Technische Universität Wien

Vienna University of Technology

DISSERTATION

Exact and Heuristic Approaches for Solving the Bounded Diameter Minimum Spanning Tree Problem

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften unter der Leitung von

ao. Univ.-Prof. Dr. Günther Raidl
Institut für Computergraphik und Algorithmen E186
Technische Universität Wien

und

ao. Univ.-Prof. Dr. Ulrich Pferschy
Institut für Statistik und Operations Research
Universität Graz

eingereicht an der Technischen Universität Wien
Fakultät für Informatik von

Mag. DI Martin Gruber
Matrikelnummer 9025080
Hietzinger Hauptstraße 97/10, 1130 Wien

Wien, am 19. Mai 2009

Martin Gruber

Kurzfassung

Das Finden eines *durchmesserbeschränkten minimale Spannbaumes* (bounded diameter minimum spanning tree, BDMST) ist ein kombinatorisches Optimierungsproblem aus dem Bereich des Netzwerkdesigns und hat Anwendungen in verschiedensten Bereichen. So unter anderem beim Entwurf von kabelgebundenen Kommunikationsnetzwerken, sofern gewisse Anforderungen hinsichtlich der Kommunikationsgüte erfüllt werden sollen. Um die Wahrscheinlichkeit für das Auftreten von Störungen möglichst gering zu halten, soll zum Beispiel ein Signal über weniger als eine festgelegte Anzahl an Routern laufen. Aber auch bei ad-hoc Funknetzwerken oder bei der Datenkomprimierung sowie bei verteilten Mutual Exclusion Algorithmen gilt es immer wieder, einen BDMST zu berechnen.

Bei dieser Problemstellung gilt es, in einem ungerichteten, gewichteten und zusammenhängenden Graphen $G = (V, E)$ mit der Knotenmenge V und der Kantenmenge E einen aufspannenden Baum minimaler Kosten zu finden. Zusätzlich muss gelten, dass die Anzahl der Kanten auf jedem Pfad zwischen zwei beliebigen Knoten innerhalb des Baumes kleiner oder gleich einem Durchmesser D ist. Dieses Problem ist für eine Durchmesserbeschränkung von $4 \leq D < |V| - 1$ $\mathcal{NP}$-schwer.

Es existiert eine Vielzahl verschiedener Verfahren, um dieses Problem zu lösen. Für eine exakte Lösung des BDMST Problems haben sich neben Modellen, die auf Miller-Tucker-Zemlin Ungleichungen aufbauen, besonders spezielle Flussformulierungen als sehr erfolgreich herausgestellt. Während die mit diesen Modellen erhaltenen Schranken sehr gut sind, ist die Anzahl der benötigten Variablen sehr groß. Aufgrund der Komplexität des Problems ist die Anwendbarkeit exakter Verfahren auf relativ kleine Instanzen mit nicht mehr als 100 Knoten beschränkt, sofern von vollständigen Graphen ausgegangen wird. Um Instanzen mit 1000 und mehr Knoten (zumindest

näherungsweise) lösen zu können, wurden auch verschiedene Heuristiken entwickelt. Einfachen, aber schnellen Konstruktionsheuristiken basieren auf dem Algorithmus von Prim zum Finden eines minimalen Spannbaumes (minimum spanning tree, MST). Da diese aber aufgrund sehr lokal getroffener Entscheidungen speziell auf euklidischen Graphen im Normalfall versagen, da hier durchaus auch längere Kanten Teil einer guten Lösung sein können, wurden auch Metaheursitiken eingesetzt – hauptsächlich evolutionäre Algorithmen (evolutionary algorithms, EAs).

In dieser Dissertation werden neben neuen Verfahren, um Instanzen des BDMST Problems moderater Größe beweisbar optimal zu lösen, auch Algorithmen vorgestellt, um auf größeren Instanzen durchmesserbeschränkte Spannbäume in einer Qualität zu berechnen, die bisher nicht möglich war.

Basierend auf drei unterschiedlichen Lösungsrepräsentationen werden fünf, sich gegenseitig ergänzende Nachbarschaftsstrukturen definiert, die dazu verwendet werden, um aus gegebenen Bäumen jeweils neue, bessere Lösungen zu generieren. Um ein möglichst effizientes Durchsuchen der Nachbarschaften zu realisieren, wird hier großes Augenmerk auf eine inkrementelle Auswertung dieser gelegt.

Zwei Formulierungen des BDMST Problems als ganzzahlige lineare Programme (integer linear programs, ILPs) werden vorgestellt, die versuchen, das Hauptproblem der Flussformulierungen, die große Anzahl an Variablen, zu vermeiden. Ein einfaches und kompaktes 0–1 ILP Modell wird, eingebettet in einen Branch&Cut Algorithmus, durch das dynamische Hinzufügen verletzter Ungleichungen zur Sicherstellung der Konnektivität und der Kreisfreiheit zusätzlich gestärkt. Das zweite Modell basiert auf sogenannten „jump inequalities“, Nebenbedingungen, die die Durchmesserbeschränkung sicherstellen. Da ihre Anzahl exponentiell mit der Knotenanzahl $|V|$ steigt, kommt auch hier ein Branch&Cut Verfahren zum Einsatz. Da das Identifizieren einer verletzten „jump“ Nebenbedingung in einer fraktionalen Lösung des zugehörigen linearen Programms mutmaßlich $\mathcal{NP}$-schwer ist, wird eine Hierarchie von (Meta-)Heursitiken eingesetzt, um dieses Separierungsproblem effizient zu lösen.

Für größere Instanzen werden drei unterschiedliche Metaheuristiken vorgestellt, die auf den für das BDMST Problem definierten Nachbarschaften aufbauen. Diese werden als lokale Verbesserungsstrategien innerhalb einer variablen Nachbarschaftssuche (variable neighborhood search, VNS), eines evolutionären Algorithmus mit einer neuen Lösungsrepräsentation, und einem Ameisensystem (ant colony optimization, ACO) eingesetzt. Zu guter Letzt wird noch eine neue Konstruktionsheuristik präsentiert, die auf einem Clustering der Knoten basiert und speziell für große euklidische Instanzen geeignet ist.

Die Testergebnisse zeigen eindrucksvoll die Effizienz der hier beschriebenen Verfahren. Mit Hilfe des ganzzahligen linearen Programms auf Basis der „jump inequalities“ konnte für mehrere bisher nicht exakt lösbare Instanzen das Optimum bestimmt werden. Besonders bei größeren Durchmessern ist dieser Ansatz auch den sehr erfolgreichen Flussformulierungen überlegen. Für Instanzen mit mehreren hundert Knoten ist der vorgestellte ACO die zur Zeit führende Metaheuristik, um qualitativ hochwertige Lösungen in vernünftiger Zeit zu berechnen. Schlussendlich liefert die hier präsentierte Konstruktionsheuristik für sehr große euklidische Instanzen durchmesserbeschränkte Spannbäume mit einer Lösungsgüte, die bisherige Verfahren aus der Literatur besonders bei kleinen Durchmessern bei weitem nicht erreichen konnten.

Abstract

The *bounded diameter minimum spanning tree* (BDMST) problem is a combinatorial optimization problem appearing in applications such as wire-based communication network design when quality of service is of major concern and, for example, a signal between any two nodes in the network should not pass more than a fixed number of routers. It also arises in ad-hoc wireless networks and in the areas of data compression and distributed mutual exclusion algorithms.

Given an undirected, weighted, and connected graph $G = (V, E)$ with a node set V and an edge set E the goal is to identify a tree structure of minimum costs connecting all nodes of this network where the number of edges on each path linking any pair of nodes is limited by a maximum diameter D. This problem is known to be $\mathcal{NP}$-hard for a diameter bound of $4 \leq D < |V| - 1$.

There exist a great variety of different approaches to solve this problem. Beside models based on Miller-Tucker-Zemlin inequalities especially multi-commodity hop-indexed flow formulations have proven to be very successful in exactly solving the BDMST problem. However, to obtain tight linear programming (LP) relaxation bounds these models require a huge number of variables. Due to the complexity of the problem exact approaches are limited to relatively small instances with clearly less than 100 nodes when considering complete graphs. Therefore, heuristics have been developed to solve instances with up to 1000 and more nodes. Fast and simple greedy construction heuristics are primarily based on Prim's minimum spanning tree algorithm, but in particular on Euclidean instances this greedy behavior misleads these heuristics since in this case in general also long edges are part of a good BDMST. For higher quality solutions metaheuristics, especially evolutionary algorithms, have been proposed.

In this thesis, various new approaches are presented to solve moderately sized instances of the BDMST problem to proven optimality, as well as constructing diameter-constrained trees on larger instances of significantly higher quality than it was possible before.

First, five local search neighborhood structures are defined to locally improve (intermediate) trees computed with one of the proposed algorithms. Since they are based on three different representations of a solution they complement each other in a perfect way. Special attention is payed on the efficient evaluation of solutions and moves when searching these neighborhoods.

Two different exact integer linear programming models for the BDMST problem are introduced trying to overcome the main problem of the multi-commodity flow formulations, i.e., the great number of required variables. A simple and compact 0–1 integer linear programming (ILP) model is further strengthened by dynamically adding violated connection and cycle elimination constraints within a Branch&Cut environment. The second model is based on so-called jump inequalities to ensure the diameter bound. Again, Branch&Cut is utilized due to the fact that the number of these jump inequalities grows exponentially with $|V|$. Since identifying a violated jump inequality for a fractional LP solution is conjectured to be $\mathcal{NP}$-hard a hierarchy of (meta-)heuristics is used to solve this separation problem efficiently.

For larger instances three different metaheuristics are proposed making use of the defined neighborhoods. They are utilized by local improvement strategies within a variable neighborhood search (VNS), an evolutionary algorithm (EA) utilizing a new encoding of solutions, and an ant colony optimization (ACO). Finally, a new fast construction heuristic based on clustering is presented designed especially for Euclidean instances.

The computational results demonstrate the efficiency of the discussed approaches. Using the ILP model based on jump inequalities it was possible to discover so long unknown optima for various problem instances, and to compete with state-of-the-art hop-indexed multi-commodity flow formulations, especially when the diameter bound is loose. For larger instances with hundreds of nodes, the ACO is currently the leading metaheuristic to get high-quality solutions within reasonable time. In the end, also the new construction heuristic outperforms standard algorithms from the literature significantly on very large Euclidean instances.

Acknowledgments

Brevity is the soul of wit.

First of all I am deeply grateful to my supervisor Prof. Günther Raidl, who gave me the opportunity to do my PhD at the Vienna University of Technology and introduced me into the field of combinatorial optimization. He provided me with invaluable advices and ideas, help and encouragement. Thank you for your tireless efforts! I further want to thank Prof. Ulrich Pferschy, who agreed to be the second assessor of this thesis, for his feedback and suggestions for improvement.

During my time at university I had a lot of colleagues, and it would be unfair to only name some of them at this place since I am really glad to also call them all my friends! So thank you all for our fruitful discussions and the exchange of ideas, your support and helpful suggestions, and also the lot of fun we had making these years really a great time.

Last but not least I also want to thank my family and all my friends for their love and support over all these years.

My special thanks go to all the people responsible to produce and deliver coffee (as well as the corresponding amount of sugar), their contribution to this thesis will never be forgotten!

Contents

1 Introduction **1**
- 1.1 Methodologies . . . 8
- 1.2 Overview of the Thesis . . . 9

2 Methodologies **13**
- 2.1 Exact Algorithms . . . 13
  - 2.1.1 Linear Programming . . . 13
  - 2.1.2 Dynamic Programming . . . 22
  - 2.1.3 Lagrangean Relaxation (LR) . . . 23
- 2.2 (Meta-)Heuristics . . . 24
  - 2.2.1 Construction Heuristics . . . 24
  - 2.2.2 Approximation Algorithms . . . 25
  - 2.2.3 Local Search . . . 25
  - 2.2.4 Greedy Randomized Adaptive Search Procedure (GRASP) . 26
  - 2.2.5 Variable Neighborhood Search (VNS) . . . 28
  - 2.2.6 Tabu Search (TS) . . . 29
  - 2.2.7 Evolutionary Algorithms (EA) . . . 31
  - 2.2.8 Ant Colony Optimization (ACO) . . . 33
- 2.3 Hybrid Algorithms . . . 34
  - 2.3.1 Incorporating (Meta-)Heuristics in Exact Algorithms . . . 34
  - 2.3.2 Incorporating Exact Algorithms in (Meta-)Heuristics . . . 35
  - 2.3.3 Collaborative Approaches . . . 36

3 Previous Work **39**
- 3.1 Exact Algorithms . . . 39

3.2 Construction Heuristics 45
3.3 Metaheuristics 47
3.4 Approximation Results 48

4 Local Search Neighborhoods 51
4.1 Incremental Evaluation 52
4.2 Tree-Structure Based Neighborhoods 53
4.2.1 Arc Exchange Neighborhood 53
4.2.2 Node Swap Neighborhood 53
4.3 Level-Based Neighborhoods 55
4.3.1 Level Change Neighborhood 56
4.3.2 Center Exchange Level Neighborhood 64
4.4 Clustering-Based Neighborhood 64

5 Level-Based Integer Linear Programming Approach 67
5.1 Introduction 67
5.2 A Compact 0–1 ILP Formulation 68
5.2.1 The Even Diameter Case 68
5.2.2 The Odd Diameter Case 69
5.3 Branch&Cut 70
5.3.1 Connection Cuts 71
5.3.2 Cycle Elimination Cuts 71
5.4 Computational Results 72
5.5 Additional Constraints 75
5.6 Conclusions 76

6 Integer Linear Programming Approach Based on Jump Inequalities 79
6.1 Introduction 79
6.2 The Jump Model 80
6.3 Jump Cut Separation 83
6.3.1 Exact Separation Model 83
6.3.2 Simple Construction Heuristic C^A 85
6.3.3 Constraint Graph Based Construction Heuristic C^B 86
6.3.4 Local Search and Tabu Search 89
6.4 Primal Heuristics 91
6.5 Computational Results 91
6.6 Conclusions and Future Work 98

7 Metaheuristics for the BDMST Problem 99
7.1 Introduction 99
7.2 Variable Neighborhood Search 100

7.3 Evolutionary Algorithm 101
7.4 Ant Colony Optimization 102
7.5 Computational Results 103
7.6 Conclusions 109

8 Clustering-Based Construction Heuristic 111
8.1 Introduction 111
8.2 Clustering-Based Construction Heuristic 112
8.2.1 Hierarchical Clustering 113
8.2.2 Height Restricted Clustering 113
8.2.3 Determining Good Root Nodes 116
8.2.4 Inherent Problem of Clustering 122
8.3 Refinement of Cutting Positions 123
8.4 The Odd-Diameter Case 124
8.5 Local Search Neighborhood 125
8.6 Computational Results 125
8.7 Conclusions and Future Work 136

9 Conclusions 137

Bibliography 141

A Curriculum Vitae 153

Chapter 1

Introduction

Network design is an active topic in research since numerous real world problems can be mapped to a formulation dealing with nodes and edges within a graph. Obviously, problems in the fields of telecommunication networks or information technology infrastructure fall into this category. One fundamental problem in this area is the *minimum spanning tree* (MST) problem where all nodes in a graph have to be linked together in a circle-free structure in the cheapest possible way. The MST problem itself is easy to solve by polynomial-time algorithms like those of Prim [109] or Kruskal [90], but adding additional constraints often make the corresponding optimization problem a hard one. For example, in the *degree-constraint MST* problem a bound on the degree, i.e., the number of incident edges, is imposed on every node in the tree to model that in a telecommunication network the used hardware (e.g., a router or switch) can only handle a limited amount of links. In the *leaf-constrained MST* problem a final solution must have at least l leaves, i.e., nodes with degree one, a problem with applications not only in network but also in circuit design. When not all but only a subset of nodes, the terminals, have to be connected in a tree of minimum costs, this is called the *Steiner tree* problem, where the non-terminals or Steiner nodes are allowed to appear in the final solution but do not have to. An extension is the *prize-collecting Steiner tree* problem where a decision has to be made which customers are connected to an existing infrastructure maximizing the profit under consideration of connection and maintenance costs. *Biconnectivity* is of major interest when designing fault tolerant networks where the loss of one transmission node or the disruption of a single link should not lead to a complete breakdown of the whole communication within the network. However, there are also other problems that can be expressed as network design problems, such as various

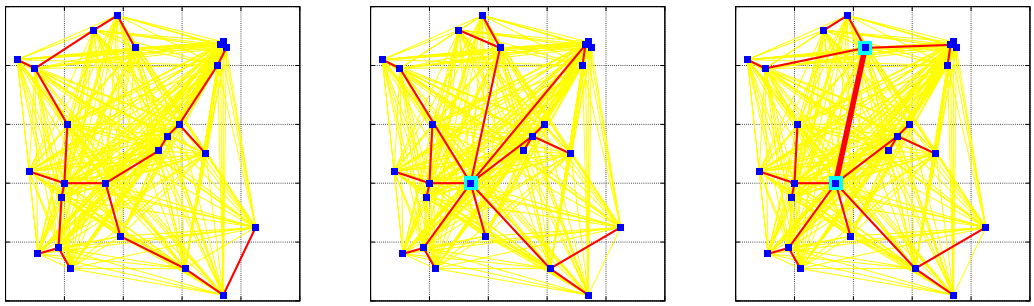
transportation and routing problems. For example, in the famous traveling salesman problem (TSP), one has to find a round trip (Hamiltonian cycle) through a set of cities (nodes) of minimal length. A practical correspondent appears in the automated manufacturing of printed circuits when one wants to minimize the time required for drilling all holes by optimizing the path for moving the drill. Already this short list of problems should give a rough idea of the economical impact and therefore interest of solving such network design problems properly in general.

