in $O(|E|log|E|)$ with Kruskal's MST algorithm, or in $O(|E| + |V|log|V|)$ with Prim's MST algorithm [CLRS01]. However, adding the diameter constraint increases the complexity of the problem. As shown in [GJ79] the bounded diameter minimum spanning tree problem with diameter D is NP-hard for $4 \leq D < n-1$ where n is the number of nodes of the graph.

1.2.3 hop-constrained minimum spanning tree

The **Hop-Constrained Minimum Spanning Tree** problem (HCMST) is very related to the BDMST. A HCMST with height H is defined as a minimum spanning tree on an undirected connected graph $G(V, E)$ rooted at $r \in V$ and a maximum path length (number of edges) of H for the path from r to any other node in the tree. A BDMST with even diameter D can be seen as a HCMST with a height of $D/2$ that is rooted at the center node of the BDMST. Thus it is possible to adapt BDMST problem formulations (at least for even diameters) such that they can be solved by an algorithm for the HCMST problem and vice versa. Note that finding the optimal center of a BDMST is part of the optimization problem, while in the HCMST problem the root node is given in advance as part of the instance. Figure 1.1 (c) depicts a HCMST with $H = 2$ rooted at node 1. The BDMST (b) could be seen as a HCMST with $H = 2$ rooted at node 5.



(a) MST (b) BDMST (c) HCMST

Figure 1.1: An MST, a BDMST with $D = 4$ and an HCMST with $H = 2$ and root node 1 based on a complete graph of 10 nodes and edge costs proportional to their Euclidean distances.

Chapter 2

Previous Work

Several algorithms have been presented in the last decades to solve the BDMST, HCMST and other similar problems. Some of the algorithms use exact techniques while others are based on heuristic approaches. This chapter gives an overview of the previous work.

2.1 Exact Methods

In [Gou96] Gouveia presents a directed multi-commodity flow formulation (DMCF) for the *terminal layout problem with hop constraints* which is equal to the HCMST problem. He compares this DMCF formulation to an undirected multi-commodity flow formulation (UMCF). For the DMCF undirected edges between two nodes u and v (except the root node) are replaced by two directed arcs (u, v) and (v, u) and each edge between the root r node and any other node v is replaced by only one arc (r, v). In addition, two Lagrangian relaxations derived from the DMCF are described. They present results showing that the bounds given by the LP relaxation of the multi-commodity flow models are very sharp. The results also show that the directed multi-commodity flow model delivers tighter lower bounds than the undirected model.

Dahl et al. present an ILP formulation for the HCMST problem that is based on so called jump constraints in [DFFG05]. The number of constraints is exponential such that not all constraints can be identified in advance. To compute upper and lower bounds with this formulation they use a Lagrangian relaxation based relax-and-cut approach which identifies and relaxes violated constraints on the fly.

In [DGR06] Dahl, Gouveia and Requejo give a survey about several ILP formulations for the HCMST and present comparisons of the different methods. In particular they describe a general framework for the construction of HCMST formulations as well as they present a generic formulation that includes previously known formulations as special cases. The presented

methods all rely on a directed formulation of the HCMST problem since it has proven to be quite useful for many network design problems to direct the problem.

Gouveia et al. present a further modeling approach for the HCMST that views the problem as defined on a layered graph in [GSU07]. It is equivalent to a transformation of the HCMST to a directed Steiner tree problem over a layered graph. With their transformation any model for the Steiner tree problem on the layered graph is a valid model for the HCMST problem. They employ the *hop-cut model* which has an exponential number of constraints and needs a cutting plane algorithm to be solved.

N.R. Achutan et al. present a multi-commodity flow based mixed integer linear programming formulation (MILP) for the BDMST problem using a directed representation of an undirected instance graph as described in [ACCG94]. They distinguish between the cases for odd and even diameters and present different formulations. For the even diameter case they introduce an artificial node r and artificial arcs from r to any other node $v \in V$. For the odd diameter case they do not introduce an artificial arc but binary variable for the selection of the center edge.

In [GM03] Gouveia and Magnanti present multi-commodity flow based formulations for the diameter constrained minimum spanning and Steiner trees. While other multi-commodity flow models for these problems use one commodity for every pair of nodes their model uses a single source of commodities. It simultaneously finds a directed tree with a central node (or edge) serving as the source for the commodities. Their results show that single-sourcing combined with directing the model and using hop-indexed formulations is very powerful. For both, the even and odd diameter case, they use an artificial root node with artificial edges. For the odd case exactly two artificial edges from the root to the other nodes are selected and an extra variable for the edge connecting the two nodes is added to the model. In contrast to a traditional multi-commodity flow model which could not solve a 20-node and 100-edge instance after one week of computation, their model was able to solve this problem in less than 1 second.

An alternative modeling approach is presented by Gouveia et al. in [GMR04], since the odd diameter case proved to be more difficult to solve than the even diameter case with the approaches presented in [GM03]. Their model views the diameter constrained minimum spanning tree as a composition of a variant of a directed spanning tree and two constrained paths – a shortest and longest paths – from the root node to any node in the tree. This leads to linear programming gaps that are one third to one tenth of the gaps of the best previous model.

Santos, Lucena and Riberio present a formulation based on lifted Miller-Tucker-Zemlin inequalities in [dSLR04]. They also use the concept of an artificial root node for the even and odd diameter case.

In [GR05] Gruber and Raidl present – in contrast to multi-commodity

flow formulations – a compact 0–1 ILP formulation. To strengthen their formulation they use a branch-and-cut approach which iteratively identifies violated connection and cycle elimination constraints in the solution of the LP-relaxation and adds them as cuts. The presented results show that their formulation performs well on dense graphs with rather tight diameter bounds.

Putz describes a modification of the relax-and-cut approach mentioned in [DFFG05] for the BDMST problem in his masters thesis [Put07] by adding an artificial root node and artificial arcs like N.R. Achutan et al. in [ACCG94]. However, only the even diameter case is considered. Lower bounds for the optimal value are computed with a relax-and-cut algorithm, which separates and relaxes violated constraints while running. This thesis is fundamentally based on this formulation and introduces many improvements for its implementation. The most significant change is the switch from a delayed to a non delayed relax-and-cut approach. This will be discussed in more detail in Section 4.2.

2.2 Heuristic Methods

While exact methods are rather bound to relatively small instances heuristic approaches can handle much larger ones. Heuristic algorithms can only be employed to compute upper bounds, however, computational results show that some heuristics mostly deliver optimal solutions within relatively short computation times for small instances (when considering complete graphs).

Abdalla et al. [ADG00] describe a greedy heuristic for constructing a BDMST. The one-time tree construction algorithm (OTTC) is a modification of Prim's MST algorithm. The OTTC starts at an arbitrarily chosen node and subsequently connects the nearest node that does not violate the diameter constraint. It is very sensitive regarding the choice of the starting node and it does not *always* create a valid solution if the instance graph is not complete.

A similar construction heuristic is presented by Julstrom. The center-based tree construction algorithm (CBTC) in [Jul04] is an improvement of the OTTC. The selected start node forms the center of the tree such that a node can be connected with a certain edge if the number of edges on the path from the center to the node does not exceed $D/2$. If the diameter is odd a center edge is selected (i.e. a pair of nodes) instead of one center node. This reduces the worst case runtime complexity from $O(n^3)$ (OTTC) to $O(n^2)$. Raidl and Julstrom describe a randomized version of the CBTC, called RTC, which chooses nodes to be connected in the cheapest possible way in random order [RJ03] which leads to better results on Euclidean instances.

Julstrom and Raidl present an evolutionary algorithm in [JR03]. For this algorithm candidate trees are encoded as permutations of vertices. The

first vertex in the permutation (or the first two vertices for the odd diameter case) represent the center node (or nodes) of the tree. The remaining vertices are selected in the order of the permutation and connected with the lowest-weight edge to the tree such that the inclusion of that edge doesn't violate the diameter constraint. This decoding scheme is very similar to the RTC heuristic described above. As crossover operator for two permutations the partially mapped crossover is applied and for the mutation two arbitrary chosen vertices swap their positions.

Several neighborhood searches are described by Gruber, van Hemert and Raidl in [GvHR06]. Based on those neighborhoods three metaheuristics are presented. The four described neighborhoods are the arc exchange, level change, node swap and center exchange level neighborhood. These neighborhoods work on two different solution representations and are combined within a strong local improvement procedure, variable neighborhood descent (VND). The three metaheuristics presented are a variable neighborhood search (VNS), an evolutionary algorithm (EA), and an ant colony optimization (ACO). Both, the EA and the ACO, are based on a node level encoding that does not directly represent a valid solution. Thus, a decoding mechanism building a valid BDMST from the node level information is needed for these metaheuristics. The third metaheuristic presented is a variable neighborhood search which is an improvement heuristic and – as the EA – relies on a construction heuristic for an initial solution. The results achieved by the metaheuristics, especially by the ACO and EA, are comparatively good. The ACO achieves better solutions while the EA converges faster and is therefore more suitable if computation time is strictly limited. The variable neighborhood descent and the ant colony optimization are discussed in more detail in Sections 3.2.3 and 4.1 as they are used as part of my work.

Chapter 3

Relax-and-Cut Approach for the BDMST Problem

The relax-and-cut approach for the BDMST problem – initially presented by Putz in [Put07] – covered in this thesis is based on the jump model for the HCMST by Dahl et al. [DFFG05]. In the following section the fundamentals for that approach are presented.

3.1 Methods

3.1.1 ILPs and Lagrangian Relaxation

Many combinatorial optimization problems can be formulated as an integer linear program (ILP). A typical integer linear program has the following form:

$$\begin{aligned} \min \quad & cx \\ \text{s.t.} \quad & Ax \geq b \\ & x \in \mathbb{N} \end{aligned} \tag{3.1}$$

where c is an n dimensional vector, b is an m dimensional vector, and A is an $(m \times n)$ matrix. $Ax \geq b$ is a constraint set of m constraints which can also be written as:

$$\sum_{j=1}^{n} a_{ij} x_j \geq b_i \qquad 1 \leq i \leq m \tag{3.2}$$

The difference between a linear program and an integer linear program is that for the linear program the values of the solution vector are real values, i.e. $x \in \mathbb{R}^n$, while in general for an integer linear program only integer values can form the solution vector: $x \in \mathbb{N}^n$. However, many integer linear programs have an integrality constraint of the form $x \in \{0, 1\}^n$. Such ILPs are called 0–1 integer linear programs.

While linear programs can be solved efficiently in general, e.g. with the simplex algorithm (see [Wol98]), integer linear programs are quite complex to solve. A common way to obtain upper or lower bounds (depending whether the optimization problem is a maximization or minimization problem) for integer linear programs is the LP-relaxation. In the LP-relaxation the integrality constraint of the integer linear program is relaxed reducing the ILP to an LP which in turn can be efficiently solved. For minimization problems the resulting solution is a lower bound for the optimal solution of the the ILP. Another way to compute upper or lower bounds for integer linear programs is *Lagrangian relaxation*. For simplicity, we only consider minimization problems in the following sections.

Lagrangian Relaxation

With Lagrangian relaxation one or more constraint sets of an integer linear program are relaxed by bringing them into the objective function such that the relaxed program gives a lower bound to the optimal solution of the original problem. Given an integer linear program

$$\begin{aligned} \min \quad & cx \\ \text{s.t.} \quad & Ax \geq b \\ & Bx \geq c \\ & x \in \{0,1\} \end{aligned} \tag{3.3}$$

a possible Lagrangian relaxation is:

$$\begin{aligned} \min \quad & cx + \lambda(b - Ax) \\ \text{s.t.} \quad & Bx \geq c \\ & x \in \{0,1\} \end{aligned} \tag{3.4}$$

One of the constraint sets of (3.3) is removed and brought into the objective function by attaching a *Lagrange multiplier* vector $\lambda \geq 0$ to that constraint set. If the constraints in A are violated $\lambda(b - Ax)$ is positive and thus can be seen as penalty. If the constraints are satisfied a negative term is added to the function. Additionally, omitting a constraint set for the ILP can only reduce its objective value. As a result, the solution of (3.4) is a lower bound to the optimal solution of the original problem. For the proof the interested reader is referred to [Bea93]. (3.4) is called the Lagrangian lower bound program (LLBP).

The obtained problem is obviously easier to solve since we relaxed a constraint set. However, the lower bound obtained by this problem often is not very tight. We are interested in the *best* lower bound, that means the

lower bound with the greatest value. This leads us to the following problem:

$$\max_{\lambda \geq 0} \left\{ \begin{array}{ll} \min & cx + \lambda(b - Ax) \\ \text{s.t.} & Bx \geq d \\ & x \in \{0,1\} \end{array} \right\} \tag{3.5}$$

It is called the *Lagrangian dual* problem. The key to a *good* value of the Lagrangian dual problem is to find good values for the Lagrange multipliers, i.e. the vector λ. A well known method for finding good values for λ is the *subgradient optimization* which is discussed in the next section.

Besides only calculating lower bounds Lagrangian relaxation can also be helpful with finding feasible solutions of good quality, i.e. upper bounds for minimization problems. Given a solution of the Lagrangian lower bound problem which is mostly infeasible together with a *repair heuristic* we can perform adjustments such that we obtain a feasible solution.