One of these problems is the *bounded diameter minimum spanning tree* (BDMST) problem where we seek a tree spanning all nodes of the network of minimum costs where the diameter, i.e., the number of edges between any pair of nodes, is limited above by a given constant.

The main application area for the BDMST is in communication network design when quality of service is of major concern, see [23, 140]. Requirements can be for example a limitation of the maximum communication delay or the guarantee for a minimum signal-to-noise ratio. Thus, the number of relaying nodes on any path between two communication partners needs to be restricted.

However, there are also other fields in computer science where the BDMST problem arises as a subproblem. An example are mutual exclusion algorithms as described by Raymond [119]. Before entering a critical section a computer in a distributed environment has to signal its intention and ask for permission. A relevant part of the costs for these operations is the length of the longest path the messages between the computers have to travel. Thus, when a tree structure is used as underlying communication infrastructure as proposed in [119] the diameter of it has a direct influence on the efficiency of the mutual exclusion algorithm.

Another application can be found in textual information retrieval systems, to be more precise in the subproblem of compressing correlated bit-vectors, see [21]. The algorithm can roughly be described as follows: Sparse bit-vectors, i.e., vectors containing only a few ones, can be compressed more efficiently. In a first step similar vectors are clustered. This allows to choose for each cluster a representative r, and to code all other vectors v of the cluster by using the result of the operation v xor r, which will produce less ones in general. To further increase the compression rate not only vectors within a cluster are coded relative to a representative but also the cluster representatives themselves relative to each other, where the relation of the clusters is expressed by a graph spanning them all. Since the decoding has to be unambiguous the created structure must be free of cycles (one representative has to stay uncoded acting as starting point for all operations), which leads to the problem of creating a minimum spanning tree where the Hamming distance between two clusters is used as cost function. The length of the paths within this tree has a considerable impact on the time required to decompress bit-vectors part of the corresponding clusters. As



(a) Unconstrained MST. (b) BDMST with $D = 4$. (c) BDMST with $D = 5$.

Figure 1.1: An unconstrained MST (a) and optimal diameter-constrained trees with a diameter bound of $D = 4$ (b) and $D = 5$ (c), respectively, on a complete Euclidean instance with 25 nodes. In the BDMSTs the respective center is highlighted.

a consequence, there has to be a trade-off between the compression rate (captured in the costs of the spanning tree) and the (de-)compression time (diameter of the tree).

Additional fields of application are described for example in [2], where the BDMST appears as a subproblem within the vehicle routing problem, in [25] dealing with ad-hoc wireless networks, or in [11] presenting dynamic routing algorithms for multicasting in a linear lightwave network.

In the following the BDMST is defined more formally. Before doing so, some additional and more basic definitions are given.

Definition 1 (Network Design Problem, NDP) *Given a graph $G = (V, E)$ with node set V and edge set E, the goal of a* network design problem *is the selection of a subset of nodes $v \in V$ and / or edges $e \in E$ such that some criteria and constraints are met while at the same time the costs of the selection with respect to some objective function have to be optimized.*

Definition 2 (Minimum Spanning Tree Problem, MST) *Let $G = (V, E)$ be a connected, weighted, undirected graph with node set V and edge set E, where each edge e has associated costs $c_e \geq 0$. A* minimum spanning tree *is a cycle-free subgraph $T = (V, E_T)$, $E_T \subseteq E$, of minimum total costs $c(T) = \sum_{e \in E_T} c_e$ connecting all nodes*

into one single component, i.e., there exists a unique path between any two nodes $\in V$ *within* T*; see Fig. 1.1 (a).*

Note that the MST for a graph has not to be unique unless all edge weights are pairwise disjoint. Furthermore, a MST contains exactly $|V|-1$ *edges, otherwise the tree would not span all nodes (*$|E_T| < |V| - 1$*) or it would contain cycles (*$|E_T| > |V| - 1$*).*

Definition 3 (Bounded Diameter Minimum Spanning Tree Problem, BDMST) *Given an undirected connected graph* $G = (V, E)$ *with node set* V *(*$n = |V|$*) and edge set* E *(*$m = |E|$*) with associated costs* $c_e \geq 0$*,* $\forall e \in E$*, we seek a spanning tree* $T = (V, E_T)$ *with edge set* $E_T \subseteq E$ *whose diameter, i.e., the longest path between any two nodes in the tree with respect to the number of edges, does not exceed a given constant* $D \geq 2$*, and whose total costs* $c(T) = \sum_{e \in E_T} c_e$ *are minimal.*

The *eccentricity* of a node $v \in V$ is defined as the maximum number of edges on the path between v and any other node within the tree T. Thus, the diameter D is an upper bound for the eccentricity allowed in the BDMST. A single node (even diameter D) respectively a single edge (odd D) connecting the two nodes of minimum eccentricity form the *center* of the diameter-constrained tree, cf. Fig. 1.1 (b) and (c). The task of identifying a BDMST can then also be seen as choosing an appropriate center and building a height-restricted tree where the unique path from this center to any node of the tree consists of no more than $H = \lfloor \frac{D}{2} \rfloor$ edges.

This problem belongs to the class of *combinatorial optimization* (CO) problems where a choice out of a set of elements, in this case a small set of edges E_T out of all edges E of the graph, has to be made considering additional constraints and optimizing a cost function. Since the BDMST is a relatively general graph problem its first appearance in the literature is hard to fix. From an algorithmic point of view, Maffioli [92] for example discusses transformations of a network where solving a diameter-constrained tree can be used to deal with more than one label (weight, costs, length, or delay) per edge in 1973, and in 1979 Garey and Johnson [51] showed that the BDMST problem is $\mathcal{NP}$-complete. Most of the first applications and algorithms with relevance nowadays – some of them already given above – date to the 80ies and 90ies of the last century.

As already mentioned, the computation of an unconstrained minimum spanning tree (MST) is an easy task using for example the polynomial-time algorithms of Prim [109] or Kruskal [90], but the BDMST problem is known to be $\mathcal{NP}$-hard for $4 \leq D < n - 1$ [51] (under the restriction that not all edge costs are equal). Since a tree spanning n nodes contains exactly $n - 1$ edges a diameter bound D of $n - 1$

and larger is always met when computing a MST. Connecting more than two nodes within a tree structure requires at least a diameter of two. In case $D = 2$ the tree forms a *star*, i.e., all nodes are linked to a single node, the center of the tree. In the $D = 3$ case the center is a single edge where all remaining nodes of the graph are connected to one of its endpoints by the cheaper edge. Therefore, the optimal BDMST can be found in polynomial time by enumerating all stars in $\mathcal{O}(n^2)$ ($D = 2$), respectively by iterating over all edges and connecting the remaining nodes in time $\mathcal{O}(m \cdot n)$ ($D = 3$), which is bounded above by $\mathcal{O}(n^3)$ for complete graphs. For deeper insights into these special cases with $D < 4$ see [62].

The $\mathcal{NP}$-completeness for the BDMST problem with a diameter $D \geq 4$ can be shown by a reduction to the *exact cover by 3-sets* (X3S) problem.

Definition 4 (Exact Cover by 3-Sets, X3S) *Given a set X with $|X| = 3 \cdot q$, and a collection Y of 3-element subsets of X. Then the question to be answered is as follows: Does Y contain an exact cover for X, i.e., a sub-collection $Y' \subseteq Y$ such that every element of X occurs in exactly one member of Y'?*

Theorem 1 *X3S is $\mathcal{NP}$-complete.*

Proof X3S is a generalization of the 3-dimensional matching which is $\mathcal{NP}$-complete, for a full proof see [85]. For the sake of completeness an entire reduction chain from the X3S to the satisfiability (SAT) problem is as follows, see also [51]:

- Exact Cover by 3-Sets (X3S) $\rightarrow$
- 3-Dimensional Matching (3DM) $\rightarrow$
- 3-Satisfiability (3SAT) $\rightarrow$
- Satisfiability (SAT). $\square$

Theorem 2 *BDMST is $\mathcal{NP}$-complete for $D \geq 4$.*

Proof Fig. 1.2 shows the transformation from an X3S to a BDMST instance with a diameter bound of $D = 4$, i.e., a tree with one single center node and a height restriction of $\lfloor \frac{D}{2} \rfloor = 2$.

The elements of the sets X and Y are represented by nodes. Each node of Y is connected to exactly the three nodes of X it represents. Furthermore, a root node r is introduced, the designated center of the BDMST, linked to each node of Y. All edges in the graph have zero costs except the edges connecting r with nodes of Y

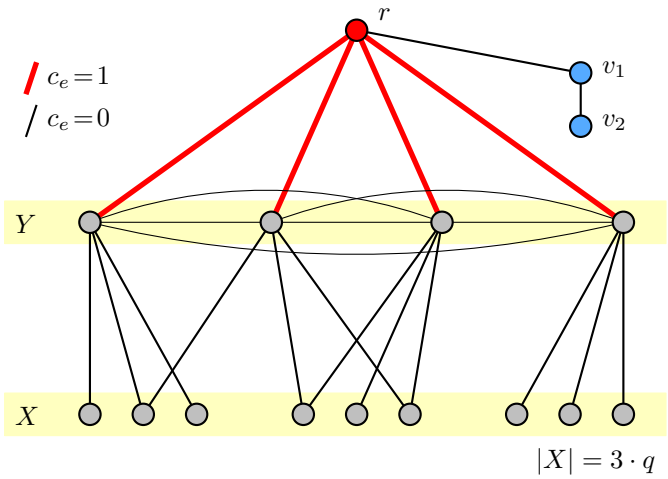


Figure 1.2: $\mathcal{NP}$-completeness of the BDMST problem: Transformation of an X3S instance to a BDMST instance with a diameter bound of $D = 4$.

which have costs of one. Thus, minimizing the costs of the BDMST also minimizes the connections between r and nodes of Y. Since a spanning tree is constructed the minimal number of nodes of Y is used to reach all nodes of X.

As also all nodes of Y have to be connected in the spanning tree, they are linked by additional edges to form a clique, a complete subgraph of zero costs. Due to the imposed height restriction it is not possible to reach a node of X via a node of Y that is not directly connected to the center r.

To avoid that – after augmenting the subgraph induced by Y to form a clique – a node $y \in Y$ becomes the center of the BDMST (this always would lead to a BDMST of costs one: one edge, (y, r), to connect r, all other nodes are reachable over zero cost edges without violating the diameter bound) an additional path (r, v_1) and (v_1, v_2) has to be included into the instance. This way r is now the only valid center for the BDMST since v_2 cannot be reached from any node $y \in Y$ with a path of length $\leq \lfloor \frac{D}{2} \rfloor$.

When solving the BDMST problem to proven optimality on an instance constructed as described the question of the X3S problem can now simply be answered: If the objective value of the BDMST equals q then the center node r is connected to exactly q nodes of set Y, a sub-collection $Y' \subseteq Y$ forming an exact cover of all nodes in X. Otherwise, the costs of the BDMST must be higher, and there exists no solution to the corresponding X3S instance.

This shows that a solver for the BDMST problem can also solve X3S instances. Since all transformations can be performed in polynomial time and X3S belongs to the class of $\mathcal{NP}$-complete problems, it is proven that also the BDMST problem with a diameter bound $D = 4$ is $\mathcal{NP}$-complete.

For $D = 5$ the graph in Fig. 1.2 can be augmented with a second center node r' which is connected via the center edge of zero costs to r. This additional center node r' is – like r – linked to all nodes representing the set Y with edges of costs one. Moreover, the path (r, v_1) and (v_1, v_2) has to be duplicated for r', i.e., (r', v_1') and (v_1', v_2'), to guarantee that the edge (r, r') will definitely be the center of the BDMST. As easily can be seen, an optimal diameter-constrained tree with $D = 5$ and costs equal to q on such an extended graph also solves the corresponding X3C instance. For all diameter bounds $D \geq 6$ the height of the computed tree has to be increased. This can be achieved, for example, by splitting the center r (respectively r and r' in the odd-diameter case) into two or more nodes $r_1, \ldots, r_i$ and adding zero-cost edges (r_1, r_2), (r_2, r_3), $\ldots$, (r_{i-1}, r_i). The last node in this chain, r_i, now is linked to all nodes of set Y. The additional path containing nodes v_k has to extended accordingly, i.e., (r_1, v_1), (v_1, v_2), $\ldots$, (v_i, v_{i+1}). □

At this place, two problems closely related to the BDMST should also be mentioned. In the diameter-constrained tree problem finding the best center is part of the optimization. A related problem, which can be seen as a more restricted version of the BDMST problem, is the *hop constrained MST* problem, where we are given a root node that corresponds to a predefined center:

Definition 5 (Hop Constrained Minimum Spanning Tree Problem, HCMST) *Given a graph $G = (V, E)$, a function c assigning each edge $e \in E$ non-negative costs, a designated node $r \in V$, and a hop limit H, the goal is to find a spanning tree $T = (V, E_T)$ on G, $E_T \subseteq E$, of minimal costs where each unique path within T from r to any other node $v \in V \setminus \{r\}$ consists of no more than H edges [30].*

A possible application is the existence of a dedicated server streaming for example multimedia content within a network and where the maximum delay to each of the clients has to be limited. In this case a fixed delay introduced by an edge or a node is assumed.

A generalization of the HCMST is the following problem:

Definition 6 (Distance or Delay Constrained Minimum Spanning Tree Problem, DCMST) *We are given a graph $G = (V, E)$, each edge $e \in E$ has associated costs $c_e \geq 0$ and an additional distance or delay $d_e \geq 0$. Furthermore, a maximum*

distance or delay limit L is given as well as a designated root node $r \in V$. The objective is to identify a spanning tree $T = (V, E_T)$ on G of minimum costs. In addition, the cumulative distance or delay of all edges of the unique path $P \subseteq E_T$ from r to any other node $v \in V \setminus \{r\}$ within T, i.e., $\sum_{e \in P} d_e$, is bounded above by L [127, 63].