Subgradient Optimization

The subgradient optimization is an iterative method for finding values of the Lagrange multiplier vector that lead to a maximal value for the solution of the LLBP. The two main components the subgradient optimization relies on are a good upper bound and a solver for the LLBP. The upper bound can be obtained with a heuristic. In the following the solution of the LLBP will be denoted as z_{LLBP} and the upper bound as z_{UB}.

The central concept introduced in subgradient optimization is the subgradient vector δ. Based on the values of z_{LLBP}, z_{UB}, and δ_i the values for the Lagrange factors λ_i are iteratively adjusted. To present a more detailed view on the subgradient vector we first switch to the other notation for the constraint set $Ax \geq b$:

$$\sum_{j=1}^{n} a_{ij}x_j \geq b_i \qquad i = 1, ..., m \tag{3.6}$$

Now δ_i, the subgradient for the ith constraint in solution x, is defined as follows:

$$\delta_i = b_i - \sum_{j=1}^{n} a_{ij}x_j \tag{3.7}$$

The value of δ_i gives a notion of how far the solution x is from saturating the ith constraint with equality. If δ_i is negative the ith constraint is *oversaturated* and if δ_i is positive the ith constraint is violated for x. In both of these cases the Lagrange factor λ_i will be changed for the next calculation of x: If δ_i is positive, λ_i is increased to increase the penalty for the violated constraint and if δ_i is negative λ_i is decreased. Only if $\delta_i = 0$ – this is the case when the ith constraint is satisfied with equality – λ_i remains unchanged.

In order to iteratively let λ_i approach the values where z_{LLBP} is maximal the adjustment of λ_i must be controlled by a so-called step size. The smaller the difference between z_{LLBP} and z_{UB} the smaller the adjustments must be.

Therefore the step size Δ is defined as follows:

$$\Delta = \frac{\pi(z_{UB} - z_{LLBP})}{\sum_{i=1}^{m} \delta_i^2} \tag{3.8}$$

The parameter π, called agility, enables a better convergence. After a specific number of iterations without any improvement, the value of π is reduced resulting in smaller changes of the Lagrange multipliers.

The steps of the procedure for the subgradient optimization as described in [Bea93] are:

1. Initialize π and the Lagrange multipliers λ_i. For π Beasley suggests a value of 2.
2. Calculate z_{LLBP} with the current values of λ_i.
3. Evaluate the subgradients as defined by (3.7).
4. Calculate the step size as defined by (3.8).
5. Adjust the Lagrangian multipliers λ_i and – if necessary – reduce π.
6. If termination condition is not satisfied continue at step 2.

Possible termination conditions could be:

- π is too small.
- The maximum number of iterations is reached.
- $z_{LLBP} = z_{UB}$ which means the optimum of the original problem was found.

Relax-and-Cut Algorithms

For several ILP formulations not all constraints can be added to the model in advance since their number is too high (e.g. exponential). In such a case violated constraints can be identified (separated) in the current solution and relaxed during computation. Two different types of relax-and-cut algorithms can be distinguished: *delayed* R&C, as utilized in the implementation of Putz (see [Put07]), and *non-delayed* R&C, as described by Lucena in [Luc05]. The main difference between a delayed and a non-delayed relax-and-cut algorithm is the strategy when to identify new constraints: While for the non-delayed relax-and-cut approach it is essential to separate new violated constraints at every iteration of the subgradient algorithm the delayed relax-and-cut algorithm identifies new constraints only after the subgradient algorithm has converged to some degree.

3.2 The Jump-Model for the BDMST

This section gives a brief overview about the relax-and-cut approach from Putz. For a detailled discussion of the algorithms see [Put07]. This approach for the BDMST problem is based on an ILP formulation for the HCMST problem. As mentioned in Section 1.2.3 a BDMST can be seen as a HCMST with an additional constraint. In fact, with the jump model a HCMST problem is solved. Therefore, the instance of the BDMST problem has to be transformed into an artificially rooted HCMST instance.

3.2.1 Artificially Rooted HCMST

Suppose a BDMST problem with an undirected instance graph $G(V, E)$, with V being the node set and E the set of edges, and an even diameter bound of D (In this work only the even diameter case is considered). To utilize the jump model for the BDMST problem an artificial root node r_{art} is introduced with artificial directed arcs $(r_{art}, v)|v \in V$ to every other node in the graph. The resulting graph can now be seen as an instance graph for the HCMST problem with a hop limit of $H = D/2 + 1$. The costs of the artificial arcs are set to a large constant such that any resulting HCMST will only contain one of these artificial arcs (r_{art}, v). In a solution this node v connected to r_{art} represents the center node of the BDMST and r_{art} itself and the artificial arc (r_{art}, v) are not included in the BDMST solution.

Additionally, the jump model is defined on directed instance graphs so the undirected instance graph $G(V, E)$ has to be transformed into a directed one, $G(V, A)$, with A being the set of directed arcs. Each edge (u, v) of the undirected graph is replaced by two directed arcs (u, v) and (v, u) with arc costs equal to that of the edge. The resulting augmented graph $G^+(V^+, A^+)$, with $V^+ = V \cup \{r_{art}\}$, $A^+ = A \cup \{(r_{art}, v) \forall v \in V\}$ can now be seen as a directed interpretation of a HCMST instance graph. On this graph the equivalent to an MST is a *minimum spanning arborescence*. A minimum spanning arborescence is a directed tree T in which for any node $v \in V$ there exists exactly one directed path from the root node – in this case r_{art} – to v where the sum over all arc costs $c_{(u,v)} \in T$ is minimal.

3.2.2 The ILP Formulation

The ILP formulation of the jump model is defined for the HCMST problem on a directed interpretation of the instance graph. By applying the transformation described above it can also be used for the BDMST problem. For the ease of reading the following definitions are based on a directed graph $G(V, A)$ which is the directed interpretation of a HCMST instance graph, obtained by replacing all undirected edges (u, v) with two directed arcs (u, v) and (v, u) with arc costs equal to that of the corresponding edge. Note that

the edges from the root node r to any other node v are only replaced by one directed arc (r, v). The formulation is based on variables $x_{u,v}$ defined as:

$$x_{u,v} = \begin{cases} 1 & \text{if the arc } (u,v) \text{ is in the solution tree } T \\ 0 & \text{otherwise} \end{cases} \tag{3.9}$$

The constraints used in this formulation to enforce the hop respectively the diameter bound are so-called jump constraints that are derived as follows:

Given a HCMST problem with hop limit H and root node r all nodes of the tree are partitioned into $H + 2$ non-empty sets such that the root node r is assigned to the set S_0 and the other nodes are assigned to the sets $S_1, .., S_{H+1}$. Upon such a node partition P we can define a jump J as follows:

$$J = \{(u, v) \in A | u \in S_i, v \in S_j, j \geq i + 2\} \tag{3.10}$$

It is the set of all arcs that jump over at least one set S_i in the partition P. Since any valid HCMST with hop limit H *must* contain at least on arc out of every possible jump the jump constraints can be formulated as:

$$\sum_{(u,v)\in J} x_{u,v} \geq 1 \qquad \forall J \in \Pi \tag{3.11}$$

where Π is the set of all jumps induced by all possible valid partitions ($r \in S_0, |S_i| > 1 \forall i \in \{1 \ldots H + 1\}$).

This leads us to the following ILP formulation:

$$\min \quad \sum_{a\in A} c_a x_a \tag{3.12}$$

$$\text{s.t.} \quad \sum_{a\in J} x_a \geq 1 \qquad \forall J \in \Pi \tag{3.13}$$

$$x \text{ is a spanning arborescence in } G(V, A) \text{ rooted at } r \tag{3.14}$$

By relaxing the constraint set (3.13) we obtain the following Lagrange relaxed ILP:

$$\min \quad \sum_{a\in A} c_a x_a + \sum_{J\in\Pi} \lambda_J (1 - \sum_{a\in J} x_a) \tag{3.15}$$

$$\text{s.t.} \quad x \text{ is a spanning arborescence in } G(V, A) \text{ rooted at } r$$

The objective function (3.15) can also be written as:

$$\min \quad \sum_{a\in A} \alpha_a x_a + \sum_{J\in\Pi} \lambda_J \tag{3.16}$$

where α_a is given by

$$\alpha_a = (c_a - \sum_{J\in\Pi|a\in J} \lambda_J) \tag{3.17}$$

The relaxed program can be efficiently solved with Edmond's algorithm for a minimum spanning arborescence. For more details on this algorithm see [Gib85].

3.2.3 Heuristics

The solutions obtained by the described relax-and-cut algorithm form lower bounds for the optimal solution of the original problem. Mostly, these solutions are infeasible and therefore must be transformed to feasible solutions to obtain upper bounds for the optimal solution. This problem can be solved with heuristics described in this section.

In order to transform an infeasible solution into a feasible one, the node levels of the infeasible solution are identified. The level of a node v is the number of arcs on the path from the root node r to v where r is assigned to level 0. This node level information will contain invalid values for nodes which exceed the hop limit in the infeasible solution. Those nodes are assigned to the maximal valid level. The resulting list of node levels is then decoded to a valid solution with a decoding algorithm described in [GvHR06]. It finds the least cost predecessor p for every node v at level l by considering all nodes from levels 0 to $l - 1$ and connects node v to its cheapest possible predecessor node p. After decoding the node level information an improvement heuristic which is described in the following section is applied to the valid BDMST.

Variable Neighborhood Descent

The variable neighborhood decent (VND) described in [GvHR06] is a local improvement heuristic using four different local neighborhoods. Based on an initial solution the four neighborhoods are completely explored in a specified order, always the best move found in one of the neighborhoods is applied. All neighborhoods lead to local optimal solutions. However, they can be different. Therefore, neighborhood 1 is completely explored until a local optimum is reached (no better solution can be found any more in this neighborhood). Then, neighborhood 2 is explored until the local optimum is reached and if an improvement was found in neighborhood 2 the algorithm restarts with neighborhood 1, otherwise the VND will continue with neighborhood 3 and so forth. Only when in all four neighborhoods no better solutions can be found any more the algorithm terminates. The four neighborhoods utilized by this algorithm are:

1. Arc exchange neighborhood:

   A neighboring solution with respect to the arc exchange neighborhood is a tree that differs in exactly one arc from the current solution. Considering a complete graph with n nodes the number of neighboring solutions is in $O(n^2)$.

2. Node swap neighborhood:

   The node swap neighborhood defines neighboring solutions to a current one by exchanging the position of a node u and one of its direct

successors v. This neighborhood consists of $O(n)$ solutions.

3. Level change neighborhood:

   For the level change neighborhood an adjacent solution to a current one is obtained by incrementing or decrementing the level of exactly one node. This neighborhood has a size of $O(n)$, however, computation time for an implementation of this neighborhood is in $O(n^2)$ since for each node to be moved it has to be checked if nodes have to be reconnected to another predecessor.

4. Center exchange level neighborhood:

   Adjacent solutions to a current one with respect to the center exchange level neighborhood are defined by replacing the center node by a non-center node. The original center node is assigned to level H such that it can be reconnected to the largest number of potential predecessors. The size of this neighborhood is $O(n)$ while the implementation of one movement has a worst-case time of $O(n^2)$ which leads to an overall complexity of $O(n^3)$.

CBTC and RTC

For calculating initial upper bounds for the subgradient algorithm other heuristics are utilized since at this time no infeasible solution that can be transformed to a feasible one yet exists. The *center based tree construction* heuristic (CBTC) selects one node as the center node and then subsequently connects – like Prim's MST algorithm – the remaining nodes to the tree with the cheapest available edge that does not violate the diameter bound.

The *randomized center based tree construction* (RTC) is a variant of the CBTC selecting the center node as well as the order of the remaining nodes for subsequent processing randomly.

Again, a solution constructed by one of the two heuristics is improved with the variable neighborhood decent local improvement procedure.

Chapter 4

Enhancements for the Relax-and-Cut Approach

The relax-and-cut approach described in Chapter 3 can be enhanced in several ways to improve the performance and the quality of the solutions obtained. This Chapter presents several enhancements effecting both, the quality of the upper and lower bounds computed by the relax-and-cut algorithm.

4.1 Employing other Heuristics

As presented in Section 3.2.3 a heuristic is employed to create a feasible solution from an infeasible one that is obtained by the solver for the LLBP. This heuristic is mainly based on node level information. Experiments have shown that – especially for larger instances – the upper bounds obtained by this heuristic are not as good as the solutions computed with metaheuristics such as the *ant colony optimization* (ACO) or the *evolutionary algorithm* (EA) described in [GvHR06]. In addition, these metaheuristics are based on node level information as well. Hence, it seems promising to employ one of these heuristics for the calculation of the upper bounds in the relax-and-cut algorithm. Since the ACO algorithm delivers the better results we prefer it to the EA. In the following a brief overview of the concepts of ACO algorithms in general as well as a description of an ACO algorithm for the BDMST problem is given.

4.1.1 Ant Colony Optimization

As several other metaheuristics the *ant colony optimization* (ACO) has its origins in processes of the nature. It is a positive feedback system. The first ant colony algorithm was presented by Dorigo [Dor92]. Several ant colony algorithms exist, mainly for optimization problems on graphs as the

traveling salesman problem [DG97].

The main idea of ant colony optimization comes from the ability of ants finding a short path from food sources to their nest. Ants deposit pheromone on their ways and prefer ways with more accumulated pheromone deposited by other ants. Suppose there are several possibilities for the path between two points. One ant cannot know in advance which is the shortest so it can be expected ants choose either way with the same probability. On their way ants deposit pheromone and since on short paths more ants can pass within the same time (if ants walk in both directions) the amount of pheromone accumulated on the shorter path is larger. Hence, ants will begin to prefer the shorter path.

The ACO Algorithm for the BDMST Problem

In the ant colony optimization for the BDMST problem [GvHR06], pheromone is not artificially deposited on paths (between nodes) but in a matrix. For a BDMST problem instance with a diameter D and n nodes in the instance graph the pheromone matrix consists of n columns and $H+1$ rows where $H = \lfloor \frac{D}{2} \rfloor$. A pheromone value at column i and row l denotes the probability for node i to be at level l in the BDMST (or HCMST). Clearly, a decoding mechanism is needed to build a BDMST or HCMST from the level information of nodes. This decoding mechanism is the same as in the heuristic described in Section 3.2.3.

The ACO algorithm now works as follows: At each iteration several (virtual) ants build a BDMST based on the values in the pheromone matrix. The ant that produced the best solution is allowed to deposit pheromone to the pheromone matrix according to the node levels of its solution. In addition, accumulated pheromone decays at each iteration to smooth the distribution of pheromone values.

The ACO Algorithm in the Relax-and-Cut Approach

The simple integration of the ACO algorithm into the relax-and-cut approach can only be done by using the ACO algorithm for the calculation of the initial upper bound before the subgradient optimization starts. The upper bounds computed by the ACO algorithm outperform the results of the heuristic presented in Section 3.2.3 in most cases such that this heuristic seems obsolete.

4.2 Non Delayed Relax-and-Cut

Based on the jump-model described in Chapter 3 Putz describes a *delayed relax-and-cut* algorithm in [Put07]. This algorithm has shown to produce a relatively small pool of constraints which was too small to compute tight

lower bounds. The scheme of the delayed relax-and-cut algorithm is the following:

1. Solve the LLBP with an empty constraint set.
2. Identify violated constraints based on the initial solution obtained in step 1.
3. Perform the subgradient algorithm until it converged.
4. Separate new constraints that are violated at the end of the run of the subgradient algorithm in step 3.
5. Continue at (3) until a termination condition is met, like no improvement in the last runs of the subgradient algorithm.

In this algorithm, constraints that are separated during a run of the subgradient algorithm are not used before the Lagrangian dual problem is solved. After a restart of the subgradient algorithm the new constraints are used. Additionally constraints are only separated at the end of each run of the subgradient algorithm which results in a smaller number of identified constraints.

In contrast to the algorithm sketched above, Lucena describes *non delayed relax-and-cut* algorithms in [Luc05]. The main difference to delayed relax-and-cut algorithms is that constraints are separated at every iteration of the subgradient algorithm. Lucena states that this is an important step. The relax-and-cut algorithm implemented to substitute the scheme described above can be seen as a modified subgradient algorithm. It is presented in more detail in Algorithm 1.

The terminate() function at line 6 returns true if the agility is too small, the maximal number of iterations is reached, or an optimal solution is found. The deleteInactive() function deletes at most as many inactive (saturated or oversaturated constraints with a Lagrange multiplier of 0) constraints as are in Π_{new}, the set of new constraints. If there are not enough inactive constraints in Π that can be deleted the set Π_{new} is reduced such that the overall maximum number of constraints is not exceeded. The function improved() at line 20 returns false if in the last 30 iterations the lower bound z_{LLBP} did not increase or the upper bound z_{UB} did not decrease. Note that not only new constraints are added but also previously added constraints are deleted. Since the number of constraints can increase rapidly when violated constraints are identified at every iteration of the subgradient algorithm it is necessary to also delete constraints, as computation time increases with the number of constraints. For the exact decision strategy about the deletion of constraints see Section 4.4.

Note that Algorithm 1 is based on the classical subgradient optimization as described in [Bea93]. The following enhancements to the subgradient optimization are applied as also shown in [Put07].

Algorithm 1 NonDelayedRelaxAndCut

```
1: Π ← ∅    // initialize the constraint set
2: λ ← 0    // initialize the Lagrange multiplier vector to 0, since no constraints are known in advance
3: Π_new ← ∅    // initialize temporary constraint set
4: π ← 2
5: lowerBound ← 0    // initialize lower bound to 0
6: while (¬terminate()) {
7:    x_LLBP ← LLBP(Π, λ)    // solve the relaxed problem to optimality
8:    z_UB ← calculateUpperBound(x_LLBP)    // calculate an upper bound with a basic heuristic
9:    δ_J ← 1 − J · x_LLBP    ∀J ∈ Π    // compute subgradient
10:   Π_new ← getViolatedConstraints()
11:   deleteInactive(Π, Π_new)    // delete some inactive constraints
12:   Θ = { ||δ||/||κ_prev||,  if δ · κ_prev < 0
            0,                 otherwise }    // compute the weighting factor for the direction vector
13:   κ_J = δ_J + Θκ_prev J    ∀J ∈ Π    // compute the direction vector
14:   Δ = π(z_ub − z_LLBP) / Σ_{J∈Π} δ_J κ_J    ∀J ∈ Π    // compute step size
15:   λ_J = max(0, λ_J + Δ · κ_J)    ∀J ∈ Π    // update Lagrangian multipliers
16:   Π ← Π ∪ Π_new
17:   if (z_LLBP > lowerBound) {
18:      lowerBound ← z_LLBP    // update lower bound if necessary
19:   }
20:   if (¬improved()) {
21:      π ← π/2    // reduce agility
22:   }
23: }
```

- For the computation of the step size Δ we can increase the upper bound such that the step size does not get too small. A factor of 1.05 is suggested in [Bea93].
- By setting the subgradient δ_i to 0 in the current iteration if $\lambda_i = 0$ and δ_i was < 0 in the previous iteration, we prevent the step size from getting too small if a lot of values in the subgradient vector would be negative.
- As described in [CFG01] the computation of the step size and the update of the Lagrange multipliers is performed with a direction vector κ. The values of this direction vector are computed as

  $$\kappa_J = \delta_J + \Theta \kappa_{prev\,J}$$

  where Θ is a weighting factor and κ_{prev} is the direction vector of the

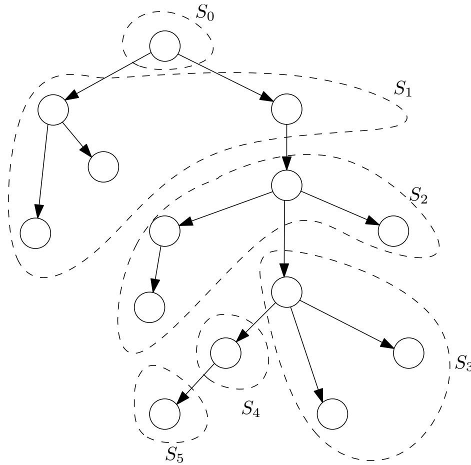


Figure 4.1: Concept of the path approach for partitioning nodes

previous iteration. The step size is computed as

$$\Delta = \frac{\pi(z_{ub} - z_{LLBP})}{\sum_{J \in \Pi} \delta_J \kappa_J}.$$

The weighting factor Θ is

$$\Theta = \begin{cases} ||\delta||/||\kappa_{prev}||, & \text{if } \delta \cdot \kappa_{prev} < 0 \\ 0, & otherwise \end{cases}$$

The update of the Lagrange multipliers is performed in the following way:

$$\lambda_J = \max(0, \lambda_J + \Delta \cdot \kappa_J)$$

4.3 Identifying Violated Jump Constraints

The methods for separating jump constraints utilized in [Put07] led to a constraint pool of suboptimal quality. Therefore, another approach for identifying violated jump constraints has been adopted which is described in the following paragraphs.

Violated jump constraints are separated based on infeasible solutions produced by the solver for the relaxed ILP. Each solution of this solver is a minimum spanning arborescence. Most of these solutions are infeasible (violating the hop respectively the diameter constraint) which means that they contain nodes at a level $l \geq H + 1$. For each such node u (which does not have to be a leaf node) we can find a node partition of $H + 2$ non-empty sets with the following strategy described in [DFFG05]:

All the nodes on the path $(r, v_1)(v_1, v_2) \dots (v_i, u)$from the root node r to node u are placed into a set according to their depth in the tree, i.e. a node of depth k will be assigned to set k. The rest of the nodes is assigned recursively to the same set as their direct predecessors. This approach is denoted as the *path approach* in [DFFG05]. Clearly, it would lead to a large set S_0 as any node on a path directly originating from r that has no common arc with the path $(r, v_1)(v_1, v_2) \dots (v_i, u)$ will be assigned to the same set as node r. Therefore, nodes v that are not on the path from r to u are assigned to set l with $l = \max(1, set(pred(v)))$ where $set(pred(v))$ denotes the index of the set the predecessor of node v is assigned to. This concept is depicted in Figure 4.1.

Clearly, for long $r - u$ paths the number of sets in the partition can be greater than $H + 2$. If the resulting node partition consists of more than $H + 2$ sets it is *compressed* by merging sets. For a partition with $H + i$ sets we can merge $i - 1$ sets. These could either be sets $S_1 \dots S_{i-1}$, sets $S_H \dots S_{H+i-2}$, or any sequence of $i - 1$ sets between S_1 and S_{H+i-2}. The first sets S_0 and the last set S_{H+i-1} should be left unaffected such that set S_0 remains singleton and the number of nodes in set S_{H+i-1} does not increase since we will make this set singleton in the next step.

If – after compression is applied – the last set (S_{H+1}) contains more than one node, we have to move all nodes except one into the set S_H such that the last set gets singleton. Dahl et al. proved in [DFFG05] that an inequality induced by such a node partition is facet defining.

4.4 Managing the Constraint Pool

Since the total number of constraints can increase very quickly it is necessary to employ a mechanism to keep the constraint pool below a specified size. Two concepts presented by Lucena in [Luc05] have been adopted for the relax-and-cut algorithm presented above.

4.4.1 Deleting Inactive Constraints

Lucena introduces three classes of constraints: *currently violated active* (CA), *previously violated active*, and *currently inactive* (CI). The first two classes describe constraints that are either violated or have non zero Lagrange multipliers in the current solution of the LLBP. The third class consists of constraints that have a Lagrange multiplier λ_i and a subgradient δ_i both set to 0. Thus all constraints of the class CI do not have any effect on the subgradient optimization. For each constraint of that class the number of iterations since it became inactive is stored and the constraints are sorted according to this number. Every time new constraints are separated and the maximum size of the constraint pool is reached, constraints of the class CI which have been inactive for the most preceding iterations are replaced

by new constraints. Additionally, only constraints that have been inactive for more than two subsequent iterations can be replaced by new ones in this approach.

As a second key, constraints are also ordered by the number of jump arcs they contain i.e. the size of the jump J. Since the number of iterations a constraint has been inactive is mostly the same for many constraints the second level ordering is still meaningful. Clearly, a constraint with a small set of jump arcs is stronger than a constraint with a large number of jump arcs since requiring one arc out of a small set to be in the solution is a tighter restriction. Thus constraints which have been inactive for the same number of iterations are replaced in decreasing order of the number of contained jump arcs.

4.4.2 Grouping Constraints into Subsets

Using the method described in Section 4.3 to identify violated jump constraints, we obtain a large number of constraints that have a high degree of similarities. Those groups are induced by the node partitions. If one node partition contains k nodes in the set S_{H+1} and we replace it by k partitions with the set S_{H+1} being singleton, the resulting partitions induce constraints that only differ in a few jump arcs. By considering only one out of all these partitions with a singleton set S_{H+1}, the total number of constraints decreases considerable reducing the need for replacing other constraints while at the same time increasing the diversity in the constraint pool.

Chapter 5

Extensions and Hybrid Methods

5.1 Generalized Jump Constraints

The jump formulation from Section 3.2 uses jump constraints defined by (3.11). Those constraints can be interpreted in the following way: If we partition all nodes of the instance into $H+2$ non-empty sets where the root node r is in the first set S_0, any valid HCMST satisfying the hop constraint H for that instance must at least contain one arc that, in the partition of the nodes into $H+2$ sets, *jumps* over one set. If the HCMST would not contain such an arc there would be at least one path with $H+1$ arcs from the root node r to a node u in set S_{H+1}. A node partition as described above with the arcs of a valid HCMST are depicted in Figure 5.1.

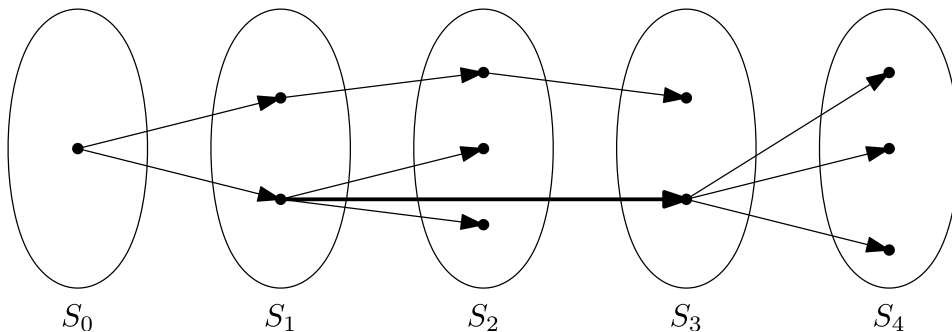


Figure 5.1: A node partition with 5 sets for a HCMST with hop constraint $H = 3$. Without the jump arc (bold) from S_1 to S_3 bypassing set S_2 the nodes in set S_4 would not be reachable without violating the hop constraint.