1.1 Methodologies

There are various techniques to solve optimization problems like these presented above. Roughly they can be classified into two main categories: *exact* and *heuristic* algorithms. Exact algorithms are guaranteed to always identify a provable optimal solution (if some exists), but often the runtime behavior does not scale satisfyingly with instance size. As a consequence, exact approaches often are only applied to small or moderately-sized instances while larger instances are solved by heuristics. Heuristics sacrifice the guarantee to reach the optimum for the sake of finding good solutions of acceptable quality within reasonable time. Somewhere in-between are the *approximation algorithms*: Mainly classified as heuristics they are able to give at least some provable bounds on the quality of the computed solution in relation to the optimum.

Examples for successful exact algorithms are *Dynamic Programming* (DP) [19], *Constraint Programming* (CP) [125], *Branch&Bound*, and especially the large family of (integer) linear programming ((I)LP) based approaches, including in particular *Linear Programming based Branch&Bound*, *Branch&Cut*, *Branch&Price*, and *Branch&Cut&Price* [102, 105].

Concerning heuristics there exist constructive methods like *Greedy Heuristics* and techniques such as *Local Search*. Usually, these approaches are highly problem specific. More general solution strategies are the so-called *metaheuristics* [53, 78], which control and manage subordinate, often problem specific heuristics, using various strategies to escape local optima simple heuristics are frequently trapped in. Usually, metaheuristcs are more reliable and robust in finding good solutions, making them an interesting choice to solve difficult optimization problems. Prominent representatives for metaheuristics are *Iterated Local Search* [91], *Tabu Search* (TS) [54], or *Variable Neighborhood Search* (VNS) [75]. Proven to sometimes be very effective are also algorithms inspired by nature and biology, such as *Simulated Annealing* [87], *Ant Colony Optimization* (ACO) [42], or population-based approaches which are especially well suited for parallel processing like *Evolutionary Algorithms* (EA) [9], *Scatter Search* [55], and *Memetic Algorithms* [100].

Both, exact and heuristic methods, have their strengths and weaknesses. In practice, the combination of them to *hybrid algorithms* often allows to improve solution quality (faster algorithms and/or better solutions) by exploiting synergies. Classifications and surveys of different hybridizations of exact optimization techniques with metaheuristics can be found in [117, 111, 118].

For solving the bounded diameter minimum spanning tree problem a great variety of the listed approaches can be used. Two different ILP formulations strengthened by additional cutting planes will be presented to solve the BDMST problem to proven optimality. Altogether five neighborhood structures for the BDMST problem will be described, acting as local search procedures within a *Variable Neighborhood Descend* (VND) [75] for various metaheuristics, as well as for the ILP based exact algorithms. Arising subproblems will be tackled with simple greedy heuristics or suited metaheuristics, but also with exact approaches like dynamic programming when they are appropriately applicable. Special attention will be paid on techniques to speed-up computation using for example preprocessing or an incremental evaluation of solutions.

1.2 Overview of the Thesis

The further organization of this thesis is as follows: In Chapter 2 an introduction to the used methodologies is given, where particular attention is paid to linear and integer linear programming, tabu search, variable neighborhood search, evolutionary algorithms and ant colony optimization. The previous work on the BDMST and related problems is summarized in Chapter 3, followed by a detailed presentation of various local search neighborhoods defined for the BDMST problem in Chapter 4.

Afterwards, various new approaches to solve the BDMST problem developed as main part of the PhD research work are discussed. In Chapter 5, in contrast to multi-commodity flow formulations which include a huge number of variables, a more compact ILP model is presented, which is further strengthened by dynamically adding violated connection and cycle elimination constraints within a Branch&Cut environment. This work has also been published in

> Martin Gruber and Günther R. Raidl: A new 0–1 ILP approach for the bounded diameter minimum spanning tree problem. In L. Gouveia and C. Mourão, editors, *Proceedings of the International Network Optimization Conference,* pages 178–185, Lisbon, Portugal, 2005.

A talk dealing with an enhanced version of this approach utilizing a larger set of different constraints within Branch&Cut, e.g., the stronger directed version of the

connectivity constraints and specialized path constraints, was given by the author at the *10th International Workshop on Combinatorial Optimization* in Aussois, France, 2006.

Chapter 6 covers a significantly improved ILP model further reducing the required number of variables and making use of so-called jump inequalities within Branch&Cut to ensure the diameter restriction. Since the separation subproblem of identifying currently violated jump inequalities is difficult, they are identified heuristically by various construction heuristics, local search, and optionally tabu search. Also a new type of cuts, the center connection cuts, is introduced to strengthen the formulation in the more difficult to solve odd-diameter case. This work was published in

> Martin Gruber and Günther R. Raidl: (Meta-)Heuristic separation of jump cuts in a Branch&Cut approach for the bounded diameter minimum spanning tree problem. In Thomas Stützle and others, editors, *Special issue on Matheuristics of Operations Research.* Computer Science Interface Series, Springer, to appear 2009.

Early versions of this article based on a slightly different model can be found in

> Martin Gruber and Günther R. Raidl: Heuristic cut separation in a Branch&Cut approach for the bounded diameter minimum spanning tree problem. *Proceedings of the 2008 International Symposium on Applications and the Internet (SAINT),* pages 261–264, Turku, Finland, IEEE Computer Society, 2008,

and – using already the final model formulation but presenting only preliminary results – in

> Martin Gruber and Günther R. Raidl: (Meta-)Heuristic separation of jump cuts for the bounded diameter minimum spanning tree problem. In P. Hansen and others, editors, *Proceedings of Matheuristics 2008: Second International Workshop on Model Based Metaheuristics,* Bertinoro, Italy, 2008.

A talk with the title *Heuristic jump cut separation in a Branch&Cut approach for the bounded diameter minimum spanning tree problem* including preliminary results was further given at a meeting of the *Austrian Society of Operations Research (ÖGOR)* in Salzburg, Austria, 2008.

After these exact approaches, Chapter 7 discusses various metaheuristics for the BDMST problem in order to deal with larger instances that cannot be solved by the ILP-based methods anymore. First, a VNS is proposed, mainly based on the neighborhood structures defined in Chapter 4. The corresponding publication is

> Martin Gruber and Günther R. Raidl: Variable neighborhood search for the bounded diameter minimum spanning tree problem. In Pierre Hansen and others, editors, *Proceedings of the 18th Mini Euro Conference on Variable Neighborhood Search,* Tenerife, Spain, 2005.

This work further led to two additional approaches, an EA and ACO, using a new solution representation which make them highly competitive. The ACO is currently still the leading metaheuristic for high-quality solutions. The EA and the ACO were published in

> Martin Gruber, Jano van Hemert, and Günther R. Raidl: Neighborhood searches for the bounded diameter minimum spanning tree problem embedded in a VNS, EA, and ACO. In Maarten Keijzer and others, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO),* pages 1187–1194, Seattle, Washington, USA, ACM Press, 2006.

A talk with preliminary results was given under the same title at the *Austrian Workshop on Metaheuristics 4'06 (AWM)* in Vienna, Austria, 2006.

To address even larger instances and to compute initial solutions for exact or metaheuristic approaches, respectively, a new construction heuristic for the BDMST problem is introduced, which is described in Chapter 8. It is specially designed to approach hard to solve Euclidean instances by using hierarchical clustering to guide the construction process. This work can also be found in

> Martin Gruber and Günther R. Raidl: Exploiting hierarchical clustering for finding bounded diameter minimum spanning trees on Euclidean instances. In Günther R. Raidl et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO),* Montréal, Québec, Canada, ACM Press, to appear 2009.

An early version of this heuristic dealing only with the simpler even-diameter case was published in

> Martin Gruber and Günther R. Raidl: Solving the Euclidean bounded diameter minimum spanning tree problem by clustering-based (meta-) heuristics. In A. Quesada-Arencibia and others, editors, *Twelfth International Conference on Computer Aided Systems Theory (EUROCAST),* Gran Canaria, Spain, Springer LNCS, to appear 2009.

Finally, Chapter 9 summarizes the work and conclusions are drawn.

Chapter 2

Methodologies

This chapter summarizes different approaches to deal with combinatorial optimization problems (COPs). First, exact algorithms are discussed which are able to prove the optimality of a found solution. Afterwards, an overview over various metaheuristics is given to handle larger problem instances no longer solvable by exact algorithms. Finally, hybrid approaches combining the advantages of exact and (meta-)heuristic techniques are described

2.1 Exact Algorithms

Whenever possible, the first attempt should be to solve a given problem to proven optimality. In this overview, the main focus will lie on (mixed) integer linear programming techniques, a powerful class of algorithms for $\mathcal{NP}$-hard combinatorial optimization problems. This part is mainly based on the books by Bertsimas and Tsitsiklis [20] as well as Nemhauser and Wolsey [102] on (integer) linear optimization. The section closes with a short description of dynamic programming, a polynomial-time approach to solve specially structured problems.

2.1.1 Linear Programming

A lot of optimization problems can be formulated as a linear program (LP) which can be solved efficiently in practice using the simplex algorithm or, in guaranteed polynomial time, with the ellipsoid-method [86] or interior-point methods [84].

The standard form of a linear program is as follows:

$$\begin{aligned} \min \quad & cx \\ \text{s.t.} \quad & Ax \quad \geq \quad b \end{aligned}$$

with $c, x \in \mathbb{R}^n,\ b \in \mathbb{R}^m,\ A \in \mathbb{R}^{m \times n}$. The n-dimensional decision vector x and the given vector c of equal dimension form the objective function $c \cdot x$ that has to be optimized subject to the m constraints given as matrix A and vector b. If constraints should be equalities this can be modeled by two corresponding inequalities, and $\leq$ inequalities can be reformulated as $\geq$ inequalities by changing the sign of all coefficients.

Since the diameter-constrained MST and all related problems discussed in this work are minimization problems this formulation will be used throughout this section. This is no restriction of generality since each maximization problem can be transformed into a minimization one by multiplying the objective function with -1. The optimal objective value z^{LP} can now be stated as

$$z^{\mathrm{LP}} = \min\{cx \mid Ax \geq b,\ x \in \mathbb{R}^n\}. \tag{2.1}$$

Integer Linear Programming

As mentioned above, LPs with $x \in \mathbb{R}^n$ can be solved efficiently by polynomial-time algorithms. In case we seek an integer solution, i.e., a solution where $x \in \mathbb{Z}^n$, this problem becomes $\mathcal{NP}$-hard in general. The corresponding integer linear program (ILP) with the optimal objective value z^{IP} can now be formulated as

$$z^{\mathrm{IP}} = \min\{cx \mid Ax \geq b,\ x \in \mathbb{Z}^n\}. \tag{2.2}$$

If the domain of x is restricted to binary variables, i.e., $x \in \{0,1\}^n$, this variant is called a 0–1 ILP. In a mixed linear program (MIP) only some and not all of the decision variables have to be integral.

Geometric Interpretation and the Simplex Algorithm

To get deeper insight into solution algorithms the geometric interpretation of (integer) linear programs can be valuable. Given a linear program LP as denoted by (2.1), the set of all feasible solutions is defined by the following *polyhedron*:

$$P = \{x \in \mathbb{R}^n \mid Ax \geq b,\ A \in \mathbb{R}^{m \times n},\ b \in \mathbb{R}^n\}. \tag{2.3}$$

Based on the characteristics of the polyhedron P the following propositions can be made:

- $P = \emptyset \Rightarrow$ the LP contains no feasible solutions and is therefore infeasible.
- $P \neq \emptyset$, but $\nexists \inf\{c^T x \mid x \in P\} \Rightarrow$ the LP is feasible but unbounded; an optimal solution does not exist.
- $P \neq \emptyset$ and $\exists \min\{c^T x \mid x \in P\} \Rightarrow$ the LP is feasible and an optimal solution denoted by $x^* \in P,\ c^T x^* = \min\{c^T x \mid x \in P\}$ exists.

The following definitions and theorems now directly lead to the *simplex algorithm* to efficiently solve feasible and bounded linear programs:

Definition 7 *A polyhedron $P \subset \mathbb{R}^n$ is* bounded *if there exists a constant k such that $|x_i| < k\ \forall x \in P,\ i = 1, \ldots, n$. Such a bounded polyhedron is called a* polytope.

Definition 8 *A set $S \subset \mathbb{R}^n$ is* convex *if $\lambda x + (1 - \lambda)y \in S,\ \forall x, y \in S,\ \lambda \in [0, 1]$ holds.*

Definition 9 *Given $X = \{x^1, \ldots, x^k\}$, with $x^i \in \mathbb{R}^n,\ \lambda_i \geq 0,\ i = 1, \ldots, k$, and $\sum_{i=1}^{k} \lambda_i = 1$. Then*

(1) the vector $\sum_{i=1}^{k} \lambda_i\, x^i$ is called a convex combination *of X, and*

(2) the convex hull *of X which is denoted as $conv(X)$ is the set of all convex combinations of X.*

Note that all polyhedra defined by a linear program are convex.

Definition 10 *Let P be a polyhedron defined by linear equality and inequality constraints, $x^* \in \mathbb{R}^n$.*

(1) The vector x^* is a basic solution *if:*

  (a) All equality constraints are satisfied, and

  (b) there are n linearly independent inequality constraints that are active *at x^*, i.e., these constraints hold with equality.*

(2) A basic solution x^* that satisfies all constraints is called a basic feasible solution.

Theorem 3 *Let $P \neq \emptyset$ be a nonempty polyhedron, and $x \in P$ a feasible solution. Then the following statements are equivalent:*

(1) x is a vertex of P;

(2) x is a basic feasible solution.

Theorem 4 *Given an LP as defined in (2.1), then the following statements are true:*

(1) If the polyhedron P in (2.3) is nonempty, then there exists a basic feasible solution.

(2) If the LP (2.1) has an optimal solution, then there is an optimal basic feasible solution.

Theorem 4 states that if the linear program is feasible and has an optimal solution x^*, then x^* is a vertex of the corresponding polyhedron P. The simplex algorithm, developed by George Dantzig in 1947, exploits this observation by reducing the search space to vertices of P.

In general, the simplex algorithm is performed in two steps *(two-phase simplex)*: First, an initial basic feasible solution has to be found. In the second phase, the algorithm move to an adjacent vertex of P by simultaneously improving the value of the objective function. This second step is repeated until no adjacent vertex leads to an improvement of the solution, i.e., the optimum has been reached, or one of the adjacent edges (a facet of P) is unbounded.

Already finding an initial basic feasible solution, that is a first vertex of P to start the second phase, can be a difficult task, especially if the point $\{0\}^n$ is not part of the polyhedron P. A method to compute such a first feasible solution transforms the inequality constraints of a linear program given in standard form, i.e., $Ax \geq b$, into equalities by introducing for each of the m inequalities a so-called *slack variable* $\sigma_i \in \mathbb{R}$. $i = 1, \ldots, m$. This linear system of equations $Ax + \sigma = b$ can now be solved using for example the Gaussian elimination method yielding a first feasible solution x^0.