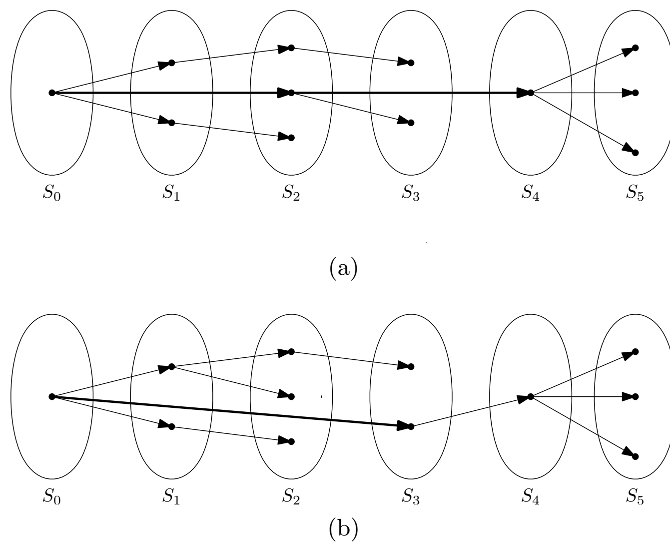


Figure 5.2: Node partitions with 6 sets for a HCMST with hop constraint $H = 3$.

5.1.1 The Jump-Formulation with Generalized Jump Constraints

The formulation of the jump constraints can be generalized to node partitions with $H+k$ sets as described in [DG04]. Suppose we partition all nodes of the instance into $H+3$ instead of $H+2$ non-empty sets, then any valid HCMST satisfying the hop constraint H must at least contain one arc that jumps over at least two sets or two arcs that jump over one set. This idea is depicted in Figure 5.2. In (5.2a) without the two jump arcs (bold) the nodes in set S_5 would not be reachable from r without violating the hop constraint. In (5.2b) only one jump arc is sufficient but it must jump over at least two sets. Let us now assign a value of 1 to any jump arc that jumps over one set, a value of 2 for jump arcs that skip two sets and so forth. Any non jump arc a value of 0 will be assigned to. Thus, we can say that for a constraint induced by a partition of all nodes into $H+k$ non-empty sets the values of the used jump arcs must sum up to a value $\geq k-1$. A jump J can now be easily written as a vector j in the formulation of the constraints. The values assigned to jump arcs are denoted as j_a in the vector j. This leads us to the following definition of the generalized jump constraints:

$$\sum_{a \in A} j_a x_a \geq b_J \qquad \forall J \in \Pi \tag{5.1}$$

with $b_J = k-1$ for a jump J that is based on a node partition with $H+k$ sets. Note that the classical jump constraints can be seen as a special case of the generalized jump constraints induced by $H+k$ sets with $k=2$. Relaxing the

generalized jump constraints and bringing them into the objective function of the integer linear program (3.12) leads to the following objective function of the Lagrangian relaxed program:

$$\min \sum_{a \in A} c_a x_a + \sum_{J \in \Pi} \lambda_J (b_J - \sum_{a \in A} j_a x_a) \tag{5.2}$$

which can also be written as:

$$\min \sum_{a \in A} c_a x_a + \sum_{J \in \Pi} \lambda_J b_J - \sum_{J \in \Pi} \left(\sum_{a \in A} j_a x_a \lambda_J \right) =$$

$$\min \sum_{a \in A} x_a \left(c_a - \sum_{J \in \Pi | j_a \geq 1} j_a \lambda_J \right) + \sum_{J \in \Pi} \lambda_J b_J \tag{5.3}$$

Generalized jump constraints can be identified and separated with the same procedure as explained in Section 4.3 whereas compression no longer is necessary but it still can – of course – be performed.

5.2 Creating an Initial Pool of Constraints

Since the number of jump constraints is exponential in the number of nodes it is not possible to add them all in advance to the model but they have to be separated dynamically from infeasible solutions. The subgradient algorithm starts with an empty set of relaxed jump cuts. Hence it seems promising to have an initial set of jump constraints before starting the subgradient algorithm. However, experiments have shown that starting with an initial set of jump inequalities that were obtained from previous runs of the subgradient algorithm does not yield to significantly better solutions or shorter computation times. On the other hand, using such a set along with corresponding Lagrange factors from a previous run lead to a noticeable improvement of the solutions.

Dahl et al. describe a method for creating an initial pool of jump constraints in [DFFG05] which is based on a dual ascent approach. However, their method relies on a certain subproblem for identifying appropriate jump inequalities:

Given a subgraph T of the instance graph, a jump constraint J has to be found such that $J \cap T = \emptyset$, i.e. the jump constraint J must not contain any arc already included in the subgraph. They claim that their problem is hard and must be solved via a time consuming call of an ILP solver. As a consequence they have only tested it for small instances.

This subproblem arises from the motivation of finding Lagrange factors along with the jump constraints. This is done by enforcing the primal complementary slackness condition and relaxing the dual complementary slackness condition.

In the further sections a similar algorithm will be described which uses a greedy heuristic to efficiently solve the mentioned subproblem. Before that, some fundamentals are presented.

5.2.1 Methods

Consider the following linear program

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax \geq b \end{aligned}$$

where c and x are n-dimensional vectors, A is an $m \times n$ matrix, and b is an m-dimensional vector. The ^{T} indicates a transposed matrix. This linear program has n variables and m constraints.

From this problem we can derive a new linear program:

$$\begin{aligned} \max \quad & \lambda^T b \\ \text{s.t.} \quad & A^T \lambda \leq c \end{aligned}$$

This new linear program is called the dual to the original (primal) problem. The two problems are related to each other in the following way: If the primal problem is bounded the dual problem is bounded as well and the values of the optimal solutions of both problems are equal. Additionally, we can see that there is a relation between the primal variables and the constraints of the dual as well as between the dual variables and the constraints of the primal: Each primal variable corresponds to one constraint in the dual problem, and each dual variable corresponds to one constraint in the primal problem. Furthermore, the following relations are given:

$$x_i > 0 \Rightarrow A_i^T \lambda = c_i \tag{5.4}$$

and

$$\lambda_j > 0 \Rightarrow A_j x = b_j \tag{5.5}$$

(5.4) is called the primal complementary slackness condition, (5.5) is called the dual complementary slackness condition. In words we can say:

1. *If $x_i > 0$, the ith constraint of the dual is binding.*
2. *If $\lambda_j > 0$, the jth constraint of the primal is binding.*

This information can be exploited by primal-dual algorithms for approximation problems.

5.2.2 A Dual-Ascent Algorithm

Several dual-ascent algorithms are presented in [GW97]. A brief description about a dual-ascent algorithm is given here since it forms the basis of the implemented method for generating an initial pool of jump constraints. For the ease of reading the formulations are presented with the same symbols as used for the jump formulation.

Let

$$\begin{aligned} \min \quad & \sum_{a \in A} c_a x_a \\ \text{s.t.} \quad & \sum_{a \in J} x_a \geq 1 \qquad J \in \Pi \\ & x_a \in \{0,1\} \qquad a \in A \end{aligned} \tag{5.6}$$

be an ILP, and

$$\begin{aligned} \min \quad & \sum_{a \in A} c_a x_a \\ \text{s.t.} \quad & \sum_{a \in J} x_a \geq 1 \qquad J \in \Pi \\ & 0 \leq x_a \leq 1 \qquad a \in A \end{aligned} \tag{5.7}$$

its corresponding LP relaxation. Then

$$\begin{aligned} \max \quad & \sum_{J \in \Pi} \lambda_J \\ \text{s.t.} \quad & \sum_{J : a \in J} \lambda_J \leq c_a \qquad a \in A \\ & 0 \leq \lambda_J \leq 1 \qquad J \in \Pi \end{aligned} \tag{5.8}$$

is the dual problem of its LP relaxation. Furthermore, let T be a set that is represented by x in the following way: If $x_a = 1$ then $a \in T$. In other words we can say that T is a – not necessarily feasible – solution for the primal problem. J is a constraint and can be understood as a set of elements of A of which at least one must appear in the solution T. Π is the set of all constraints.

Now let us consider a given primal infeasible solution T and a feasible dual solution λ. Obviously the infeasibility of T says that a constraint of the primal problem is violated. If T satisfies the primal complementary slackness condition for λ

$$a \in T \Rightarrow \sum_{J : a \in J} \lambda_J = c_a \tag{5.9}$$

then it is not possible to find a feasible primal solution still satisfying the primal complementary slackness condition for the same λ. That means

that there exists a constraint J_k with $T \cap J_k = \emptyset$. Hence we have to add an a to T that is contained in the violated constraint J_k. To achieve this without violating the primal complementary slackness conditions we increase the dual solution by increasing λ_{J_k} until $\exists a \in J_k$ such that the primal complementary slackness condition is satisfied for that a. Now a can be added to the set T. In this procedure the maximal value that can be assigned to λ_{J_k} is given by:

$$\min_{a \in J_k}(c_a - \sum_{J \neq J_k : a \in J} \lambda_J) \tag{5.10}$$

Exceeding that value would violate the feasibility of the dual solution and the primal complementary slackness condition for a. This procedure of finding a J_k, with $T \cap J_k = \emptyset$, increasing λ_{J_k} and adding a to T, can be repeated until T represents a feasible solution.

Several enhancements for this algorithm exist including the *reverse delete step*, or the *uniform increase rule*. However, those enhancements have not been applied in the implementation of the dual-ascent algorithm for the BDMST and HCMST problems and are left for further investigations. For more details about those enhancements the reader is referred to [GW97].

5.2.3 Utilizing the Dual-Ascent Algorithm for the BDMST and HCMST Problems

The method described above can be employed for the relax-and-cut approach for the BDMST and HCMST problems that is based on the jump formulation. Consider a simplified version of the jump formulation for the BDMST and HCMST problem that is obtained by ignoring the spanning-tree constraints and only keeping the jump constraints. The resulting ILP is exactly that one given by (5.6). Furthermore, we can utilize the dual variables λ of the dual problem (5.8) as Lagrange factors for the jump constraints, which are represented by J.

Starting without an initial solution, i.e. an empty set T, we try to find a jump constraint J with $J \cap T = \emptyset$ and set its corresponding dual variable (Lagrange factor) λ_J to the value of the cheapest arc $a \in J$ which in turn is added to the set T. Note that finding a jump constraint J with $J \cap T = \emptyset$ is only trivial for the first constraint where T is still empty. Before continuing, the costs of all arcs $a \in J$ are decremented by the value of λ_J. This step ensures that the value of the cheapest arc in subsequently found jump constraints is equal to the term (5.10) based on the original arc costs and so the primal complementary slackness condition is satisfied. However, to avoid a zero cost arc as the minimum cost arc in a newly identified jump constraint all minimal cost arcs of a jump constraint J are added to T.

The approach for generating an initial pool of jump constraints described in [DFFG05] is equivalent to the above described procedure. As mentioned before the central issue in applying this procedure is the part of finding a

jump constraint J such that $T \cap J = \emptyset$. Since a jump constraint is induced by a partition of nodes into $H+2$ non-empty sets where H is the height of the HCMST, it is too complex to directly select a set of arcs to appear in a jump constraint. Dahl et al. solve that problem via an ILP that in turn is solved by an appropriate solver. They claim that solving this problem is very time consuming.

Algorithm 2 createPool()

```
Input: Subgraph T, Solution S, Instance I
 1: T ← ∅
 2: while (constraintsFound) {
 3:   constraintsFound ← false
 4:   for each (arc ∈ S) {
 5:     partition ← createPartition(T, S, arc, I.root)
 6:     if (isValid(partition)) {
 7:       constraint ← storeConstraint(partition)
 8:       for each (arc ∈ constraint) { // add cheapest arcs of the new constraint to T
 9:         if (c(arc) = c(minArc)) {
10:           T ← T ∪ arc
11:         }
12:       }
13:       constraint.lambda ← c(constraint.minArc)
14:       for each (arc ∈ constraint) { // reduce arc costs of arcs contained in constraint
15:         c(arc) ← c(arc) - constraint.lambda
16:       }
17:       constraintsFound ← true
18:     }
19:   }
20: }
```

Another approach for solving this problem which is utilized in the algorithms presented in this section is based on a breadth first search (BFS). For simplicity we now consider T to be a set of arcs together with the nodes adjacent to these arcs. Considering the subgraph T of the instance graph, nodes are assigned to the sets in the partition according to their depth in a BFS on T starting at the root node, i.e. a node labeled with 2 is assigned to the set S_2 in the partition. Not that T does not necessarily have to be connected. If after the first BFS run some nodes of T are not yet labeled the BFS is restarted with the last label number incremented by 1 at one of the unlabeled nodes until the whole subgraph T is labeled. This approach is relatively efficient by means of time complexity but the total number of jump constraints that can be found is limited. Additionally, we do not only try to find jump constraints with $J \cap T = \emptyset$. We also want to force arcs that are part of a good heuristic solution as jump arcs into the constraints.

Based on an initial feasible solution S which was generated by the ACO, Algorithm 2 iterates over all solution arcs and tries to generate constraints containing arcs of the solution as jump arc until no constraints can be found at a complete loop over all solution arcs. The function isValid() checks if a partition generated by createPartition() with the mentioned BFS algorithm contains at least $H+2$ non-empty sets. If the number of sets in the partition is $< H+2$ no constraint can be created from the partition.

Algorithm 3 createPartition()

Input: Subgraph T, Solution S, Arc *arc*, Node *root*
Output: NodePartition partition

```
    // label nodes based on arcs in T
 1: max ← 0
 2: for (i = 0 ... MaxLoop) {
 3:    shuffle the order of nodes in T;
 4:    sort nodes in T according to their outdegree;
 5:    l ← BFS(T, root) // label nodes according to depth in a BFS started at root
 6:    if (l > max) {
 7:       store labels of this so long best run
 8:       max ← l
 9:    }
10: }
11: restore labels of the best found run
12: initialize partition
13: arrangeUnlabeledNodes(S, arc)
14: for each (v ∈ S) {
15:    partition.putNodeIntoSet(v, v.label)
16: }
17: return partition
```

In Algorithm 3 the node labeling based on a BFS is executed $MaxLoop$ times and the whole set of node labels with the highest label number is stored. This is done after sorting the nodes in T by their out-degree. The nodes are shuffled in advance such that the order of nodes with the same out-degree is different for each iteration. The motivation for repeatedly calling the node labeling algorithm with different node orders is the higher chance of finding a partition that has at least the minimal required number $(H+2)$ of sets. In practice, the loop at line 2 increases the total number of separated jump constraints since it can be possible that several attempts are needed to create a partition that has enough (at least $H+2$) sets. However, the benefit is limited, i.e. $MaxLoop$ cannot be rised arbitrary to further increase the total number of separated jump constraints. Also note that after the BFS some nodes may remain unlabeled and the highest label can be smaller than $H+1$ ($H+2 sets \rightarrow S_0 \dots S_{H+1}$). If the number of unlabeled nodes

is higher than $H + 1 - k$ where k is the highest label given to a node in the BFS, these unlabeled nodes can be assigned to different sets with an index $> k$ such that a valid partition of all nodes into $H + 2$ non-empty sets can be created. This is performed by the procedure depicted in Algorithm 4 by labeling the unlabeled nodes with the index of the appropriate sets. If, on the other hand, all nodes are labeled in the BFS, i.e. the subgraph T contains enough arcs such that every node is connected to another one with an arc contained in T, the maximal depth k must be at least $H + 1$ such that a partition inducing a jump constraint J with $J \cap T = \emptyset$ can be created.

Algorithm 4 arrangeUnlabeledNodes

```
Input: Solution S, Arc arc
1: for each (v ∈ S) {
2:    if (v is unlabeled by preceding BFS) {
3:       if (nodeInSubTree(v, arc) { // check if arc is on the path from root to v
4:          v.label ← S.level[v] +1
5:       } else {
6:          v.label ← S.level[v]
7:       }
8:    }
9: }
```

Algorithm 4 assigns a label to all nodes which are not present in T and therefore are not yet labeled. The check in line 3 causes the two nodes adjacent to *arc* to be labeled with numbers such that the label of the source node is smaller than the label of the target node minus 1, which in turn causes *arc* to be in the set of jump arcs. However, with an increasing number of nodes in the subgraph T the probability of forcing an arc contained in the heuristic solution to be a jump arc decreases.

5.3 A hybrid Algorithm Based on ACO and Relax-and-Cut

To improve the ACO algorithm with information obtained from the relax-and-cut approach we do not consider the node levels of a solution since the relax-and-cut approach only delivers meaningful information on the arcs.

The Lagrangian relaxation of the ILP formulation for the HCMST can be interpreted as follows: The costs of an arc a are reduced by the sum of the Lagrange multipliers of all separated jump constraints that contain arc a. The resulting reduced arc cost is denoted by α_a as defined in (3.17). These reduced arc costs provide some information about the potential of using an arc in a solution. The larger the sum of the Lagrange multipliers for one

arc a, the smaller will be the reduced arc cost α_a, and the more promising it will be to include this arc in the solution.

Now a hybrid version of the ACO algorithm can be developed using two graphs as input: The instance graph with the original arc costs for the evaluation of solutions and a modified version of that graph with reduced arc costs as a heuristic component for the construction of solutions.

5.3.1 Adding a Heuristic Value to the Ant Colony Optimization

The original ACO algorithm for the BDMST problem only uses the information from the pheromone matrix to build a solution in level representation which has to be decoded to get the final tree. As described in [Kop06] the probability $P_{v,l}$ for node v to be assigned to level l is defined as

$$P_{v,l} = \frac{\tau_{v,l}}{\sum_{l'=1}^{\lfloor \frac{D}{2} \rfloor} \tau_{v,l'}} \tag{5.11}$$

with $\tau_{v,l}$ being the pheromone value stored in the matrix. Now the goal is to introduce some heuristic information into that probability as described by Blum et al. in [BR03]. Let $H_{v,l}$ denote the heuristically computed benefit of assigning node v to level l. Then

$$P_{v,l} = \frac{\tau_{v,l}^{\beta} H_{v,l}^{\gamma}}{\sum_{l'=1}^{\lfloor \frac{D}{2} \rfloor} \tau_{v,l'}^{\beta} H_{v,l'}^{\gamma}} \tag{5.12}$$

is the probability for node v to be assigned to level l that is extended by the heuristic information $H_{v,l}$ where $\gamma = 1-\beta$. β and γ are weighting exponents that control how much the probability $P_{v,l}$ depends on the pheromone values or on the heuristic information.

5.3.2 Deriving a Heuristic Value from Arc Costs

As mentioned above the heuristic value $H_{v,l}$ describes the benefit of assigning node v to level l. This heuristic value could now be derived from the reduced arc costs of the relax-and-cut algorithm. Suppose we are decoding the node level information on the fly while assigning nodes to their levels. Each time a node v is assigned to a certain level l we can determine which of the previously assigned nodes in levels $j < l$ is the *cheapest predecessor*, i.e. the node u that can be used as predecessor in the tree with the cheapest arc (u, v). The costs of that arc (u, v) can be seen as a penalty (the connection of node v to the tree increases the objective value) and we are looking for the node u causing the smallest penalty when connecting node v to it.

Beside the penalty, assigning a node to a level can also produce a bonus. Again, when assigning node v to level l we now consider all previously inserted nodes at levels $k > l$. If node v is a cheaper predecessor for a node

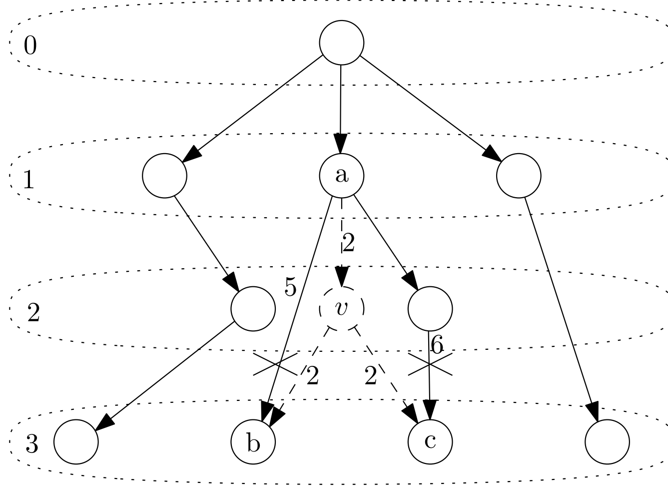


Figure 5.3: Concept of the penalty and bonus for the level assignment of node v.

w at a level $k > l$ currently connected to a node q at any level $< k$, node w can be disconnected from node q and be reconnected to node v. This in turns can be evaluated for all nodes w at a level $k > l$. Thus the bonus is the sum of the positive differences between arcs (q, w) and (v, w).

The concept of the bonus and penalty is depicted in Figure 5.3. When inserting node v at level 2 and the cheapest arc from a node at level 0 or 1 has costs of 2 the penalty is 2. Since two nodes (b and c) at level 3 can be reconnected to node v, while at the same time saving overall costs, the resulting bonus is $(5 - 2) + (6 - 2) = 7$.

We can define the penalty for assigning node v to level l as follows:

$$Q_{v,l} = \min_{u \in V' | lev(u) < l} c(u, v) \tag{5.13}$$

where V' is the set of nodes that are previously assigned to levels, $lev(u)$ is the level node u is assigned to, and $c(u, v)$ denotes the costs of the arc from u to v.

The bonus for assigning node v to level l can be defined as follows:

$$B_{v,l} = \sum_{w \in V' | lev(w) > l} \max\left(0, \left[\min_{q \in V' | lev(q) < lev(w)} c(q, w)\right] - c(v, w)\right) \tag{5.14}$$

Note that assigning the root node to level 0 does not result in any penalty or bonus. It is treated separately. The resulting heuristic value $H_{v,l}$ always has to be > 0. A value of 0 would result in a probability $P_{v,l} = 0$ in (5.12).

We can now define $H_{v,l}$ as:

$$H_{v,l} = \frac{1 + B_{v,l}}{Q_{v,l}} \tag{5.15}$$

5.3.3 Node Level Assignment with a Heuristic Component

To be able to compute $H_{v,l}$ efficiently for every node that is assigned to a certain level, temporary values must be stored for every node. With Algorithm 5 the assignment based on the probability $P_{v,l}$ defined by (5.12) can be performed efficiently. $c(v, w)$ denotes the reduced costs α_a of arc $a = (v, w)$, $lev(v)$ denotes the level of node v, and $pred(v)$ denotes the direct predecessor of node v in the tree. For the loops at lines 11 and 18 the obtained information is cached in *temppred* and *tempsucc* such that after assigning the node v to a certain level l the predecessors can be updated in an efficient way.

After assigning every node to its level, the predecessor information obtained during Algorithm 5 is discarded, and a new tree is build based on the node level information with the original decoding procedure based on the original arc costs described in [GvHR06].

5.3.4 The New ACO Algorithm

For the new ACO algorithm that employs the assignment of nodes to levels as described above, the whole pheromone matrix is initialized to $\tau_{v,l} = \frac{1}{|V| \cdot T_0}$ where T_0 is the objective value of an initial solution obtained with another heuristic algorithm such as RTC or CBTC. Based on the values from the pheromone matrix and the heuristic information $H_{v,l}$, which uses the reduced arc costs from the relax-and-cut algorithm instead of the original arc costs of the instance graph, nodes are assigned to levels. This level information must be decoded by the original decoding procedure of the ACO algorithm with the original arc costs. Every ant produces a solution in the described way and the best ant will deposit pheromone. For details about the pheromone evaporation and depositation process see [GvHR06].

The combination of the relax-and-cut algorithm with the ACO algorithm can be achieved with two strategies.

1. **Batch approach:** After the relax-and-cut algorithm converged to a solution, a graph with the reduced arc costs α_a of the best solution is taken as input graph for the heuristic component of ACO algorithm.

2. **Interleaved approach:** The ACO algorithm and the relax-and-cut algorithm run in parallel respectively intertwined and the reduced arc costs for the ACO are updated regularly with information from the relax-and-cut algorithm.

However, only the first strategy has been implemented as part of this work.

Algorithm 5 ComputeNodeLevels

```
1: lev(center) ← 0
2: levelList[0] ← {center}, levelList[l]← ∅      for l = 1 . . . H
3: V′ ← {center}
4: while (|V′| ≠ |V|) { // while there are unassigned nodes. . .
5:    v ← pop(V\V′)
6:    penalty ← ∞, bonus ← 0
7:    bonusforlevel[l] ← 0      for l = 1 . . . H
8:    temppred[0] ← center
9:    for (l = 1 . . . H) {
10:      temppred[l] = temppred[l − 1]
11:      for each (u ∈ levelList[l − 1]) { // search for the cheapest possible predecessor
12:         if (c(u, v) < penalty) {
13:            penalty ← c(u, v)
14:            tempred[l] ← u
15:         }
16:      }
17:      if (l = 1) {
18:         for (k = l + 1 . . . H) { // store all nodes assigned to level k in tempsucc[k] that can
            have node v as cheapest predecessor
19:            bonusforlevel[k]= 0
20:            tempsucc[k]← ∅
21:            for each (w ∈ V′ | lev(w) = k) {
22:               if (c(v, w) < c(pred(w), w)) {
23:                  bonus = bonus + c(pred(w), w) − c(v, w)
24:                  bonusforlevel[k] = bonusforlevel[k] + c(pred(w), w) − c(v, w)
25:                  tempsucc[k] ← tempsucc[k]∪ {w}
26:               }
27:            }
28:         }
29:      } else {
30:         bonus = bonus - bonusforlevel[l]
31:      }
32:      H_{v,l} ← (1+bonus)/penalty
33:   }
34:   lev(v) ← level for node v according to P_{v,l} = τ^β_{v,l} H^γ_{v,l} / ∑_{l=1}^{H} τ^β_{v,l} H^γ_{v,l}
35:   pred(v) ← temppred[lev(v)]
36:   for each (w ∈ tempsucc[k]| k > lev(v)) {
37:      pred(w) ← v
38:   }
39:   V′ ← V′ ∪ {v}
40:   levelList[lev(v)] ← levelList[lev(v)] ∪ {v}
41: }
```

Chapter 6

Implementation

The program `bdmstsolver` which was developed as part of this thesis is implemented as a command line application in C++ on a Linux system. It has its origin in the program developed by Putz [Put07]. It replaces several old classes from Putz' implementation and introduces some new ones. Additionally, many new algorithms have been included into existing classes and several data structures have been replaced by more efficient ones.

Another existing part that was integrated into the `bdmstsolver` is an implementation of the ACO algorithm written by Gruber [GvHR06]. Also in this implementation several changes have been performed.