For the given LP, a set of m linear independent column vectors of A forms a basis. Let $B(1), \ldots, B(m)$ denote the indices of the basic variables of solution x^0, and $B = [A_{B(1)} \ldots A_{B(m)}]$ the corresponding basic matrix. Moving from a vertex of P to an adjacent one, called *pivoting*, can now be interpreted as removing one of the basic variables from B, and simultaneously inserting a new variable into the basis. The decision which variable x_j to include into the basis can be predicated on the corresponding *reduced costs* $\overline{c_j}$:

Definition 11 *Let x be a basic solution, B the associated basis matrix as defined above, and c_B the vector of costs of the basic variables. For each j, the* reduced costs $\overline{c_j}$ *of the variable x_j are defined as*

$$\overline{c}_j = c_j - c_B B^{-1} A_j. \tag{2.4}$$

The reduced costs directly correspond to the per unit change in the objective function. Therefore, negative reduced costs indicate a variable that can be inserted into the basis to improve the objective value of the corresponding solution when considering minimization problems. In case all variables have reduced costs greater or equal to zero, an optimum has been reached.

As already mentioned, the simplex algorithm is very efficient in practice to solve LPs to proven optimality, but there exist worst-case scenarios where it degrades, so it is no full polynomial-time algorithm. There are other approaches like the *ellipsoid-method* by Khachiyan [86] or the more successful *interior-point methods* introduced by Karmakar [84] which guarantee a polynomial runtime. Modern LP-solvers utilize in addition to the simplex algorithm different interior-point methods such as the primal-dual or the barrier algorithm.

Duality

For each linear program, now called the *primal* LP, there also exists a corresponding *dual* LP where each constraint of the primal problem has an associated variable in the dual problem, and vice versa. To be more precise, for a primal problem as stated in (2.1), its dual problem can be formulated as:

$$w^{\mathrm{LP}} = \max\{ub \mid uA \leq c,\ u \in \mathbb{R}^m\}, \tag{2.5}$$

where the vector of decision variables $u \in \mathbb{R}^m$ directly corresponds to the m constraints of the primal LP.

Proposition 1 *The dual of the dual problem is the primal problem.*

Proposition 2 (Weak Duality) *If x is primal feasible and u is dual feasible, then*

$$cx \leq ub.$$

The following theorems present fundamental results in duality theory and build the basis for primal-dual algorithms:

Theorem 5 (Strong Duality) *If z^{LP} or w^{LP} is finite, then both, the primal problem (2.1) and the dual (2.5), have the same finite optimal value*

$$z^{\mathrm{LP}} = w^{\mathrm{LP}}.$$

Proposition 3 (Complementary Slackness) *Let x^* be a feasible solution for the primal problem (2.1), as well as u^* a feasible solution for the corresponding dual problem (2.5), then x^* and u^* are optimal solutions if and only if*

$$\begin{aligned} u_i(b - Ax)_i &= 0, \quad \forall i, \text{ and} \\ x_j(uA - c)_j &= 0, \quad \forall j. \end{aligned}$$

LP-Based Branch&Bound

As described above, LPs can be solved efficiently in polynomial time. However, when some (MIP) or all of the decision variables (ILP) have to be integral, the problem becomes $\mathcal{NP}$-hard in general. Branch&Bound is a general purpose optimization algorithm well suited for combinatorial and discrete problems based on two ideas: Split the whole problem into smaller, easier to handle subproblems *(divide&conquer)*, and use bounds computed for the various subproblems to prune whole parts of the *search tree* (the hierarchy of the subproblems, also called the Branch&Bound tree) definitely not containing the optimal solution.

In the following, we assume a minimization problem where $x \in \mathbb{Z}^n$. An upper bound $\overline{z}$ for the whole problem under consideration is any feasible solution and can be obtained for example by heuristics. A lower bound $\underline{z_i}$ for a (sub-)problem P_i can be computed by *relaxing* the integrality constraints for x and solving the corresponding LP, called the *LP relaxation*.

Definition 12 *The LP relaxation of the ILP $z^{\mathrm{IP}} = \min\{cx \mid Ax \geq b,\ x \in \mathbb{Z}^n\}$ is the LP $z^{\mathrm{LP}} = \min\{cx \mid Ax \geq b,\ x \in \mathbb{R}^n\}$.*

Proposition 4 *If a LP is the relaxation of an ILP, then $z^{\mathrm{LP}} \leq z^{\mathrm{IP}}$ (minimization problem).*

Based on the lower and upper bounds the following decisions can be made for a (sub-)problem P_i within in the search tree:

- $\underline{z}_i = \overline{z}$: The optimal solution for P_i has been found.
- $\underline{z}_i > \overline{z}$: The objective value of the best obtainable solution in P_i already exceeds the global upper bound, a feasible solution for the original problem. Therefore, this subproblem cannot contain the optimal solution and can be pruned.
- $\underline{z}_i < \overline{z}$: No conclusions can be drawn at this stage for problem P_i, so further divide it into subproblems and continue with Branch&Bound.

Branching: In general, a problem P_i is split into two different subproblems P_{i_1} and P_{i_2}. This partitioning of the search space is called *branching*. In case the solution $x^{LP,i}$ of the LP relaxation of P_i is not integral, there has to be at least one fractional variable, i.e., a variable $x_j^{LP,i}$ assigned a fractional value. By rounding this variable up and down and using these values as bounds, respectively, we obtain the following two subproblems:

$$\begin{aligned} P_{i_1} &= P_i \cap \{x : x_j \leq \lfloor x_j^{LP,i} \rfloor\}, \text{ and} \\ P_{i_2} &= P_i \cap \{x : x_j \geq \lceil x_j^{LP,i} \rceil\}. \end{aligned}$$

A question arising is which fractional variable to choose for branching. A simple rule would be to take the most fractional one, that is the variable where $x_j^{LP,i} - \lfloor x_j^{LP,i} \rfloor$ is as close to 0.5 as possible. A more advanced method is *strong branching* [138] which computes bounds for all fractional variables and based on these bounds chooses the most promising one for branching.

Selecting next subproblem: Another important issue for the performance of Branch&Bound is the strategy to select the next subproblem to be considered. One possibility is to try to get a good upper bound $\overline{z}$ as fast as possible to be able to prune parts of the search tree early. This can be achieved by preferring newly created subproblems and therefore going deep into the tree first, i.e., performing a *depth-first search*. Another strategy is to reduce the number of subproblems that have to be created thus reducing the overall size of the search tree by following a *best-first search* where the subproblem with the smallest lower bound is favored. Often a combination of these two strategies is used, first diving deep into the tree to find a good and feasible solution fast, and afterwards switching to best-first search. Algorithm 1 gives a detailed overview on the Branch&Bound procedure for a minimization problem.

Algorithm 1: LP-Based Branch&Bound

Input: initial problem P (minimization)
Initialization: upper bound $\overline{z} := \infty$

```
1  set of problems S ← {P}
2  while S ≠ ∅ do
3      choose and remove a problem P_i from S
4      solve LP relaxation of P_i: solution x_i^LP, objective value z_i
5      if P_i is infeasible then
6          prune
7      else if z_i ≥ z̄ then
8          prune
9      else if x_i^LP is integral then
10         z̄ ← z_i
11         new incumbent x* ← x_i^LP
12         prune
13     else
14         create subproblems P_i1 and P_i2
15         S ← S ∪ {P_i1, P_i2}
16 x* is optimal solution of P
```

Cutting Plane Algorithms and Branch&Cut

The size of a linear programming model, i.e., the number of variables and constraints, in general has direct influence on the time required to solve it. There exist LP models requiring a huge number of constraints, sometimes even exponentially many, but usually only a small set of these constraints is required within a solver like the simplex algorithm to identify the optimal solution. In such a case the *cutting plane algorithm* can be used to efficiently solve the complex model as depicted in Algorithm 2 by starting with only a small subset of constraints and dynamically adding only those which are violated by the current LP solution. The algorithm stops if no violated constraints can be found anymore thus the optimal solution of the original problem has been discovered.

The most important part in this algorithm is solving the *separation problem*, which can be defined as follows:

Algorithm 2: Cutting Plane Algorithm

```
start with simplified model containing only a subset of constraints
loop
    solve model → x*
    solve separation problem for x*
    if ∃ constraint a^T x ≥ b_j such that a^T x* < b_j then
        add constraint a^T x ≥ b_j to model
    else
        return x* which is the optimal solution of the original model
```

Definition 13 *Given a solution $\hat{x} \in \mathbb{R}^n$ for a LP (2.1) subject to only a subset of all m constraints. If $\hat{x} \notin conv(X)$, the* separation problem *is to identify a valid inequality $a^T x \geq b_j$ not considered so far that is violated by $\hat{x}$.*

Branch&Cut is an extension of this basic idea for MIPs and ILPs that are solved using Branch&Bound discussed in the previous section. Here the cutting plane algorithm is performed for each subproblem in the search tree, where usually the main intention is to *strengthen* the LP of the corresponding subproblem, i.e., to provide better bounds and therefore to prune the search tree as much as possible. Since the separation problem has to be solved for each subproblem appearing in the search tree, an efficient algorithm for it is crucial. Other aspects to be considered in this context are for example the management of the identified cuts, the maximal number of cuts to be computed for one single LP solution $\hat{x}$, or the search strategy, i.e., to be satisfied with any cut or looking for the best one to improve the bounds.

Column Generation and Branch&Price

As already mentioned above, also the number of variables of a LP model has a strong influence on the runtime behavior when solving it. When such a model contains a large number or even exponentially many variables the same basic idea can be applied as in the cutting plane algorithm: Start with a small subset of variables, solve the corresponding LP, and add only those variables not already part of the model which are able to eventually improve the current solution $\hat{x}$. Again, this is done until no new variable can be identified improving $\hat{x}$, thus the optimum of the original problem has been found. Since a new variable introduced into the LP corresponds to a new column in matrix A of (2.1) this method is named *column generation*.

The subproblem of deciding which variable to add to the LP is called the *pricing problem*. Following the description of the simplex algorithm, a variable with negative reduced costs (minimization problem) is allowed to enter the basis by simultaneously reducing the costs of the objective function. Therefore, the pricing problem is to identify variables with negative reduced costs not already contained in the current LP model.

Branch&Price is the hybridization of Branch&Bound to solve MIPs/ILPs and column generation. In this case, special attention has to be paid on the branching process since not all variables of the original model are available in each subproblem. Also note, that column generation and the cutting plane algorithm are somehow dual to each other since the constraints in the primal correspond to variables in the dual problem, and vice versa.

2.1.2 Dynamic Programming

Beside the class of algorithms based on linear programming primarily applied to $\mathcal{NP}$-hard problems, *dynamic programming* is a technique used for optimization problems solvable in polynomial time [19] (for the quite humorously story behind the name of this method see [44]). Famous representatives for dynamic programming approaches are Dijkstra's shortest path algorithm [38] or the Needleman-Wunsch algorithm to find an optimal pairwise sequence alignment [101] used in the field of bioinformatics.

Following the nomenclature of Cormen et al. in [27], there are two properties an optimization problem has to fulfill so dynamic programming is applicable:

1. **Optimal substructure:** The original problem can be split recursively into smaller subproblems, whereas the optimal solutions of the subproblems can be used in a bottom-up fashion to compute the optimum of the respective parent problem. In contrast to simple divide&conquer, the solution of a parent problem is not simply collecting results from the subproblems but making a choice which optimal subproblem solutions to use and how to combine them to obtain the optimum for the parent problem.
2. **Overlapping subproblems:** The space of subproblems should be small in a sense that the number of distinct subproblems is polynomial in the input size, and one and the same subproblem can appear more than once when splitting the original problem recursively. A crucial point in this context is that each unique subproblem is only solved exactly once, and the corresponding solution is stored for further usage in some sort of memory (which – of course – has to

be organized in a way the solution for a subproblem can be found efficiently when available).

Defining the search space and, strongly related, how to combine the solutions of the subproblems to an optimal one for the parent problem is the fine art of dynamic programing.

2.1.3 Lagrangean Relaxation (LR)

The *Lagrangean relaxation* [138, 18] is a somehow special technique since its main focus is not to create an optimal (or heuristic) solution to an optimization problem but to compute good bounds for the optimal objective value. These bounds can be used either within the class of Branch&Bound algorithms for MIPs and ILPs, or to give some information on the quality of solutions computed by other (meta-)heuristic approaches. With tight bounds in addition to good heuristic solutions even the optimality of such a solution sometimes can be proved when their objective values correspond.

Given an integer linear program for an $\mathcal{NP}$-hard optimization problem in standard form, cf. (2.2), but this time the set of constraints is split into two:

$$\begin{array}{lrcl} \min & cx & & \\ \text{s.t.} & Ax & \geq & b \\ & Bx & \geq & d \\ & x & \in & \mathbb{Z}^n. \end{array} \tag{2.6}$$

Often, an $\mathcal{NP}$-hard optimization problem is composed of a simple one with additional constraints, or multiple simple problems combined with so-called *coupling constraints*. Let us assume that the optimization problem without the inequalities $Ax \geq b$ can be solved efficiently, for example by a polynomial-time exact algorithm. To compute a lower bound for the original ILP, the inequalities $Ax \geq b$ can be relaxed, since removing constraints of a minimization problem can only reduce the corresponding objective value. Instead of directly considering these constraints, they are brought into the objective function with a vector of *Lagrange multipliers* $\lambda \geq 0$ attached to the relaxed inequalities:

$$\begin{array}{lrcl} \min & cx & + & \lambda(b - Ax) \\ \text{s.t.} & Bx & \geq & d \\ & x & \in & \mathbb{Z}^n. \end{array} \tag{2.7}$$

The program (2.7) is called the *Lagrangean lower bound program*, and $\lambda(b - Ax)$ can be seen as penalty in case $Ax \geq b$ is violated. To get the best, i.e., the maximal lower

bound, Lagrange multipliers have to be found optimizing the following *Lagrangean dual program:*

$$\max_{\lambda \geq 0} \left\{ \begin{array}{lrcl} \min & cx & + & \lambda(b - Ax) \\ \text{s.t.} & Bx & \geq & d \\ & x & \in & \mathbb{Z}^n \end{array} \right\} \tag{2.8}$$

As this problem is piecewise linear and concave it can be solved efficiently for example using *subgradient optimization* [18] or the *volume algorithm* [14]. In general, the solutions of the Lagrangean lower bound program are primal infeasible, i.e., no solutions for the original problem (due to relaxed and violated inequalities), but *repair heuristics* can be utilized to compute upper bounds.

2.2 (Meta-)Heuristics

When confronted with $\mathcal{NP}$-hard combinatorial optimization problems, exact approaches often are only applicable to relatively small problem instances due to runtime and sometimes also memory restrictions. Heuristics and especially *metaheuristics* can be seen as alternatives when large instances have to be solved in reasonable time, whereas these approaches are not able to guarantee to reach the optimum. Nevertheless, for real-world optimization problems they are often the only opportunity to get high-quality solutions with limited resources.

The term metaheuristic has been introduced by Glover [52] and denotes a problem-independent high-level solution strategy managing and controlling subordinate heuristics, which themselves are highly problem specific in general.

This section starts with an introduction to some basic heuristics, and afterwards, an overview on some important metaheurstics is given.

2.2.1 Construction Heuristics

Usually, *construction heuristics* are simple and fast algorithms to compute a first feasible solution, which can be used as an initial solution for iterative approaches to refine it, or as bound for exact algorithms. A certain characteristic of these heuristics is that no backtracking is performed, i.e., once a decision has been made during the construction process it will never be put in question again, a reason for their computational efficiency. A special class are the *greedy* algorithms considering in each construction step only the locally optimal choice.

2.2.2 Approximation Algorithms

In contrast to simple heuristics, *approximation algorithms* [136] are able to provide some information about the quality of a computed solution, i.e., they give a bound for the distance from a created solution to the optimum. More formally, this can be stated as follows:

Definition 14 *Given an optimization problem P (minimization) and an algorithm A computing feasible solutions for every instance $I \in P$, let $c_A(I)$ denote the objective value of the solution generated by A, and let $c_{\text{opt}}(I)$ be the optimal objective value. If*

$$\exists\, \varepsilon > 0 : \quad \frac{c_A(I)}{c_{\text{opt}}(I)} \leq \varepsilon, \quad \forall I \in P$$

holds, then A is an ε-approximation algorithm for P, and ε is the approximation factor of A.

There exist different classes of approximation algorithms: Those with an absolute or relative approximation factor, and (fully) polynomial time approximation schemes where runtime complexity can be traded for better approximation factors. Also note that not every optimization problem is approximable.

2.2.3 Local Search

While construction heuristics build a feasible solution from scratch, *local search* starts from an initial solution and refines it iteratively by performing (small) local changes, so-called *moves*, which improve the objective value. The terms *neighborhood structure* and *neighborhood* are of vital importance for local search and can be defined as follows:

Definition 15 *Given the set $\mathcal{S}$ of feasible solutions for an optimization problem. A neighborhood structure $\mathcal{N} : \mathcal{S} \to 2^{\mathcal{S}}$ is a function associating every solution $x \in \mathcal{S}$ a set of neighbors, called the neighborhood $\mathcal{N}(x) \subseteq \mathcal{S}$ of x.*

Algorithm 3 shows the basic local search algorithm in more detail. A crucial point of this procedure with significant impact on the runtime behavior and the achievable solution quality is the selection process of a neighboring solution $x' \in \mathcal{N}(x)$. There are three possibilities:

Algorithm 3: Basic Local Search (x)

Input: initial solution x
Output: (potentially) improved solution x

```
repeat
    choose a neighboring solution x' ∈ N(x)
    if f(x') better than f(x) then
        x ← x'
until some termination criterion is met
return x
```

- **Best improvement:** The complete neighborhood $\mathcal{N}(x)$ is explored, the best solution x' is chosen.
- **Next improvement:** The neighborhood $\mathcal{N}(x)$ is searched systematically, the first solution x' with $f(x')$ better than $f(x)$ is chosen.
- **Random neighbor:** A random solution $x' \in \mathcal{N}(x)$ is chosen and evaluated. Here the selection process itself is very fast, but in most cases $f(x')$ will be worse than $f(x)$.

There is no strategy dominating all others in the general case. However, using best or next improvement depends on various parameters like the problem to be solved, the definition of the neighborhood structure, or if there exists an efficient incremental evaluation scheme.

Local search is performed until a stopping criterion is met, which can be the maximal number of iterations, a time limit, or when reaching a local optimum:

Definition 16 *Let x be a feasible solution in $\mathcal{S}$, and $f(x)$ the objective value of x. Then x is a local optimum of a minimization problem $\leftrightarrow \forall x' \in \mathcal{N}(x) : f(x') \geq f(x)$.*

Of course, a local optimum with respect to a chosen neighborhood structure is not necessarily a *global optimum*, but each global optimum is also a local one.

2.2.4 Greedy Randomized Adaptive Search Procedure (GRASP)

The *greedy randomized adaptive search procedure* [46, 47] is a multi-start method where each iteration consists of two phases: A construction phase, where a feasible solution is computed, which is refined in the second phase using local search.

Algorithm 4: Greedy Randomized Adaptive Search Procedure

Input: instance I of optimization problem P
Output: best found feasible solution x

```
1  initialize so far best solution x+
2  repeat
3      solution x ← ∅
4      repeat
5          build RCL                          // restricted candidate list
6          select an element x_j from RCL at random
7          add x_j to solution x
8      until x is a complete solution
9      locally improve x
10     if f(x) better than f(x+) then
11         x+ ← x
12 until some termination criterion is met
13 return best found solution x+
```

The overall best solution is stored and return as result of GRASP when a stopping criterion like exceeded runtime or maximal number of iterations is met, see Algorithm 4.

The construction phase does not follow a pure greedy approach but makes use of a so-called *restricted candidate list* (RCL). Whenever a new element will be added to the solution, such a list is computed in advance where each candidate for incorporation is evaluated according to a greedy cost function. The best elements are allowed to enter the RCL, and one of them is chosen at random to be part of the current solution. The size of the list can be static, i.e., the $l_{\max}$ best candidates are used for the RCL, or can depend on the incremental costs caused to objective value of the considered elements. In the latter case, a parameter $\alpha \in [0, 1]$ is used to control the selection pressure, reaching from only accepting the best candidate (pure greedy) to a complete random approach where each candidate element leading to a feasible solution is allowed to join the RCL. If α is a self-tuned parameter this is referred to as *reactive GRASP*.

To improve the performance of the basic algorithm there exist various modifications and enhancements, including for example cost perturbations to enforce search diversification, the introduction of long-term memory to store several good solutions to be used to bias the construction phase, or the replacement of the uniformly random

selection of a candidate element from the RCL by a probability distribution taking into account the incremental costs of the different elements.

2.2.5 Variable Neighborhood Search (VNS)

Variable neighborhood search, introduced by Hansen and Mladenović [75], is a metaheuristic making use of multiple neighborhood structures defined for the considered optimization problem, and a technique to escape local optima. It is based on following observations:

- A local optimum with respect to one neighborhood structure is not necessarily a local optimum to another one.
- A global optimum is a local optimum with respect to every possible neighborhood structures.
- Often local optima with respect to different neighborhood structures are relatively close to each other.

Variable Neighborhood Descend (VND)

Variable neighborhood descend implements the first observation given above and can be seen as a local search procedure using not just one but multiple neighborhood structures, $\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_{k_{\max}}$. An initial solution x is thereby systematically improved with respect to the various neighborhood structures until a local optimum for all of them is reached, see Algorithm 5.

In general, the ordering of the neighborhood structures is crucial for the performance of the VND, not only with respect to the runtime but also to the achievable solution quality. Different properties of these neighborhood structures have to be considered, like their relationship to each other (overlapping, (partly) including one another, mutual exclusive), the complexity to search them, or their coverage of the whole solution space. The simplest choice is to use a static ordering, usually based on the runtime complexity since the first neighborhood $\mathcal{N}_1$ is more often searched than $\mathcal{N}_{k_{\max}}$. However, there also exist more sophisticated approaches like the *self-adaptive VND* which dynamically reorders the neighborhood structures according to their execution time and their success to improve a solution [79]. Another method quickly evaluates relaxations of the different neighborhood structures to be able to choose the most promising one next [112].

Algorithm 5: Variable Neighborhood Descent (x)

Input: initial solution x; neighborhoods $\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_{k_{\max}}$
Output: (probably) improved solution x

```
1  k ← 1
2  repeat
3  |  choose a neighboring solution x' ∈ N_k(x)
4  |  if f(x') better than f(x) then
5  |  |  x ← x'
6  |  |  k ← 1
7  |  else
8  |  |  k ← k + 1
9  until k = k_max
10 return x
```

Basic Variable Neighborhood Search

In addition to a strong local improvement component, VNS also includes a mechanism to escape local optima, the so-called *shaking* process. For this purpose, a set of neighborhood structures $\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_{l_{\max}}$, which are usually ordered by size and can be different from those exploited within VND, is used to perform random moves from the so far best found solution, cf. Algorithm 6.

When following a best improvement strategy in the local search step, a single random shaking move in the same neighborhood would not be sufficient to escape a local optimum. In such a case a sequence of moves is often used to perform shaking in corresponding larger neighborhoods.

Like GRASP, VNS is a very simple to implement metaheuristic with only a few parameters to tune making it easy it use in practice. Variants of the basic VNS include *reduced VNS* which completely omit the local search phase, thus relying only on the shaking process, as well as the *general VNS* utilizing VND to locally improve solutions.

2.2.6 Tabu Search (TS)

A problem almost all iterative metaheuristics are confronted with is *cycling*, i.e., returning to an already visited and evaluated point in the solution space. It is obvious that evaluating a solution more than once is waste of computational effort.

Algorithm 6: Basic Variable Neighborhood Search (x)

Input: initial solution x; neighborhoods $\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_{l_{\max}}$
Output: (probably) improved solution x

```
1  repeat
2      l ← 1
3      repeat
4          choose random solution x' ∈ N_l(x)                    // shaking
5          locally improve x'                          // local search, VND, ...
6          if f(x') better than f(x) then
7              x ← x'
8              l ← 1
9          else
10             l ← l + 1
11     until l = l_max
12 until some termination criterion is met
13 return x
```

Of course, all visited solutions can be stored within a long-term memory but the size and management of such a memory can quickly make the positive effects undone. *Tabu search* [54] makes a compromise and uses a so-called *tabu list* L of restricted length to share information for avoiding or at least reducing cycling. Algorithm 7 shows the procedure in detail.

The tabu list may store whole solutions which are not allowed to be reached again, but depending on the considered optimization problem and instance size this can lead to high memory requirements. More often, only some characteristic attributes of a solution are recorded in L for a certain time (number of iterations). This approach is much more memory efficient, but has the drawback to exclude all solutions from search containing the stored attributes, whether already visited or not.

An especially critical parameter is the *tabu tenure* t_L which controls the length of the tabu list. If it is chosen too small, cycling is not avoided reliably, and if it is too large, the search space is restricted too strongly, eventually disallowing also highly promising not yet evaluated solutions. In practice, identifying a good tabu tenure for the concrete problem and instance can be a difficult task. It can be chosen static, or t_L can be adapted dynamically during runtime [37, 97, 15].

To be sure not to miss a good solution or even the optimum, *aspiration criteria* can be defined when to accept a solution although it should be tabu due to L. A standard one is to keep a solution if it is the overall best so far.

Algorithm 7: Tabu Search (x)

Input: initial solution x
Output: (probably) improved solution x

```
1  L ← {x}                                                    // tabu list
2  x' ← x
3  repeat
4  |  x' ← best solution ∈ N(x') with respect to L and aspiration criteria
5  |  add x' to L
6  |  remove all elements from L older than t_L               // tabu tenure
7  |  if f(x') better than f(x) then
8  |  |  x ← x'
9  until some termination criterion is met
10 return x
```

2.2.7 Evolutionary Algorithms (EA)

Evolutionary algorithms are a family of optimization techniques inspired by nature and the theory of evolution as proposed by Darwin [32] as well as the work of Mendel [96] concerning genetics. Part of this family are for example *genetic algorithms* [77] and *evolution strategies* [120, 128], and there exist a lot of other variants and extensions, cf. [9].

A special feature of evolutionary algorithms when compared to most other metaheuristics like GRASP or VNS is the concept of operating not on a single solution but a whole set, called *population*. Such a population can be created by fast construction heuristics, which require a probabilistic component to compute different *individuals*. This diversity of a population, which has to be preserved during optimization, in general allows to cover a larger part of the search space, helps to escape from local optima, and leads to more robust results.

Not only the concept of a population but also the operators are inspired from evolutionary theory: *selection*, *recombination*, and *mutation*, see Algorithm 8.

- **Selection:** In a first step individuals of the population have to be selected which are allowed in the following to create offspring solutions. The *selection pressure*, i.e., the expected probability of how much the best solution in the current population is preferred over other ones to become a parent, has a major impact on the final solution quality as well as on the runtime of the EA. Various selection mechanisms have been developed, including fitness-proportional [77] and rank-based [10] selection, where individuals are chosen with a probability

Algorithm 8: Basic Evolutionary Algorithm

Output: best solution found during optimization

```
1  create population P of initial solutions
2  evaluate P
3  repeat
4      repeat
5          select parental solutions from population          // selection
6          offspring x ← recombine parental solutions         // recombination
7          mutate x                                           // mutation
8          evaluate x
9      until enough offspring solutions created
10     update population
11 until some termination criterion is met
12 return best found solution
```

according to their fitness or rank (position in the list of solutions sorted by fitness), respectively, or tournament selection [45], where a small number of individuals is chosen at random and from them the best one is selected to be a parental solution.

- **Recombination/Crossover:** In the recombination step, new offspring solutions are computed based on selected parents by inheriting as much as possible parental attributes. This operator is highly problem dependent and has to respect the solution representation in order to create a feasible offspring. In the literature a lot of various crossover operators have been proposed, for example one point crossover, partially mapped crossover [56], order crossover [33], uniform order based crossover [132], or edge recombination crossover [28].
- **Mutation:** To (re-)insert attributes not present in the initial population or lost during the optimization process, small random manipulations are applied to an offspring solution. Usually, mutation is only performed with a relatively low probability.

The basic evolutionary algorithm as depicted in Algorithm 8 contains no local improvement step for individuals of the population, so the returned solution is not guaranteed to be even local optimal with respect to any neighborhood structure. *Memetic algorithms* [99] close this gap by introducing local search (or other local improvement techniques) into the EAs, not only to enhance the solution quality but also to speed up computation. However, a balance has to be found between diversification (solutions as different as possible in the population) and intensification (locally improved individuals) to avoid a bad convergence behavior.

2.2.8 Ant Colony Optimization (ACO)

Another class of metaheuristics motivated by nature is *ant colony optimization* [39]. It is mainly based on the ability of ants to find the shortest path between their nest and a food source by means of a *pheromone* trail. Since in the same period of time more ants can pass the shortest of all possible paths the pheromone on it will increase much faster thus directing other ants with a higher probability towards this path.

ACOs are constructive metaheuristics where (artificial) ants build solutions from scratch. Within this construction phase, an ant usually can access two types of information: A local one indicating in a greedy fashion the best elements to be incorporated into the current solution, and a more global view represented by pheromone deposited by complete and feasible solutions to bias the local decision. The strength of ACOs are combinatorial optimization problems where in a broader sense the computation of a shortest path appears at least as a subproblem, for example in vehicle routing, scheduling, or routing problems. The basic ant colony optimization algorithm is very simple as can be seen in Algorithm 9.

Algorithm 9: Ant Colony Optimization

Output: best solution found during optimization

```
repeat
    manage ant activities     // build solution(s), deposit pheromone, ...
    evaporate pheromone
    perform optional daemon activities
until some termination criterion is met
return best found solution
```

Beside the construction of solutions by ants there are two additional procedures within an ant colony optimization algorithm. In the *pheromone evaporation* phase the pheromone deposited by ants in former iterations of the algorithm is decreased which is required to avoid a too fast convergence. The optional *daemon activities* allow to implement some global actions like a local improvement of constructed solutions or the depositing of extra pheromone for the so long best solution (also called a *off-line pheromone update*).

Ant colony optimization algorithms have a relatively large number of strategy parameters, e.g., the relation between local and global information when selecting an element in the construction phase, the number of ants, and the *pheromone decay coefficient* controlling the evaporation process, thus tuning them can be a challenging task. There are also some strategic decisions to be made like for example which ants

are allowed to deposit pheromone, all of them or only the best ones. This leads to a great variety of different ant algorithms. Some prominent members are *elitist ant systems* [41] where the so far best ant is allowed to deposit additional pheromone, and the $\mathcal{MAX}$–$\mathcal{MIN}$ *ant system* ($\mathcal{MM}$AS) [131] where only the globally most successful or the best ant of an iteration deposits pheromone and upper and lower bounds on the pheromone values are introduced to encourage the exploration of the search space.

2.3 Hybrid Algorithms

All the exact and (meta-)heuristic approaches presented in this chapter so long have their assets and drawbacks, but often they also can complement each other and benefit from synergy thus leading to *hybrid algorithms* [117]. According to [111], such hybrid algorithms can be classified into two main categories, namely collaborative combinations where exact approaches and metaheuristics are executed in sequential order, in parallel, or intertwined and exchange information with each other, or integrative combinations where exact algorithms are subordinates of metaheuristics or vice versa. In the following, some of the main hybridization ideas and techniques with focus on integrative approaches are presented.