6.1 Classes and Libraries

This section gives a brief overview about the classes used in the program. Note that not all classes are presented here, only the more important ones and those that have been added or significantly changed are mentioned.

Classes Representing the Solution

- `BDMST_Solution`

  This class describes an abstract interface of a solution to the BDMST problem. It contains a reference to a `BDMST_Instance` object.

- `Predecessor_Solution`

  The class `Predecessor_Solution` is an abstract base class representing a solution as a set of predecessor variables (for each node in the directed tree the direct predecessor is stored). Additionally, it contains a reference to a `Bidirected_Instance` object.

- `Predecessor_IntSolution`

  The class `Predecessor_IntSolution` is an implementation of the abstract class `Predecessor_Solution`.

Classes for Instances

- `Instance`

  This class comes from the library implemented by Gruber [GvHR06]. It represents an instance for the BDMST Problem. This class was extended by variables to optionally hold a predefined root node such that this class can either represent an HCMST or a BDMST instance.

- `BDMST_Instance`

  The class `BDMST_Instance` represents a simple BDMST instance containing an `Instance` object and providing some more functionality for the program `bdmstsolver` such as output procedures.

- `Bidirected_Instance`

  Derived from `BDMST_Instance` this class represents the instance as a bidirected graph providing the possibility of using algorithms that work on bidirected input graphs.

- `Bidirected_ArtificialRooted_Instance`

  This class is derived from `BDMST_Instance` and adds an artificial root node and artificial arcs from this root node to any other node in the instance. The uniform length of the artificial arcs is set to a multiple of the longest arc in the instance graph.

  This class also was changed in order to be able to represent both, BDMST and HCMST instances. For HCMST instances the node used as artificial root node is set to the original root node of the HCMST instance and a flag is set to indicate that the object represents a HCMST instance.

Classes for Handling Jump Constraints

- `Node_Partition`

  Jump constraints are induced by partitions of nodes into $H + 2$ nonempty sets. Since for the creation of jump constraints manipulations on the partitions such as making the last set singleton are necessary the class `Node_Partition` has been introduced to group all relevant manipulation methods for node partitions in one class.

- `Jump_Constraint`

  Jump constraints are represented by the class `Jump_Constraint`. It consists of a set of arcs (the jump arcs) and some functions needed for the subgradient optimization.

- `Jump_ConstraintFactory`

  This class contains several methods for generating jump constraints from given node partitions. Additionally, it contains some methods for creating node partitions from paths violating the diameter respectively hop constraint in an infeasible solution and checking their the validity of produced partitions.

- `Constraint_Comparator`

  The class `Constraint_Comparator` provides ordering functionality for jump constraints. With `Constraint_Comparator` objects a multilevel ordering of jump constraints can be performed. The original ordering is performed in the following hierarchy:

  Number of arcs, length of the shortest jump arc, average length of jump arcs and finally the identifier. While the former three criteria can be equal for several constraints, the identifier is unique for every jump constraint.

Classes with Implementations of Heuristic Algorithms

- `LevelConstructionHeuristics`

  The class `LevelConstructionHeuristics` implements the heuristic used to create feasible solutions from infeasible ones obtained by the LLBP solver. The heuristic is described in Section 3.2.3.

- `BDMST_UpperBoundCalc`

  This class provides a method to perform the computation of the upper bound based on different heuristic algorithms. Three heuristics are supported: CBTC, RTC (see Section 3.2.3), and ACO (see Section 4.1.1). For the CBTC and RTC heuristics a modification in this class has been necessary to support HCMST instances.

- `PheromoneMatrix`

  The class `PheromoneMatrix` is one of the core classes of the ACO algorithm implemented by Gruber. Modifications for the support of HCMST instances have been implemented in this class.

  The extensions to the ACO algorithm described in Section 5.3 are implemented by using an alternative method for assigning nodes to levels.

Classes for the ILP

- `Subgradient_Solver`

The subgradient algorithm as described in Section 4.2 is implemented in the class `Subgradient_Solver`. It contains a reference to the class `LLBP_Solver` which is an abstract base class for `Jump_Solver`.

- `Jump_Solver`

  This class provides the complete functionality for solving the relaxed ILP as described by Equation (3.15).
  It has a reference to a `Bidirected_ArtificialRooted_Instance` object and a `Predecessor_IntSolution` object. Additionally, it contains lists of previously and newly separated jump constraints.

6.1.1 Employed Third Party Libraries

For the ease of implementation several third party libraries have been employed:

- LEDA version 5.1.1

  For the representation of instance graphs and the solution the LEDA library (*Library of Efficient Data types and Algorithms*) is used. It is a C++ class library providing a large number of data structures and algorithms in the field of graph- and network problems (see [Led06]).

- GOBLIN version 2.7.2

  The GOBLIN graph library (*A Graph Object Library for Network Programming Problems*) contains an efficient implementation of Edmond's algorithm that is used to solve the minimum spanning arborescence in the LLBP solver (see [FPSE06]).

- Log4cpp version 0.3.5-rc3-1

  Log4cpp is a logging library for C++. In the program `bdmstsolver` this library controls the output (see [Bak05]).

6.2 New Data Structures and Algorithms

For efficiency reasons several data structures in the program based on Putz' implementation have been replaced such that more efficient processing of the contained data is possible. The following sections provide a brief overview of new data structures and algorithms introduced to the implementation of the `bdmstsolver`.

6.2.1 Constraint Sets and Parameters for Constraints

Originally jump constraints have been organized in a hash table based on string keys. Since a constraint is exactly described by the set of contained

jump arcs the concatenation of all strings representing the jump arcs is unique for a constraint if all arcs are processed in the same order for every jump constraint.

The problem with this approach is the poor performing behavior of the vast number of string operations required when generating constraints. Additionally, the more sophisticated handling of the constraint pool requires the constraints to be ordered in several hierarchical categories.

To overcome this bottleneck the constraints are organized in a sorted sequence provided by the LEDA library [Led06]. Therefore, a comparator object for every constraint is necessary which, besides other information for ordering, also contains a unique identifier for each constraint. Instead of a string representing a jump constraint a reference to the vector of jump arcs of the corresponding constraint is used as unique identifier with the benefit of both, dramatically reduced memory consumption and shorter computation time, especially in the process of creating constraints.

Not only the organization of the constraint pool itself has been redesigned, also the organization of the parameters used by the subgradient algorithm has been changed. Instead of organizing the parameters λ_J and δ_J (see Section 3.1.1) in a hash table with the constraint identifiers as key, those parameters have been relocated to the `Jump_Constraint` class such that every `Jump_Constraint` object contains its corresponding Lagrange factor λ_J and component of the subgradient δ_J.

6.2.2 Creation of Node Partitions

For the creation of node partitions with the path approach (see Section 4.3) and the approach described in Section 5.2.3 new algorithms have been implemented. When building a partition starting with a path from the root node r to a node v, with all nodes on the path being labeled according to their depth on the path, a recursive algorithm simply assigns every node the label of its predecessor with a special treatment for the level 0 and 1. This recursive implementation makes the algorithm very simple.

The implementation of the BFS algorithm for the creation of initial constraints is designed as an iterative algorithm using the classes `queue` and `node_array` provided by the LEDA library.

6.2.3 Solution and Jump Constraint Representation

The original implementation of the class `Predecessor_IntSolution` used the class `edge_array` for the representation of the predecessor variables. As a consequence, to completely describe a solution it was necessary to iterate over all arcs A and check if arc a is part of the solution. The time complexity of this procedure is $O(|V|^2)$ in complete graphs where $|V|$ is the number of nodes in the instance. As can be seen in Equation (3.7) for the computation

of the subgradient vector we have to iterate over the whole solution vector for each constraint. With the `edge_array` representing the solution as a vector of all instance arcs with values of 0 and 1 (a 1 indicates the arc is part of the solution), the time complexity for computing the complete subgradient vector is $O(|C||V|^2)$ where $|C|$ is the total number of jump constraints. To overcome this bottleneck two new data structures have been introduced:

- A list of all solution arcs for the solution objects which clearly has length $= O(|V|)$ for an instance with $|V|$ nodes.
- An `edge_array` for the constraints indicating for each arc of the instance if it is a jump arc.

As a consequence, for the computation of the subgradient vector δ we only have to iterate over all solution arcs instead of all arcs in the instance graph and check if the arc is a jump arc in a constraint. This reduces the time complexity for computing the complete subgradient vector to $O(|C||V|)$ while not affecting the time complexity of solution and constraint generation.

6.3 Usage

In the following a detailed overview about the options and parameters of the program `bdmstsolver` is given. The general syntax is:
`bdmstsolver -i <instance_file> [options]`
The options can be grouped in several categories:

General Options

- `-h, --help`

  Prints a message about the correct usage of the `bdmstsolver`.

- `-H, --version`

  Prints the version of the `bdmstsolver`.

Instance Options

- `-i <filename>, --instance <filename>`

  `<filename>` specifies the filename of the instance file.

- `-I <type>, --instance_type <type>`

  Specifies the type of the input file. The following values for `<type>` are supported:

  – `gnuplot` or `gp`

- santos or s
- gouveia or g
- ea
- rand_ea or r

- -d <diameter>, --diameter <diameter>

  The diameter of the instance. Note that only even numbers are accepted as diameter.

- -S <number>, --startnode <number>

  When specifying a root node, the instance will be interpreted as a HCMST instance with <number> being the index of the root node.

- -g <filename>, --gp_lines <filename>

  For GNUPLOT instances this option is used to specify the file holding the information about the edges. In this case the file specified by option -i only contains information about the nodes.

- -G <number>, --gouveia_edges <number>

  This option can be used to reduce the number of edges in a Gouveia instance.

Options for the Lagrangian Relaxation

- -l, --lifted

  Use generalized jump constraints as defined in Section 5.1.1.

- -j <strategy>, --jump_separation <strategy>

  Use this option to specify the strategy for the separation of jump constraints. For <strategy> the following values are supported: p for the path approach and l for the layered approach.

- -C <number>, --max_constraints <number>

  This option is used to specify the maximum size of the constraint pool.

Options for the Subgradient Optimization

- -m <number>, --maxIterations <number>

  Specifies the maximal total number of iterations of the subgradient algorithm before it terminates.

- -a <number>, --SG_baseAgility_TerminationLevel <number>

  This option specifies the agility level that – when reached – causes the subgradient algorithm to terminate. The default is 0.005.

- `-A <number>, --SG_baseAgility_ReductionAfterNoImprove <number>`

  Reduce the agility after `<number>` iterations without any improvement. The default is 30.

ACO Options

- `-O <number>, --aco <number>`

  Use the ACO as additional heuristic to compute an upper bound. If `<number>` is 0 the classical ACO is used to compute an initial upper bound. If `<number>` is 1 the hybrid version of the ACO algorithm is additionally used after the relax-and-cut algorithm has converged.

- `--acoAnts <number>`

  Specifies the number of ants to be used in the ACO algorithm. The default is 40.

- `--acoRho <number>`

  Specifies the pheromone decay coefficient for the ACO algorithm. The default is 0.003.

- `--acoIterations <number>`

  Specifies the number of iterations without any improvement as a termination condition for the ACO algorithm. The default is 1000.

- `--acotTimeLimit <number>`

  Specifies the total running time of the ACO algorithm in seconds. The default value is -1 which means no limit.

Miscellaneous Options

- `-u <method>, --upper_bound_method <method>`

  With this option it is possible to specify the heuristic used for the computation of an initial upper bound. The two possible values are `rtc` and `cbtc` (see Section 3.2.3). By default both methods are employed and the best upper bound is used.

- `-U <number>, --upper_bound_iterations <number>`

  Specifies the number of iterations for the calculation of the initial upper bound computed by the heuristics RTC and CBTC. The default is 100.

- `-Z, --onlyUB`

  Only compute an upper bound and do not start the relax-and-cut algorithm for computing a lower bound.

- `-r, --relax`

  Relax an initial set of constraints before starting the relax-and-cut algorithm (see Section 5.2).

- `-V <number>, --lev_nh_switch <number>`

  This option specifies the percentage of nodes at which, when decoding level information to the tree representation, the algorithm switches from level lists to neighbor lists to identify the cheapest possible predecessor (see [GvHR06]). This affects the VND used in the RTC and CBTC algorithms. Note that the VND used in the ACO algorithm is not affected by this parameter in the implementation. The default is 0.02.

Output Options

- `-o <prefix>, --outputPrefix <prefix>`

  Specifies the output prefix for all files.

- `-w (yes|no), --writheMinArbosGnuplot (yes|no)`

  Write the minimum arborescence to a file in gnuplot format at each iteration. The default is `no`.

- `-W (yes|no), --writheMinArbosGoblin (yes|no)`

  Write the minimum arborescence to a file in GOBLIN format at each iteration. The default is `no`.

- `-L <file>, --logrc <file>`

  Specify the path to the log4cpp configuration file. The default is `log4cpp.properties`

Chapter 7

Computational Results

To investigate the achievements of the various developments computational experiments have been performed on benchmark instances previously used in literature. These instances have been taken from three different sources: The *TE* and *TC* instances with 40 and 80 nodes and the *estein* have been taken from Beasley's OR-Library [Bea90], while *TE* and *TR* instances with 20 and 30 nodes have been published by Gouveia et al. [GM03]. The *c* and *g* instances have been originally published by Santos et al. [dSLR04]. The computations have been performed on a HP ProLiant ML110 G4 server with an Intel Xeon 3040 CPU with 1.86 GHz and 2 MB cache and 2.5 GB of RAM. The obtained results have been compared to results published by Dahl et al. [DFFG05] and to the results obtained with the implementation of Putz described in [Put07].