2.3.1 Incorporating (Meta-)Heuristics in Exact Algorithms

For Branch&Bound to exploits its full potential, it requires good feasible solutions to the original problem yielding tight global bounds to prune as much as possible from the search tree. Each of these parts of the algorithm can benefit from heuristics. Providing a good initial solution to start with is an obvious contribution of heuristics to Branch&Bound. Whenever a new incumbent solution has been found, it can instantly be locally improved to enhance the global bound further. (Meta-)heuristics can also be applied on fractional solutions appearing during the optimization process, i.e., on every node within the Branch&Bound tree. However, the computational effort invested has to pay off, and especially metaheuristics can require a lot of runtime to converge, thus a decision has to be made where in the search tree and when to apply such heuristics. For example, in [139] Woodruff presented a chunking-based selection strategy that measures a distance between the already explored nodes of the search tree and the current one to restrict the execution of runtime-intense procedures to relatively distant nodes.

However, there are also other techniques inspired by (meta-)heuristics to get good global bounds early in the optimization process when using Branch&Bound. This

includes for example *guided dive* [31], a modification to the standard approach of using depth-first search where the subproblem is explored first in which the branching variable is set to the value of the currently best feasible solution, thus biasing the search to neighbors of the current incumbent solution. Another very successful method is *local branching* [49] where the problem under consideration is split into two subproblems based on an incumbent solution. The smaller of these two subproblems is restricted to solutions that differs only in k variables from the current incumbent solution and is forced to be solved first. In case a better solution is found, this new incumbent solution is further investigated following the same scheme recursively. This way, the neighborhood of good feasible solutions is explored first before going back to the remaining subproblems.

Another application of (meta-)heuristics within exact algorithms is the computation of cutting planes [8, 121] and solving the pricing problem in an column generation approach [48, 110]. Whereas the identification of violated inequalities for Branch&Cut is not critical in the sense that missing a cut only affects the tightness of bounds but does not touch the feasibility or optimality of the whole optimization problem, the pricing problem has to be solved exactly in the end in case we seek for the optimal solution to the original problem. However, every decision variable with negative reduced costs improves the solution, thus a hierarchy of different methods can be used to speed up the column generation process: First, fast heuristics and metaheuristics are applied, and only in case they are no longer able to identify new variables with negative reduced costs, an exact approach is used to solve the pricing problem or to prove that the current solution is optimal.

2.3.2 Incorporating Exact Algorithms in (Meta-)Heuristics

The exact solution of a relaxed and therefore easier to solve problem can act as starting point for further investigations by heuristics to identify, e.g., good initial solutions for various other algorithms. Using an LP relaxation, fractional variables can be rounded for example and the resulting solution – if necessary – may be repaired to become feasible. Adding a stochastic component to the rounding and repair process can lead to a whole population of promising solutions [115, 107]. Another possibility is to fix some of the integral variables of a solution to the LP relaxation thus reducing the size of the remaining search space significantly. This approach is referred to as *core method* [12], whereas the definition of the fixed core can further be extended in several ways, see for example [114, 135].

Metaheuristics based on local search techniques rely on the definition of one or more neighborhood structures. To perform a local improvement, there have to exist efficient algorithms to search such a neighborhood. Usually, classical enumeration

techniques are used in this context limiting the size of the neighborhoods that can be explored within reasonable time. However, finding the best neighboring solution can also be defined as an optimization problem of its own and more sophisticated approaches can be applied to search them thus making it possible to also deal with neighborhood structures of exponential size. In *very large-scale neighborhood* search for example (I)LP solvers can be utilized to explore the surrounding of an incumbent solution by fixing some decision variables to their respective values and performing a search over the remaining ones [22, 108]. In case the search space of the large neighborhood is defined appropriately also dynamic programming can be applied, then this approach is referred to as *dynasearch* [26].

Crossover is a crucial operator for the performance, runtime as well as solution quality, of evolutionary algorithms. Usually, the recombination of two or more parental solutions to create a new offspring by merging their attributes is done in a simple and therefore fast way relying heavily on random decisions. However, the offspring is often worse than its parents leading to a relatively large number of iterations to achieve an improvement. Therefore, investing more computational effort in identifying a better offspring can pay off. *Path relinking* [55] follows this basic idea by tracing a path between two parental solutions in the search space, i.e., one solution is transformed in small steps into the other one whereas the best solution found during this procedure is taken as result. *Solution merging* extends this approach by considering not only a single path between the parental solutions but the whole solution space spanned by their attributes. Identifying the best solution within this subspace can be a hard to solve optimization problem, but usually the parental solutions only differ in a few attributes thus making it manageable. This solution merging technique can be used to replace the crossover operator within an EA, then referred to as *optimized crossover* [5], but can also be applied as a post-processing step to two or more heuristic solutions as in [7]. Finally, another application of exact algorithms in the field of EAs is the decoding of indirect or incomplete solution representations, see for example [130].

2.3.3 Collaborative Approaches

The key issue of hybrid algorithms following a collaborative strategy is to exchange some sort of information so at least one of the partners can profit from discovered knowledge about the search space.

An interesting approach in this context is to start with a Lagrangean relaxation of the original problem using the volume algorithm to solve the Lagrangean dual program, which gives some primal feasible solutions as a by-product. They now can be used to restrict the search space for a subsequent metaheuristic by only

considering attributes already appearing in these solutions [76, 106], a similar idea to solution merging discussed above. Also information about dual variables gathered during the volume algorithm can be exploited to bias the search to interesting regions of the solution space.

Another example for a collaborative approach is the hybrid of an EA and Branch&Bound proposed in [50] to solve general ILPs, where the two algorithms are executed intertwined. Branch&Bound starts to explore the search space and collects information about promising solutions. When a certain criterion is met, this gathered information is used to augment the initial population of the EA. The best solution found by the EA is then passed back to Branch&Bound and so forth.

Of course, the exchange of information is not restricted to complete or parts of solutions. In [113] a Branch&Cut approach cooperates with a memetic algorithm in a parallel and asynchronous way, where values of dual variables are transfered to improve the repair and local search procedures. Furthermore, a multi-agent based application called TECHS (TEams for Cooperative Heterogenous Search) is described in [35]. Teams of one or more agents following the same search strategy exchange not only positive (solutions) but also negative (parts of the search space already explored) information. Each agent alternates between performing its optimization algorithm and processing messages, whereas the communication is filtered for the various agents by so-called send- and receive-referees.

All these listed cooperative approaches achieve better results on the corresponding problems they were tested on than the individual algorithms executed for their own thus demonstrating impressively the potential of hybrids between exact techniques and (meta-)heuristics.

Chapter 3

Previous Work

This chapter summarizes previous work in the field of the bounded diameter minimum spanning tree problem, as well as other relevant work such as on the hop constrained MST problem.

3.1 Exact Algorithms

When confronted with a combinatorial optimization problem, the first attempt always should be to consider solving it with an efficient algorithm to proven optimality. Fundamental work of using mixed integer program (MIP) methods in the field of network design was done by Magnanti and Wong [93], who proposed multi-source multi-destination network flow models.

Achuthan and Caccetta [3] suggested a simpler flow model specifically for the BDMST operating on a directed version of the instance graph, a standard technique to strengthen the formulation where each undirected edge e is replaced by two oppositely directed arcs, both with the same costs c_e. For the even-diameter case they introduce a virtual source s and a target t, see Fig. 3.1 (a). The number of arcs leaving s is restricted to one, and all the directed paths from s to the target are limited in their length to $\lfloor \frac{D}{2} \rfloor + 2$. When D is odd they proposed a model containing two different target nodes t_1 and t_2, where all directed paths end at t_2 except one directed to t_1 which is also allowed to contain one additional arc in length, the center edge connecting c_1 and c_2. At this point the formulation is incorrect, as can be seen in Fig. 3.1 (b). The result is a diameter-constrained tree, but in an optimal solution

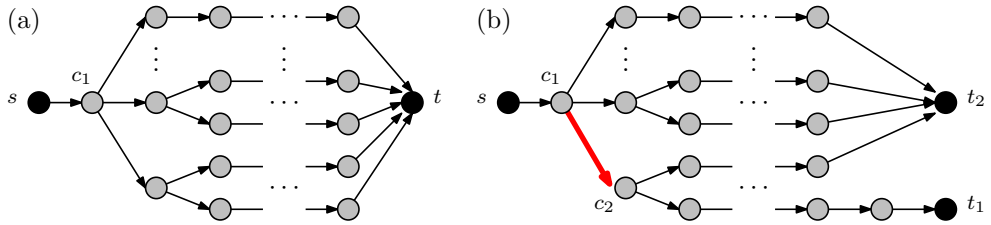


Figure 3.1: Flow model by Achuthan and Caccetta introducing a virtual source s and target nodes t, t_1, and t_2 for the even (a) and (incorrect) odd-diameter case (b), respectively. c_1 and c_2 denote the center of the BDMST, the bold arc in (b) the center edge.

all directed paths crossing the arc (c_1, c_2) are allowed to have length $\lfloor \frac{D}{2} \rfloor + 3$ to exploit the full height, not just a single one ending at t_1. An advanced version no longer requiring virtual target nodes (thus also formulating the odd-diameter case correct) was published by Achuthan et al. in [4] together with a Branch&Bound framework utilizing different branching rules and simple heuristics to improve the performance.

Gouveia and Magnanti [59] investigated different extended variants of multi-commodity flow (MCF) formulations for the BDMST problem, in which they count and limit the hops on paths from a virtual root node to any other node. They subdivided the problem of finding a diameter constrained tree into different subproblems: creating a rooted spanning tree, modeling hop-constrained paths from the virtual root, and selecting the center edge if the diameter bound is odd. For each of these subproblems they presented various approaches leading to a pool of formulations to solve the entire problem, which were evaluated in detail.

The most successful one is a hop-indexed single-source multi-commodity flow formulation which will be recapitulated here for the even-diameter case due to its high relevance. As done by Achuthan and Caccetta, the undirected edges $(i, j) \in E$ of the instance graph are replaced by directed arcs (i, j) and $(j, i) \in A$ with the same costs as the undirected edge. In addition, a virtual root node r is introduced, i.e., $V_r = V \cup \{r\}$, and $A_r = A \cup (r, i)\ \forall i \in V$, whereas all arcs (r, i) have zero-costs. The model makes use of the following variables:

- x_{ij}, $(i, j) \in A$: the directed arc (i, j) is part of the diameter-constrained MST
- y_{ij}^k, $(i, j) \in A_r$: the arc (i, j) is part of the unique path from r to $k \in V$
- z_{ij}^{hk}, $(i, j) \in A_r$, $h = 1, \ldots, H = \lfloor \frac{D}{2} \rfloor + 1$: (i, j) is the h^{th} arc on the unique path from r to $k \in V$

$$\text{minimize} \quad \sum_{(i,j)\in A_r} c_{ij}\, x_{ij} \tag{3.1}$$

$$\text{subject to} \sum_{(i,j)\in A_r} x_{ij} = |V| \tag{3.2}$$

$$\sum_{j\in V} y_{ij}^k - \sum_{j\in V_r} y_{ji}^k = \begin{cases} 1 & i = r \\ 0 & i \neq r, k \quad \forall k \in V, i \in V_r \\ -1 & i = k \end{cases} \tag{3.3}$$

$$y_{ij}^k \leq x_{ij} \qquad \forall (i,j) \in A_r, k \in V \tag{3.4}$$

$$y_{ij}^k \geq 0 \qquad \forall (i,j) \in A_r, k \in V \tag{3.5}$$

$$x_{ij} \in \{0,1\} \qquad \forall (i,j) \in A_r \tag{3.6}$$

$$\sum_{j\in V} x_{rj} = 1 \tag{3.7}$$

$$\sum_{j\in V} z_{rj}^{1k} = 1 \tag{3.8}$$

$$\sum_{j\in V} z_{ij}^{2k} = z_{ri}^{1k} \qquad \forall i \in V \tag{3.9}$$

$$\sum_{j\in V} z_{ij}^{h+1,k} - \sum_{j\in V} z_{ji}^{hk} = 0 \qquad \forall i \in V, h = 2, \ldots, H-1 \tag{3.10}$$

$$\sum_{j\in V} z_{ij}^{Hk} = 1 \tag{3.11}$$

$$y_{ij}^k = \sum_{h=1,\ldots,H} z_{ij}^{hk} \qquad \forall (i,j) \in A_r, k \in V \tag{3.12}$$

$$z_{ij}^{hk} \in \{0,1\} \qquad \forall (i,j) \in A_r, h = 1, \ldots, H \tag{3.13}$$

$$z_{kk}^{hk} \in \{0,1\} \qquad \forall k \in V, h = 2, \ldots, H \tag{3.14}$$

(3.1)–(3.6) describe a spanning arborescence of minimum costs rooted at r: The costs of all arcs part of the BDMST are accounted for the objective value (3.1), and the spanning arborescence including the virtual root node consists of $|V|$ arcs (3.2). The flow conservation constraints (3.3) guarantee that there is a unique directed path from r to each node $k \in V$. The coupling constraints (3.4) ensure that each arc carrying flow for a commodity k is considered in the objective function. Finally, the variables y_{ij}^k have to be positive or equal to null (3.5), and the x_{ij} are binary variables (3.6).

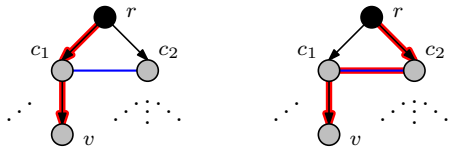


Figure 3.2: Triangle-tree: When D is odd there are two paths from the virtual root r to any node v of the instance graph, a direct one and one crossing the center edge (c_1, c_2).

Equation (3.7) now selects exactly one node to be the center of the BDMST with an even diameter bound D, and (3.8)–(3.14) formulate that no directed path from r to any node $k \in V$ contains more than H arcs: The constraints state that – starting at the virtual root r (3.8) – an arc is only allowed to enter a node i in position h when there is another arc emanating from i in position $h+1$, and there have to be one arc entering node k in position H (3.9)–(3.11). Since not all paths to nodes $k \in V$ in the final BDMST are of length H there have to be so-called loop variables z_{kk}^{hk}, $h = 2, \ldots, H$ to artificially extend short paths to the desired length. In the end, equations (3.12) couple the binary hop-index with the corresponding multi-commodity flow variables.

Using this model Gouveia and Magnanti achieved extremely tight LP bounds with a gap of less than 1% to the optimal solution for almost all tested problem instances. They were able to solve BDMST instances with up to 60 nodes and 600 edges and a diameter constraint $D \leq 8$ to proven optimality, although especially on Euclidean graphs the runtime as well as the memory requirements increase quickly due to the large number of variables ($\mathcal{O}(|V| \cdot |E| \cdot H)$) and corresponding constraints in the formulation.

In [60] (an enhanced version can be found in [61]) Gouveia et al. introduced an extended flow formulation for the odd diameter case based on the model given above. The BDMST is viewed as being composed of a variant of a directed spanning tree from an artificial root node r together with two constrained paths from the root to any node in the so-called *triangle-tree*: r is connected to the two center nodes which are themselves linked via the center edge building a cycle of length three. As a consequence, there exist two directed paths from r to any other node v of the instance, a direct one from r to one of the center nodes and then to v, and a second, one edge longer path from r to the other center node over the center edge to v, see Fig. 3.2. The strength of this formulation is based on the possibility to explicitly formulate the fact that the center edge always has to be the second arc on the longer directed path from the artificial root to any node of tree. This model for the complex odd-diameter case is characterized by relatively small LP relaxation

gaps of usually less than 5%, and it allows to handle instance sizes comparable to the case where D is even, i.e., up to 60 nodes and 600 edges. Gouveia et al. [59, 60, 61] also applied modified MCF formulations to the related Steiner tree problem with hop constraints.

A formulation based on lifted Miller-Tucker-Zemlin inequalities responsible for avoiding cycles and ensuring the maximum diameter are presented in Santos et al. [43]. This approach is claimed to work well in particular on dense graphs but is not able to reach the LP gaps as well as solution times of the MCF models presented above. See Voß [137] for the use of MTZ constraints to compute Steiner trees.

Recently, a constraint programming approach has been proposed by Noronha et al. in [103] for the BDMST problem using one single formulation to model the even as well as the odd diameter case. Although giving respectable results for extremely tight diameter bounds it cannot compete with state-of-the-art multi-commodity flow and Branch&Cut formulations.

Hop Constrained Minimum Spanning Tree Problem

Identifying a diameter constrained MST can be reformulated as a special variant of the HCMST problem by introducing an artificial root node and restricting its connection to the center of the BDMST as described above. Therefore, publications in the field of hop-constrained trees also have some relevance for the BDMST problem.

Not only for the diameter- but also for the hop-constrained MST problem hop-indexed multi-commodity flow formulations have proven to be very successful with their tight LP bounds, see for example [30] which also gives an overview on several other models and solution approaches for this problem. Due to the size of the strong MCF formulations it is sometimes promising to use Lagrangean relaxation to compute good bounds. In [58] Gouveia proposed such an approach by dualizing the coupling constraints between the arc (x_{ij}) and flow variables (y^k_{ij}) for the various commodities k (cf. inequalities (3.4)). Especially for larger and dense instances, Gouveia and Requejo [64] use an alternative to derive a Lagrangean relaxation by dualizing the flow-conservation constraints (cf. equations (3.3)). The computed lower bounds can compete with the corresponding LP relaxation values but require in general less running time.

An also well working approach in particular for a smaller hop limit H is the reformulation of the problem as a Steiner tree problem on a layered directed graph [65], see Fig. 3.3 (a) and (b). Here the nodes of the instance graph are duplicated to different layers according to the numbers of allowed hops, numbered from 1 to H.

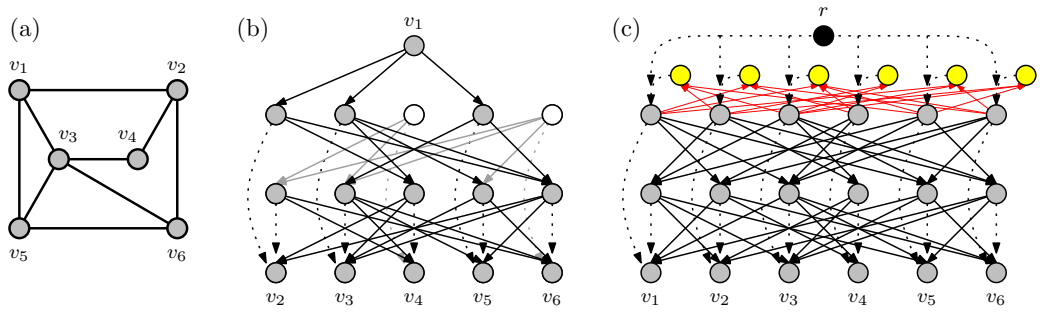


Figure 3.3: An instance graph (a) and its representation as a layered graph (b) with root node v_1 and hop limit $H = 3$. In (c) the transformation for an odd-diameter BDMST ($D = 5$) is shown where an artificial root r and an additional level have been introduced. Dotted lines denote arcs of zero cost.

The root r of the HCMST is the only node assigned to layer 0. Each node v at a layer 1 to $H-1$ is directly connected to its copy at layer H by a zero cost arc. The edges of the instance are replaced by arcs linking the corresponding nodes of directly adjacent layers with appropriate costs, directed towards layer H. The nodes at the highest layer represent the terminal nodes to be reached by directed paths from r across the different layers. There exist efficient algorithms for solving even large Steiner tree problems, see for example [34], but of course with increasing H also the size of layered graph grows quickly thus making this reformulation reasonably applicable only to instances with moderate hop limits. In the revised and significantly extended version [66] also the BDMST problem is covered as a by-product. The transformation of the even-diameter case is straightforward by introducing an artificial root r, connecting it with zero-cost arcs to all instance nodes of layer 1, and limiting the arcs emanating from it in a solution to one. The hop limit is set to $H = \lfloor \frac{D}{2} \rfloor + 1$. In case the diameter is odd, an additional layer -1 containing all instance nodes is introduced, cf. Fig. 3.3 (c). This layer can only be reached from layer 1 using arcs corresponding to the underlying graph with appropriate costs. Each node of layer -1 is linked with its copy in 1 by a zero-cost arc. An additional constraint guarantees that exactly one arc from layer 1 to layer -1 is chosen, the center edge. Applying this transformation and using a Branch&Cut formulation to solve the corresponding Steiner tree problem the layered graph approach is very successful. The authors report that in most cases BDMST instances with up to 161 nodes can be solved without branching.

Another strong formulation is based on so-called *jump inequalities* [29] used to enforce the hop constraint; they will be described in more detail in Section 6.2. Unfor-

tunately, their number grows exponentially with $|V|$, and the problem of separating them in a cutting plane algorithm is conjectured to be $\mathcal{NP}$-hard. Therefore, Dahl et al. [29] exploited them in a Relax&Cut algorithm where violated jump inequalities only need to be identified for integer solutions, which is straightforward using a simple depth first search procedure.

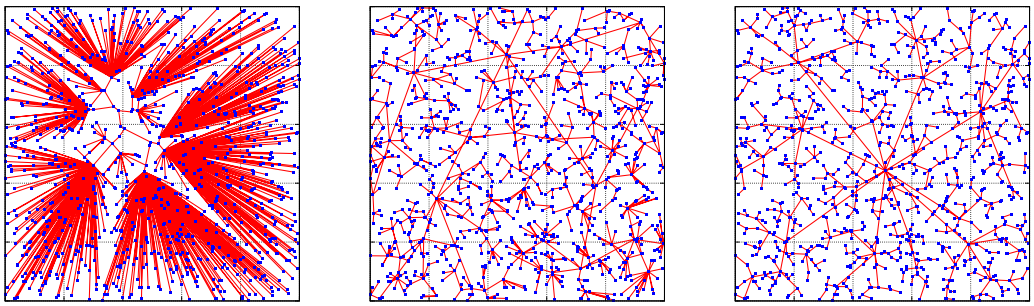
A noticeable result about the approximability of height-constrained minimum spanning and Steiner trees with implications also for the BDMST problem was published by Manyem and Stallmann [95]. They showed that these problems are not within APX, the class of $\mathcal{NP}$ optimization problems for which it is possible to give polynomial time heuristics that guarantee a constant approximation bound.

3.2 Construction Heuristics

Since exact algorithms are not able to solve instances of even moderate size to proven optimality within reasonable time, heuristics have been developed. To get acceptable solutions fast for instances with hundreds of nodes which can also be utilized as basis for more sophisticated metaheuristics simple construction heuristics play an important role.

They are primarily based on Prim's classical minimum spanning tree (MST) algorithm [109]. It starts growing the tree at an arbitrary node and then always adds the node with the cheapest connection to the partial spanning tree until all nodes are part of the MST.

Abdalla et al. describe in [1] several construction heuristics for the BDMST problem. For the special case of a diameter bound of $D = 4$ they proposed a transformation from the optimal BDMST with a diameter of three which can be determined in polynomial time (cf. Chapter 1). A more general approach is the *one-time tree construction* (OTTC), a heuristic based on Prim's MST algorithm and especially suggested for the case of instances with random edge costs and a smaller diameter bound. While performing the standard algorithm of Prim starting at an arbitrary node the procedure keeps track of the eccentricities of all nodes in the so long constructed partial tree. This allows for verifying if the cheapest link connecting a node so long not part of the tree will violate the diameter constraint. If so, the connection is established and all eccentricities have to be updated, a procedure that can be implemented in $\mathcal{O}(n)$. Otherwise, this link has to be omitted and the next edge has to be considered. Including the update procedure OTTC requires time $\mathcal{O}(n^3)$ to compute a BDMST for one starting node. Since OTTC is sensitive to the choice of this initial starting node it has to be performed for each node of the instance leading to an overall runtime complexity of $\mathcal{O}(n^4)$ to find the best tree OTTC is able



(a) CBTC (271.4). (b) RTC (41.18). (c) ACO (31.09).

Figure 3.4: A diameter constrained tree with $D = 10$ constructed using (a) the CBTC heuristic, compared to (b) RTC (best solution from 1000 runs) and (c) a solution obtained by an ant colony optimization approach (complete, Euclidean graph with 1000 nodes distributed randomly in the unit square, the corresponding objective values are given in parentheses).

to construct. Finally, in [1] also two iterative refinement algorithms are proposed which are designed for rather loose diameter bounds. Starting from an unconstrained MST paths that are too long are iteratively shortened until the diameter bound is met. In [36] Deo and Abdalla report results for all these algorithms implemented on a massively parallel SIMD machine (single instruction, multiple data) with more than 8000 processors.

For the *center-based tree construction* (CBTC) Julstrom [82] exploits the observation that a BDMST can be viewed as a height-restricted tree (height $H = \lfloor \frac{D}{2} \rfloor$) rooted at the center to reduce the runtime of OTTC significantly. When starting at the center there is no need to keep track of eccentricities and the corresponding update procedure can be omitted. Ensuring the height limit is a much simpler task and can be done in constant time, thus reducing the complexity of creating the best possible tree to $\mathcal{O}(n^3)$ ($\mathcal{O}(n^2)$ for each instance node as center) in the even diameter case. Nevertheless, if D is odd CBTC has to be executed for each edge as center leading to an overall runtime complexity of $\mathcal{O}(n^4)$ for complete graphs.

This approach works reasonably well on instances with random edge costs, but on Euclidean instances which are – also for exact algorithms – much harder to solve this leads to a backbone (the edges near the center) of relatively short edges where the majority of the nodes have to be connected to the backbone via relatively long edges, see the example in Fig. 3.4(a). On the contrary, a good solution as shown in Fig. 3.4(c) obtained from a metaheuristic, demonstrates that the backbone

should consist of a few longer edges to span the whole area so the large number of remaining nodes can be connected as leaves by much cheaper edges. In a pure greedy construction heuristic this observation is difficult to realize. In the *randomized tree construction approach* (RTC, Fig. 3.4(b)) from [82] not the overall cheapest node is always added to the partial spanning tree but the next node is chosen at random and then connected by the cheapest feasible edge. Thus at least the possibility to include longer edges into the backbone at the beginning of the algorithm is increased. For Euclidean instances RTC is so far the best choice to quickly create a first reasonable solution as basis for exact or metaheuristic approaches.

There also exists another well-known algorithm for the unconstrained MST by Kruskal [90] where in a first step the edges of the graph are sorted ascending according to their costs. Afterwards, the edges are considered in this order and an edge is accepted for the tree as long as it does not lead to a cycle. So this algorithm starts with a forest of independent trees (the single nodes) and iteratively connects them until this procedure results in a single spanning tree of minimum costs. This approach makes it much more difficult to impose additional constraints like a diameter restriction to the whole MST. Furthermore, the construction heuristics for the BDMST problem operate in general on complete graphs which is no restriction since each graph can be augmented with sufficiently costly edges. However, on dense or even complete graphs Kruskal's algorithm has a higher runtime complexity than Prim's MST algorithm since it is dominated by sorting all edges [98]. Nevertheless, for the related delay-constrained MST problem a successful approach has been proposed by Ruthmair et al. [126].

3.3 Metaheuristics

Beside the greedy construction heuristics various evolutionary algorithms (EAs) were developed for the BDMST problem in order to obtain better results. Raidl and Julstrom [116] presented an EA employing a direct edge-set encoding where in the recombination procedure special attention is payed to efficiently derive an offspring having most edges in common with its parent solutions. In addition, they introduce four variation operators that are able to produce new candidate solutions in almost $\mathcal{O}(n)$ expected time. In [83] the same authors suggested an EA using a permutation representation, which determines the order in which the nodes are appended by an RTC-like decoding heuristic. For recombination partially mapped crossover (PMX) [56] is utilized which tends to preserve the positions of nodes from the two parents to the offspring. This approach leads to better solutions, but with the drawback of longer running times for large instances since decoding a chromosome requires $\mathcal{O}(n^2)$

time. Another EA based on random-keys has been proposed in [81]. A comparison to the permutation-coded approach indicated a similar performance.

Singh and Gupta [129] improved the permutation encoded EA by including a local improvement into their decoding procedure which leads to better results much faster. They use a local search neighborhood already found in [68] where a subtree is truncated from the solution and is reconnected – without changing the root of the subtree – at another feasible place in the tree but at lower costs. When searching the neighborhood they exploit knowledge of the construction phase to reduce the search space considerable in practice. The same RTC-like decoding procedure is utilized to derive a BDMST from a permutation π of all nodes, i.e., a node v at position i in π already was checked against all nodes with an index $j < i$, and the cheapest connection was established. In the following local improvement step, after a complete diameter constrained tree was built, a cheaper place than its current one for the subtree rooted at v can only be found when connecting it to a node not already part of the tree when v was initially inserted during construction. Therefore, the search for a cheaper and feasible place in the BDMST to move v with its complete subtree to can be restricted to nodes with an index $k > i$ within π.

3.4 Approximation Results

As mentioned in section on exact approaches for the hop-constrained MST, Manyem and Stallmann showed in [95] that the height-constrained minimum spanning and Steiner trees, and therefore also the BDMST problem, are not within APX, so there exists no polynomial time heuristic guaranteeing a constant approximation bound. In [13] Bar-Ilan et al. describe a logarithmic ratio approximation algorithm for the special cases of a diameter bound of $D = 4$ and $D = 5$ where they assume special conditions for the costs of the instance graph edges. The result was generalized by Kortsarz and Peleg in [88] where they present a polynomial time approximation algorithm of ratio $\mathcal{O}(D \cdot \log n)$ for a constant D, as well as an algorithm for general D of ratio $\mathcal{O}(n^{\varepsilon})$ for any fixed $0 < \varepsilon < 1$. From the same authors similar results also exist for Steiner trees, see [89].

Recently, Angel et al. [6] gave a sharp bound for the costs of height respectively diameter constrained minimum spanning and Steiner trees when $n \to \infty$ and the edge costs of the complete graph are exponential random variables between 0 and 1. They showed that for $H \geq \log_2 \log n + \omega(1)$, where $\omega(1)$ is any function going to ∞ with n but slower, the costs of both, a height restricted and a diameter constrained MST with $D = 2H$, tend to that of an unconstrained MST, which is $\zeta(3) = 1/1^3 + 1/2^3 + 1/3^3 + \cdots$. If $1 \leq H \leq \log_2 \log n - \omega(1)$, the overall costs tend to $(1 -$

$2^{-H})\sqrt{8}(\sqrt{2n^2}/2^H)^{1/(2^H-1)}$ in both expectation and probability. More theoretical work concerning different properties of spanning trees, amongst others the diameter, can be found for example in [124, 122, 133].

Chapter 4

Local Search Neighborhoods

In the following chapters, various new exact as well as (meta-)heuristic approaches will be presented for the BDMST problem. They all have in common that they can benefit from a local improvement of candidate solutions, even the exact algorithms where intermediate trees can be improved to obtain better bounds to speed up Branch&Bound or Branch&Cut. In this chapter, five local search neighborhoods designed for the BDMST problem are defined. All of them will only consider feasible solutions. Based on the representation of the diameter-constrained tree the neighborhoods can be categorized into three classes: tree-structure based, level-based, and clustering-based neighborhoods.