In the following sections a detailed overview about the experiments and their results are given. Every experiment has been performed 30 times and the arithmetic mean values and standard deviations (denoted as *sdev* in the tables) are presented for lower and upper bounds. Values for the computation time are arithmetic mean values too.

Parameters used for the computations which were chosen for all experiments have been selected as follows:

- Parameters for subgradient optimization:

  The number of iterations without any improvement after which the agility is reduced was set to 30. For the factor to reduce the agility 0.5 was used and the agility level used as termination condition was set to 0.005. These values are standard values suggested by Beasley in [Bea93].

- Parameters for the constraint pool:

  The maximal pool size was limited to 1000.

- Parameters for the ACO algorithm:

  The number of ants was set to 40, the iteration limit was set to 1000 and the value of the pheromone decay coefficient ρ was set to 0.003. No time limit was used for the ACO algorithm.

7.1 Results for Small BDMST Instances

The instances for which results are presented in this section are relatively small (20 - 40 nodes) and most of them do not describe complete graphs. Therefore, the ACO algorithm could not be employed to compute upper bounds for them since it requires a complete graph to ensure a tree can be built out of the level information. The *TE* and *TR* instances describe dense graphs, while the g instances describe sparse graphs and the c instances are complete. In Table 7.1 *NDRC* denotes the non-delayed relax-and-cut algorithm described in Chapter 4, and *DRC* the algorithm presented by Putz in [Put07].

As easily can be seen in these results, the new non-delayed relax-and-cut approach with all enhancements outperforms the original delayed relax-and-cut approach by far. The computed upper bound is the optimal value for every instance in almost every of the 30 runs. For six instances the computation of the lower bound could be stopped because the difference between the lower and upper bounds was less than 1 which means that the subgradient optimization has converged to the optimal solution (all instances have only integer costs assigned to the edges). Note that the reason for the large differences in computation time is not only based on the more efficient data structures but also on very inefficient output procedures for logging in the implementation of Putz.

7.2 Benefit of the Constraint Pool Management

As described in Section 4.4 constraints can also be deleted to be replaced by newly separated ones if the constraint pool has reached a maximum size. In the following the benefit of the sophisticated management of the constraint pool is demonstrated. For comparison the computations have been executed with the same algorithm but with a different handling of the constraint pool. These experiments were performed on medium sized instances (complete graphs with 41 and 81 nodes and edge costs corresponding to the Euclidean distances taken from Beasley's OR-Library [Bea90]).

In Table 7.2 the right results where obtained by the non-delayed relax-and-cut algorithm with all enhancements described in Chapter 4. The results on the left hand side have been computed with the same algorithm but the constraint pool was modified to the following behaviour: Instead of only replacing jump constraints that have been inactive for more than two

Instance					*NDRC*					*DRC*		
T	$\|V\|$	$\|E\|$	D	opt	LB		UB		Time	LB	UB	Time
					mean	sdev	mean	sdev				
TE	20	100	4	369	365.43	1.27	*369.00	0	7.5	321.53	370	364.9
TE	20	100	6	322	320.07	0.47	*322.00	0	5.5	305.25	332	498.8
TE	20	100	8	308	†307.05	0.04	*308.00	0	2.7	301.96	308	738.6
TE	30	200	4	599	555.31	4.91	*599.13	0.72	20.5	456.51	599	2151.2
TE	30	200	6	482	452.50	1.31	*483.53	2.92	11.6	420.34	491	1358.4
TE	30	200	8	437	419.16	0.42	*437.00	0	7.9	409.37	437	4448.4
TR	20	100	4	233	204.77	2.08	*233.00	0	8.8	192.79	233	563.8
TR	20	100	6	178	165.16	0.62	*178.13	0.84	6.3	164.77	178	1626.9
TR	20	100	8	154	151.98	0.08	*154.00	0	2.4	152.00	154	654.4
TR	30	200	4	234	182.54	2.57	*234.00	0	14.6	170.99	234	1074.5
TR	30	200	6	157	144.67	0.31	*157.00	0	8.7	139.63	157	963.8
TR	30	200	8	135	†134.14	0.11	*135.00	0	1.5	†134.31	135	21.3
c	20	190	4	349	344.21	0.70	*349.00	0	10.0	300.39	349	1642.3
c	20	190	6	298	285.92	0.47	*298.17	0.38	7.4	271.53	299	4617.8
c	20	190	10	324	†323.00	0.07	*324.00	0	4.6	319.20	332	1685.9
c	25	300	4	500	487.22	1.94	*500.00	0	21.7	411.97	500	4737.7
c	25	300	6	378	371.49	0.92	*378.07	0.37	15.5	354.60	378	5693.6
c	25	300	10	379	376.27	0.51	*379.80	1.00	8.1	369.48	379	2351.1
g	20	50	4	442	435.68	1.13	*442.00	0	4.5	389.03	446	406.5
g	20	50	6	329	†328.27	0.22	*329.00	0	0.7	307.00	329	316.8
g	20	50	10	359	†358.29	0.23	*359.00	0	0.3	†358.30	359	0.8
g	40	100	4	755	734.40	1.22	*755.00	0	8.5	595.54	755	768.8
g	40	100	6	599	†598.25	0.20	*599.13	0.5	3.1	584.67	599	209.5
g	40	100	10	574	572.97	0.06	*574.00	0	3.3	571.50	574	367.7

Table 7.1: The *non-delayed relax-and-cut* (NDRC) approach with all enhancements presented in this thesis vs. the *delayed relax-and-cut* (DRC) approach presented by Putz in [Put07]: T denotes the type of the instance (TE: Euclidean, TR: random, c: complete Euclidean, and g: sparse Euclidean). The number of nodes ($|V|$), the number of edges ($|E|$), and the diameter bound (D), as well as the optimal objective value (opt) are listed. For the two approaches the lower (LB) and upper bounds (UB) together with the running times in seconds are given.

*In most or all of the 30 runs the optimal solution was found.

†The computation stopped because the lower bound was greater than UB $-$ 1 i.e. the optimal solution was identified.

Instance			*simplified pool management*					*full pool management*				
T	$\|V\|$	D	LB		UB		Time	LB		UB		Time
			mean	sdev	mean	sdev		mean	sdev	mean	sdev	
tc40	41	4	658.43	7.60	747.00	0	78.5	673.46	6.41	747.00	0	71.5
tc40	41	6	535.64	2.02	607.07	1.01	45.3	538.23	2.30	606.80	1.00	37.9
tc40	41	8	503.34	1.03	544.57	0.50	34.6	504.45	0.84	544.50	0.51	30.8
tc40	41	10	491.39	0.73	516.00	0	30.0	492.56	0.33	516.00	0	29.7
tc80	81	4	1086.72	17.48	1303.00	0	364.3	1123.44	9.63	1303.00	0	385.6
tc80	81	6	865.64	2.34	1078.60	3.33	152.0	869.76	2.77	1078.73	2.86	130.3
tc80	81	8	842.90	2.29	980.50	1.20	127.6	846.34	1.33	980.23	0.90	132.9
tc80	81	10	834.75	0.71	924.70	0.84	105.9	835.99	0.72	924.67	0.66	111.7
te40	41	4	665.57	7.65	742.00	0	93.8	683.08	6.78	742.00	0	92.8
te40	41	6	542.58	2.22	606.00	0	42.0	545.04	1.90	606.00	0	40.0
te40	41	8	522.91	0.91	562.10	0.55	36.4	524.14	0.80	562.50	1.14	33.1
te40	41	10	513.57	0.84	537.00	0	35.5	514.76	0.73	537.03	0.18	37.2
te80	81	4	1682.93	30.43	2045.00	0	399.4	1751.54	24.49	2045.00	0	470.0
te80	81	6	1280.36	7.18	1565.63	2.03	183.3	1292.95	6.01	1566.00	1.76	173.7
te80	81	8	1204.20	2.23	1402.13	1.25	148.4	1211.26	2.44	1402.80	2.22	144.3
te80	81	10	1178.77	1.80	1299.30	3.93	157.2	1181.76	1.20	1301.80	5.14	142.2

Table 7.2: The NDRC algorithm with a simplified (left) and with the full constraint pool management (right): Experiments have been performed for complete BDMST instances with 41 and 81 nodes.

subsequent iterations all inactive jump inequalities can be replaced by new ones. Additionally, they are not ordered by the number of iterations they have been inactive but only by the number of jump arcs they contain and the costs of the cheapest jump arc.

Clearly, simplifying the management in the way described above significantly reduces the quality of the lower bounds as can be seen in the table. This demonstrates how sensitive the relax-and-cut algorithm is regarding to the parameters for the management of the jump constraint pool.

7.3 Results for HCMST Instances

To compare the relax-and-cut approach presented in this thesis with the implementation of Dahl et al. [DFFG05] which is a delayed relax-and-cut algorithm experiments have been performed on the same HCMST instances. For these computations the constraint pool size was limited to 1000.

Table 7.3 summarizes the results for these experiments on the HCMST instances from Beasleys OR-Library [Bea90]. Due to the utilization of the ACO heuristic the upper bounds are significantly better than Dahl's upper bounds except for the tc 40 instances where they are more or less equal. The lower bounds do not significantly differ as determined with a Wilcoxon signed-rank test. For the statistical test the lower bounds where rounded up to the next integral value since the instances only contain integral edge costs. Optimal values have been taken from [DGR06]. Note that the computation times cannot be directly compared since Dahl et al. performed their exper-

Instance				*NDRC*					*DRC ([DFFG05])*		
T	$\|V\|$	H	opt	LB		UB		Time	LB	UB	Time
				mean	sdev	mean	sdev				
tc	40	3	609	594.01	1.07	609	0	17.5	595	611	42
tc	40	4	548	539.07	0.79	548	0	16.2	541	549	40
tc	40	5	522	513.12	0.64	522	0	15.9	515	522	38
tc	40	6	-	497.03	0.07	498	0	8.7	498	498	11
tc	40	7	-	489.38	0.30	490	0	3.6	490	490	0.72
tc	40	8	-	487.21	0.22	488	0	4.5	488	488	2.8
tc	40	9	-	482.87	0.03	484	0	6.8	484	484	5.6
tc	40	10	-	481.41	0.33	482	0	3.5	482	482	0.07
tc	80	3	1072	1003.72	3.46	1078.30	2.04	102.3	986	1120	131
tc	80	4	981	897.09	2.19	982.57	0.50	88.4	901	1037	117
tc	80	5	922	862.23	1.62	925	1.78	84.1	858	971	119
tc	80	6	-	843.69	0.75	887.87	3.56	70.7	843	923	120
tc	80	7	-	836.79	0.28	864.73	1.11	57.5	838	886	107
tc	80	8	-	835.10	0.26	847.87	0.51	50.0	836	854	105
tc	80	9	-	834.18	0.81	838	0	54.7	834	838	99
tc	80	10	-	830.09	0.26	834	0	24.8	831	834	100
te	40	3	708	650.03	4.61	708	0	29.2	649	725	38
te	40	4	627	572.46	2.60	627	0	31.7	569	668	87
te	40	5	590	539.26	1.02	592.53	1.07	26.7	541	625	76
te	40	6	-	526.05	0.65	567.1	1.30	25.9	527	586	35
te	40	7	-	513.70	0.58	544.8	1	23.7	517	552	79
te	40	8	-	508.53	0.62	536	0	22.8	511	538	76
te	40	9	-	505.84	0.38	528	0	20.2	511	534	73
te	40	10	-	503.79	0.34	520	0	18.5	506	530	74

Table 7.3: Results for the NDRC algorithm (left) and the DRC algorithm described in [DFFG05] (right): The tc and te instances describe complete graphs with edge costs equal to the Euclidean distances. For the tc instances the root node is located in the center of the graph, while the root node of the te instances is located on the fringe of the graph. Note that $|V|$ denotes the number of nodes without the root node and H denotes the hop limit.

iments on a 2.0 GHz Pentium CPU which is not as powerful as the Xeon CPU used for the computations of the non-delayed relax-and-cu approach presented in this work.

7.4 Extensions of the R&C Approach

7.4.1 Initial Constraint Pool

In Section 5.2 a method for identifying a set of constraints with corresponding Lagrange multipliers before the subgradient optimization starts was presented. To evaluate the benefits of this development tests have been performed on larger instances taken from Beasley's OR-Library [Bea90]. These instances consist of 80, 90 and 100 nodes in the unit square and have edge costs corresponding to the Euclidean distances.

For 80-node instances the number of initially identified constraints was relatively small. On the other hand for all instances it can be observed

Instance			*With I.C.*						*Without I.C.*		
T	$\|V\|$	D	LB		pool size		Time	Diff.	LB		Time
			mean	sdev	mean	sdev			mean	sdev	
estein	80	6	7.3606	0.0367	950	46.10	177.9	0.0497	7.3109	0.0319	174.6
estein	80	8	6.8513	0.0139	706	69.80	154.2	0.0093	6.8420	0.0111	142.7
estein	80	10	6.6643	0.0081	601	31.20	135.0	0.0020	6.6623	0.0063	130.7
estein	90	6	7.2768	0.0269	1000	0.00	237.2	0.0460	7.2308	0.0244	248.7
estein	90	8	6.8163	0.0146	938	45.50	194.2	0.0144	6.8019	0.0155	195.9
estein	90	10	6.6358	0.0089	775	33.70	181.8	0.0012	6.6346	0.0063	177.2
estein	100	6	7.6186	0.0421	1000	0.00	329.3	0.0435	7.5751	0.0465	316.9
estein	100	8	7.1030	0.0109	998	8.70	256.6	0.0054	7.0976	0.0103	256.8
estein	100	10	6.9029	0.0084	902	72.50	237.7	-0.0001	6.9030	0.0066	232.6

Table 7.4: Comparison of the lower bounds for BDMST instances computed with (left) and without (right) identifying initial jump constraints.

T	$\|V\|$	D	*Original ACO*		*Hybrid ACO*	
			mean	sdev	mean	sdev
estein	100	8	8.3911	0.0000	8.6088	0.0392
estein	100	10	7.8528	0.0067	8.1583	0.0195
estein	100	12	7.5485	0.0013	7.7849	0.0227
estein	100	14	7.3456	0.0055	7.5271	0.0217
estein	250	8	14.5659	0.0842	15.0912	0.0793
estein	250	10	13.4172	0.0317	14.0329	0.0492
estein	250	12	12.6901	0.0384	13.3669	0.0552
estein	250	14	12.1801	0.0190	13.0112	0.0313

Table 7.5: Upper bounds computed for compete BDMST instances with 100 and 250 nodes by the original and the hybrid ACO algorithm.

that the number of initial constraints is higher in case the diameter bound is tighter. Table 7.4 also shows a comparison of the lower bounds. Unsurprisingly, the approach with the initial identification of jump constraints leads to tighter lower bounds whereas a significant improvement according to running times cannot be noticed . As determined with a Wilcoxon rank sum test for the individual instances, the initial constraint pool especially improved the lower bounds for instances with small diameters.

7.4.2 Hybrid ACO Algorithm and Generalized Jump Constraints

For both, the hybrid ACO algorithm and the utilization of the generalized jump constraints, the results are relatively clear. Neither of the two approaches resulted in any improvements.

The hybrid ACO algorithm could not reach the value of the original one from [GvHR06] (which makes no use of a heuristic component during the assignment of nodes to levels) in any of 80 experiments for complete instances with 100 and 250 nodes. In fact there was a significant gap between the values of the original and the hybrid algorithm as shown in Table 7.5.

Also using the generalized jump constraints in the relax-and-cut algo-

rithm has shown a very poor performance. The main point is that the R&C approach already has problems to converge to a solution that satisfies all separated simple jump constraints. Therefore, the expected improvement was limited, not last to the fact, that the reduction of the solution space by using generalized jump inequalities is small in contrast to the simple ones. Preliminary tests have confirmed these assumptions.

Chapter 8

Conclusions

In this thesis an existing relax-and-cut algorithm for the BDMST problem has been extended to also provide solutions for the HCMST problem. In addition, it has been enhanced in several ways. Computational experiments have been performed on BDMST and HCMST instances and results were compared to previously published ones for both problems.

The basic ILP model to solve the problem makes use of so-called jump constraints to ensure the diameter respectively the hop bound. Since the number of jump constraints is exponential they have to be separated dynamically. The original delayed relax-and-cut algorithm based on Lagrangian relaxation presented in [Put07] to compute high quality lower bounds has been changed to a non-delayed relax-and-cut algorithm. While in the delayed R&C algorithm violated jump constraints are only identified and relaxed after the subgradient optimization has converged to some degree, the non-delayed R&C algorithm identifies constraints at every iteration of the subgradient optimization. In addition, another method for identifying constraints was employed and a sophisticated management of the pool of jump constraint was developed to limit the number of jump inequalities to be handled simultaneously. The mentioned enhancements and the utilization of a metaheuristic, namely the ant colony optimization to compute upper bounds, have lead to significant improvements compared to the original algorithm. With the new approach significantly tighter lower bounds are obtained and computation times are by far lower. Computational experiments have also shown that the management of the constraint pool has a great impact on the quality of lower bounds.

In addition to those enhancements, several extension have been developed. Since the relax-and-cut algorithm identifies jump constraints during the subgradient optimization it starts in general without any constraints. Therefore, a method for identifying promising constraints in advance was developed that does not rely on infeasible candidate solutions obtained during the runtime of the subgradient optimization. This identification of an

initial constraint pool is based on a dual ascent algorithm. While for small instances the initial pool of constraints does not have any effect on the lower bounds a slight benefit of this extension can be seen for larger instances with tight diameter bounds. Although this extension results in longer running times the overall penalty in computation time is relatively small.

The other two extensions developed in this thesis are the modification of the model for using generalized jump constraints and the development of a hybrid version of a previously existing metaheuristic, namely an ant colony optimization (ACO).

The concept of the original ACO algorithm is the assignment of nodes to levels according to a probability given by pheromone values in a positive feedback system. A valid BDMST or HCMST is built from this node level information using a decoding procedure. The hybrid version determines this probability not only based on pheromone values but also on heuristic information obtained by the relax-and-cut algorithm, while the decoding procedure itself remains unchanged. Unfortunately, experiments have shown that solutions obtained with the hybrid ACO algorithm are not as good as solutions from the original ACO.

The approach using generalized jump constraints has shown a very poor performance in preliminary tests and therefore has not been subject to exhaustive experiments.

To conclude, all enhancements developed for the relax-and-cut algorithm have improved its performance and the quality of the lower as well as upper bounds. Also the extension with an initial pool of jump constraints had a positive effect on the lower bounds obtained by the relax-and-cut algorithm. However, the hybrid ACO algorithm and the employment of the generalized jump constraints did not lead to any improvements.

8.1 Future Work

Results of this work suggest that there is still room to further improve the presented algorithms. The dual ascent algorithm for the initial identification of jump constraints could be modified such that not every identified constraint is directly added to the constraint pool. It could be possible to iteratively identify sets of constraints and only select the tightest constraint out of one set to be added to the constraint pool. Therefore, the tightness could be measured using the number of jump arcs and the cost of the minimum cost jump arc in the constraint. Also the *uniform increase rule* or *reverse delete step* (see [GW97]) could be adopted for this algorithm.

Probably the hybrid ACO algorithm could be enhanced such that it computes results better than that of the original ACO algorithm. A possible approach would be to include a more sophisticated weighting of the heuristic values that – besides the pheromone values – form the basis for the proba-

bilistic assignment of nodes to levels. The heuristic values computed with the described method in this thesis might not be meaningful enough for the first nodes assigned to levels. Thus, it could be promising to increase the weights of the heuristic values for nodes that are inserted after most of the other nodes have already been assigned to levels.

Regarding the relax-and-cut algorithm it could be interesting to replace the classical subgradient optimization with the volume algorithm (see [BA00]). It could also be interesting to extend the management of the constraint pool since the relax-and-cut algorithm has shown to be relatively sensitive regarding the management of this pool.

Bibliography

[AC90] N. R. Achuthan and L. Caccetta. Models for vehicle routing problems. In *Proceedings of the 10th National Conference of the Australian Society for Operations Research*, pages 276–294, 1990.

[ACCG94] N. R. Achuthan, L. Caccetta, P.A. Caccetta, and J.F. Geelen. Computational methods for the diameter restricted minimum weight spanning tree problem. *Australasian Journal of Combinatorics*, 10:51–71, 1994.

[ADG00] Ayman Abdalla, Narsingh Deo, and Pankaj Gupta. Random-tree diameter and the diameter -constrained mst, 2000.

[BA00] Francisco Barahona and Ranga Anbil. The volume algorithm: producing primal solutions with a subgradient method. *Mathematical Programming*, 87(3):385–399, 2000.

[Bak05] Bastiaan Bakker. Log for c++, version 0.3.5-rc3-1, 2005. `http://log4cpp.sourceforge.net/`.

[Bea90] John E. Beasley. Or-library., 1990. Last update: February 2008 `http://people.brunel.ac.uk/~mastjjb/jeb/info.html`.

[Bea93] John E. Beasley. *Lagrangean Relaxation*, pages 243–303. Modern heuristic techniques for combinatorial problems. John Wiley & Sons, Inc., New York, NY, USA, 1993.

[BK91] Abraham Bookstein and Shmuel T. Klein. Compression of correlated bit-vectors. *Information Systems*, 16(4):387–400, 1991.

[BR03] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv.*, 35(3):268–308, 2003.

[CFG01] Teodor Gabriel Crainic, Antonio Frangioni, and Bernard Gendron. Bundle-based relaxation methods for multicommodity capacitated fixed charge network design. *Discrete Appl. Math.*, 112(1-3):73–99, 2001.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition edition, 2001.

[DFFG05] Geir Dahl, Truls Flatberg, Njål Foldnes, and Luis Gouveia. Hop-constrained spanning trees: The jump formulation and a relax-and-cut method. Technical report, Centre of Mathematics for Applications and Department of Informatics. University of Oslo, Norway., e-mail:geird@math.uio.no, September 2005.

[DG97] Marco Dorigo and Luca Maria Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, April 1997.

[DG04] Geir Dahl and Luis Gouveia. On the directed hop-constrained shortest path problem. *Operations Research Letters*, 32(1):15–22, 2004.

[DGR06] Geir Dahl, Luis Gouveia, and Cristina Requejo. *On Formulations and Methods for the Hop-Constrained Minimum Spanning Tree Problem*, chapter 19. Spinger Science + Business Media, New York, 2006.

[Dor92] Marco Dorigo. *Optimization, learning and natural algorithms*. PhD thesis, Politecnico di Milano, Italy, 1992.

[dSLR04] Andréa C. dos Santos, Abílio Lucena, and Celso C. Riberio. Solving diameter constrained minimum spanning tree problems in dense graphs. In *Proceedings of the International Workshop on Experimental Algorithms*, volume 3059 of LNCS, pages 458–467. Springer, 2004.

[FPSE06] Christian Fremuth-Paeger, Bernhard Schmidt, and Birk Eisermann. Goblin, a graph object library for network programming problems, 2006. `http://www.math.uni-augsburg.de/~fremuth/goblin.html`.

[Gib85] Alan Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, 1985.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[GM03] Luis Gouveia and Thomas L. Magnanti. Network flow models for designing diameter-constrained minimum-spanning and steiner trees. *Networks*, 41(3):159–173, 2003.

[GMR04] Luis Gouveia, Thomas L. Magnanti, and Christina Requejo. A 2-path approach for odd-diameter-constrained minimum spanning and Steiner trees. *Networks*, 44(4):254–265, 2004.

[Gou96] Luis Gouveia. Multicommodity flow models for spanning trees with hop constraints. *European Journal of Operational Research*, 95(1):178–190, November 1996. available at `http://ideas.repec.org/a/eee/ejores/v95y1996i1p178-190.html`.

[GR05] Martin Gruber and Günther R. Raidl. A new 0–1 ilp approach for the bounded diameter minimum spanning tree problem. In *Proceedings of the 2nd International Network Optimization Conference*, volume 1, pages 178–185, January 2005.

[GSU07] Luis Gouveia, Luidi Simonetti, and Eduardo Uchoa. Modelling the hop-constrained minimum spanning tree problem over a layered graph. In *INOC: International Network Optimization Conference, Spa, Belgium*, 2007.

[GvHR06] Martin Gruber, Jano van Hemert, and Günther R. Raidl. Neighbourhood searches for the bounded diameter minimum spanning tree problem embedded in a VNS, EA, and ACO. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1187–1194, New York, NY, USA, 2006. ACM.

[GW97] Michel X. Goemans and David P. Williamson. *The Primal-Dual Method for Approximation Algorithms and its Application to Network Design Problems*, chapter 4. Approximation Algorithms. Course Technologyn, MA, 1997.

[JR03] Bryant A. Julstrom and Günther R. Raidl. A permutation-coded evolutionary algorithm for the bounded-diameter minimum spanning tree problem. In A. Barry, F. Rothlauf, and D. Thierens et al., editors, *Genetic and Evolutionary Computation Conference's Workshops Proceedings, Workshop on Analysis and Design of Representations*, pages 2–7, 2003.

[Jul04] Bryant A. Julstrom. Greedy heuristics for the bounded-diameter minimum spanning tree problem. Technical report, St. Cloud State University, St. Cloud, MN 56301 USA, 2004.

[Kop06] Boris Kopinitsch. An ant colony optimisation algorithm for the bounded diameter minimum spanning tree problem. Master's thesis, Technische Universität Wien, January 2006.

[Led06] Leda. library of efficient data types and lgorithms, version 5.1.1, 2006. `http://www.algorithmic-solutions.com/leda/index.htm`.

[Luc05] Abílio Lucena. Non delayed relax-and-cut algorithms. *Annals of Operations Research*, 140(1):375–410, November 2005.

[Put07] Peter Putz. Subgradient optimization based lagrangian relaxation and relax-and-cut approaches for the bounded-diameter minimum spanning tree problem. Master's thesis, Vienna University of Technology, October 2007.

[RJ03] Günther R. Raidl and Bryant A. Julstrom. Greedy heuristics and an evolutionary algorithm for the bounded-diameter minimum spanning tree problem. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 747–752, New York, NY, USA, 2003. ACM.

[WA88] Kathleen A. Woolston and Susan L. Albin. The design of centralized networks with reliability and availability constraints. *Comput. Oper. Res.*, 15(3):207–217, 1988.

[Wol98] Laurence A. Wolsey. *Integer Programming*. Wiley-Interscience, 1998.