For an efficient implementation, we represent a solution as an outgoing arborescence, i.e., a directed tree rooted at a center, using the following data structures, cf. Fig. 4.1:

- An array *pred* containing for each node $v \in V$ its direct predecessor in the directed path from the center to it, respectively `NULL` in case v is a center node;
- for each node $v \in V$ a list $succ(v)$ of all its direct successors; for a leaf this list is empty;
- an array *lev* storing the level for each node $v \in V$, which is the length of the path from the center to v;
- for each level $l = 0, \ldots, H$ a list V_l of all nodes at level l.

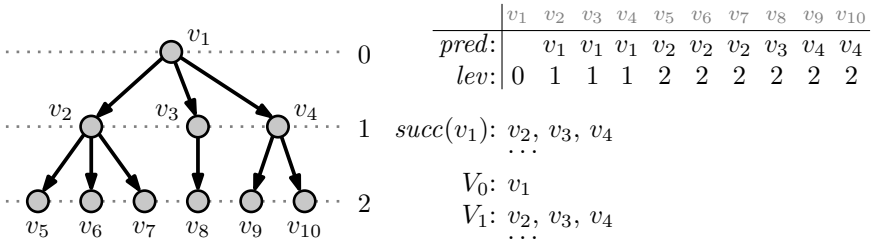


Figure 4.1: Data structures required for an efficient implementation to search the various neighborhoods.

4.1 Incremental Evaluation

A standard technique within local search to reduce the computational effort is to incrementally evaluate the objective value of an solution. For the concrete example of a BDMST solution, only the contributions to the objective value of those parts of the tree have to be recalculated which have been modified by a move, an operation (slightly) altering a solution.

In this work we go one step further. Whenever possible, not only the objective value but also neighboring solutions are evaluated incrementally: All neighborhoods are searched using a best improvement strategy where from all possible moves within a neighborhood the one with the biggest gain is executed; ties are broken arbitrarily. As a consequence, the whole neighborhood has to be searched to identify the best improving move. This means, that from a given diameter-constrained tree every neighboring solution that can be reached within one move has to be evaluated. Depending on the size of the neighborhood these computations have an important impact on the running time and therefore should be limited. In case now a single performed move within a neighborhood has only small effects on the already evaluated neighboring solutions, these solutions can be stored in some sort of cache. Then only the affected part of this cache has to be reevaluated and updated accordingly whenever a move has been executed, and the subsequent best improvement move within the same neighborhood structure can be identified without checking again every possible move. Of course, this strategy can only be exploited if a solution is modified within one neighborhood until a local optimum is reached, i.e., best improvement moves are performed until the solution could no longer be refined. In the following sections it will be explicitly pointed out, together with algorithmic details, whenever such an incremental evaluation of neighboring solutions is reasonably possible.

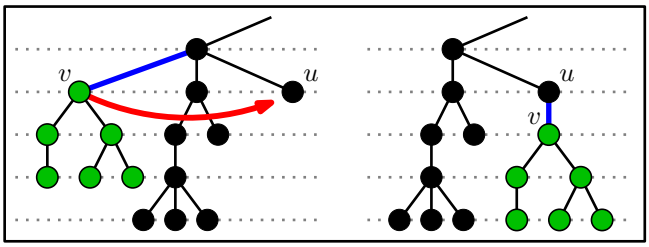


Figure 4.2: Arc exchange neighborhood: Disconnecting a subtree and reconnecting it at another feasible place.

4.2 Tree-Structure Based Neighborhoods

The first two neighborhoods are based on the tree structure defined by the relationship between predecessors and successors in the directed tree rooted at the center of the BDMST.

4.2.1 Arc Exchange Neighborhood

The arc exchange neighborhood of a current solution T consists of all feasible trees differing from T in exactly one single arc (directed edge). The associated move can be interpreted as disconnecting some sub-tree and reconnecting it at another feasible place, see Fig. 4.2.

This neighborhood consists of $\mathcal{O}(n^2)$ solutions. A single neighbor can be evaluated in constant time when only considering cost differences. To ensure that the diameter constraint is not violated for each node $v \in V$ the height $h(v)$ of the sub-tree rooted at it is predetermined; feasible candidates for becoming new predecessor of a node v after disconnecting it are all nodes at levels less than or equal to $H - h(v) - 1$. Under this conditions, the total time for examining the whole neighborhood in order to identify the best move is in $\mathcal{O}(n^2)$.

4.2.2 Node Swap Neighborhood

This neighborhood focuses on the relationship between nodes and their set of direct successors. A neighboring solution is defined as a tree in which a node v and one of its direct successors u exchange their positions within the tree: In the example of Fig. 4.3 node u becomes predecessor of v and successor of v's former predecessor.

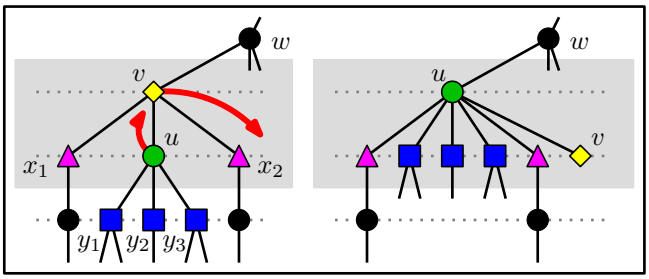


Figure 4.3: Node swap neighborhood: Reorganizing the positions of a node and its direct successors in the BDMST.

While node u can keep its successors (u is moved to a smaller level), the successors of v are reconnected to u in order to always ensure feasibility with respect to the diameter constraint in an easy way.

In contrast to arc exchange, a move in this neighborhood can result in several new connections, which means that on sparse graphs it could be difficult to identify a valid move. Nevertheless, the whole neighborhood has only size $\mathcal{O}(n)$ and it can be efficiently examined in time $\mathcal{O}(n \cdot d_{\max})$, where $d_{\max}$ is the maximum degree of any node in the current tree.

Furthermore, this neighborhood allows for an incremental evaluation of improvement moves as suggested in Section 4.1 since the rearrangement of nodes only has small local effects. When searching the whole neighborhood, all improvement moves are stored within a priority queue, sorted according to the achievable gain. Within this queue, for a move denoted by a node v the following data have to be recorded: v itself, the successor u which will exchange its position with v, and the improvement in the objective value. After executing an improvement move for a node v, the priority queue have to be updated. This includes the reevaluation of moves for the following nodes: The old successors x_i of v, because the subtree resulting in swapping a node x_i with one of its direct successors now has to connect to u instead of v; the new root of the subtree u since it has new direct successors as well as a new predecessor; and w as u is now a new direct successor which has to be considered accordingly. Afterwards, the priority queue contains again all improvement moves for the current solution with respect to the node swap neighborhood, so the best one can be chosen to be executed until the queue becomes empty, i.e., a local optimum has been reached.

node i	1	2	3	4	5	6	7	8	9
$lev(i)$	0	2	2	2	1	2	1	2	1

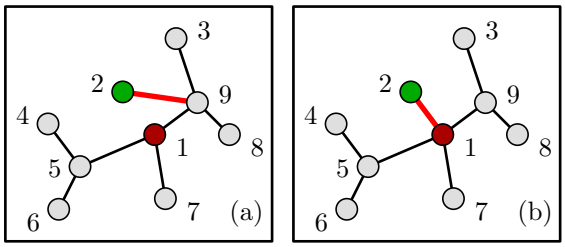


Figure 4.4: Decoding a level vector (Euclidean distances, $D = 4$) strictly following the levels (a), respectively using the relaxed interpretation of a level (b).

Algorithm 10: Decoding a level vector *lev*.

```
1 for every node v ∈ V do
2     if Σ_{l=0}^{lev(v)-1} |V_l| < δ then
3         search level lists V_l, l = 0, ..., lev(v) − 1, for the cheapest predecessor p
          for node v
4     else
5         search sorted nearest neighbor list of v for the first node p assigned to
          a level < lev(v)
6     pred(v) ← p;
```

4.3 Level-Based Neighborhoods

For the next two neighborhoods the representation of a solution is changed and therefore the search space: Instead of the predecessor respectively successor information, the level a node is assigned to is of main interest. Given the level $lev(v)$, $\forall v \in V$, it is straight-forward to derive an optimal bounded diameter spanning tree with respect to *lev*: To each non-center node v we assign a least-cost predecessor from $V_{lev(v)-1}$, cf. Fig. 4.4-a.

In order to obtain even better solutions we go one step further and relax the meaning of "level" in this decoding procedure: For a node at level l, any node at a level smaller than l (not just $l-1$) is allowed as predecessor, and an overall cheapest connection is chosen. In case of ties a node of minimum level is selected. See Fig. 4.4-b for an example. Note in particular that node 2 has level 2 and is connected to the center node 1 at level 0 since this is the nearest node at a smaller level.

Algorithm 10 shows this decoding in a more detailed pseudo-code. In order to find the least-cost predecessor of a node i as quickly as possible, we use the following

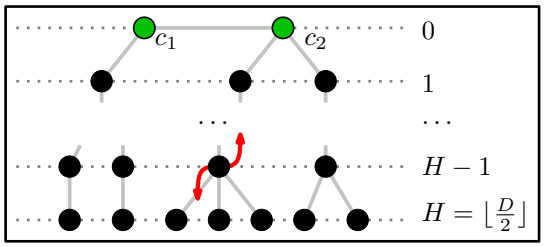


Figure 4.5: Level change neighborhood: Moving a non-center node one level towards or away from the center.

strategy: Only if the number of nodes assigned to a level smaller than $lev(i)$ is less than a threshold δ, we scan the lists V_0 to $V_{lev(i)-1}$ for the cheapest connection. Otherwise, we make use of a precomputed nearest neighbor list for node i, which contains all nodes adjacent to i in increasing edge cost order. The first node in this nearest neighbor list at a level less than $lev(i)$ is chosen as predecessor for node i. In preliminary experiments $\delta \approx 0.1 \cdot n$ turned out to be a good choice in our implementation.

All neighborhoods operating on this level structure have to take into account that the number of nodes assigned to the lowest level 0 is restricted. This level contains the center node(s) of the BDMST, therefore only exactly one node in the even diameter case, and two nodes connected via the center edge if the diameter is odd.

4.3.1 Level Change Neighborhood

In the level change neighborhood an adjacent solution is reached by incrementing or decrementing the level of exactly one node v at level $1 \leq lev(v) \leq H-1$ and $2 \leq lev(v) \leq H$, respectively, and reapplying the decoding procedure presented in Algorithm 10; see Fig. 4.5.

If a node v at level l is connected to a predecessor u assigned to a level smaller than $l-1$, as it is allowed by the decoding procedure, it is advantageous to reduce $lev(v)$ further by consecutive moves until $lev(v) = lev(u)+1$ because v can act as potential predecessor for more nodes. This is accomplished by accepting decrement moves towards the center of the BDMST even if they have no immediate impact on the objective value.

The size of this neighborhood is $\mathcal{O}(n)$ and an exhaustive examination can be implemented in time $\mathcal{O}(n^2)$. Incremental evaluation speeds up the computation substantially in practice but does not reduce this worst-case time complexity.

Incremental Enumeration of Moves within the Level Change Neighborhood

For each node $v \in V$ two possible moves have to be considered, a decrement move, shifting the node from a level $l+1$ down to level l thus bringing it closer to the center of the BDMST, and the oppositely directed move, an increment move. Evaluating a decrement move of a node v includes finding a new predecessor for v if the current one is at level l, and to check if nodes at level $l+1$ can now have v as cheaper predecessor. An increment move of v from level $l-1$ to l can only be profitable in case v can connect much cheaper to a new predecessor at level $l-1$ thus compensating that all old successors of v at level l have to find new predecessors (v was the cheapest available predecessor for these nodes, now they have to connect to other nodes at – in general – higher costs).

After exploring the whole neighborhood once the goal is to find consecutively the currently best improvement move and execute it until a local optimum is reached, i.e., no further improvement moves within the level change neighborhood are available. Shifting a node one level up or down has effects on only some of the previously evaluated moves. For the affected nodes these moves simply can be recomputed from scratch, in general already saving a lot of computational effort. However, this neighborhood allows for a much more fine grained reevaluation of moves, which will be described in more detail in the following.

When exploring the whole neighborhood at the beginning, all found improvement moves can be stored within a priority queue Q. This queue is sorted according to the improvement $\Delta \geq 0$, the value that the current costs of the BDMST can be decreased when the corresponding move is applied. Since for each node there can at most be two improvement moves stored within the queue, a decrement and an increment move, its size is bounded by $\mathcal{O}(2n)$.

The record stored for a move in Q contains the node v to be moved, the direction, the corresponding improvement Δ, and a list L of all new connections implied by this move, i.e., a list of pairs of nodes, where one pair describes a new relationship of a predecessor and a successor. The latter list is not mandatory but once again accelerates the computation significantly since one of the most time consuming procedures is to identify for a node the cheapest possible predecessor at a lower level, cf. Algorithm 10. This has to be done already when evaluating a move. To avoid that the same computations have to be performed once again when an improvement move is executed all this information can be cached in the mentioned list. The first pair in this list is the moving node v and its new predecessor (in case this move really implies a new predecessor for v) because this information will be required often when moves in the queue have to be updated.

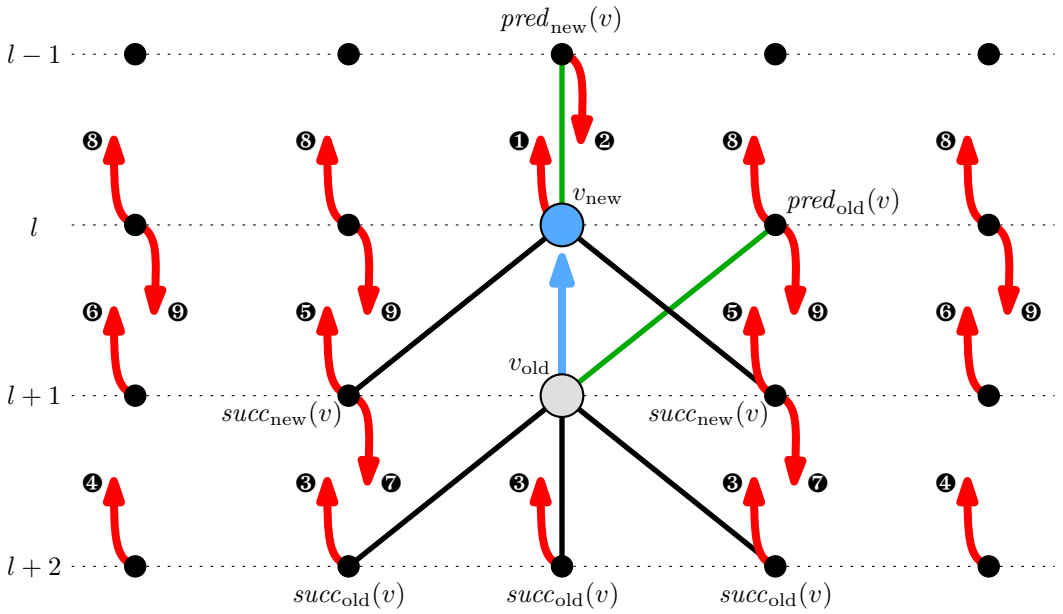


Figure 4.6: Incremental enumeration of the level change neighborhood: Moves that have to be reevaluated after a decrement move of node v from level $l+1$ to level l towards the center of the BDMST. The numbers ❶,...,❾ denote the items in the corresponding description of the various moves.

After extracting and executing the best improvement move from Q, two different cases have to be distinguished in order to have again all possible improvements in the priority queue.

Case 1: Decrement move. The performed move decremented the level of node v from level $l+1$ to l. In case for v also an increment move was filed in Q this entry now can be deleted without any further investigations. Additionally, this move leads to various rechecks and updates in Q, cf. Fig. 4.6:

❶. Decrement move of v to level $l-1$:

This move was not possible before and therefore has to be evaluated completely new.

❷. Increment move of the new predecessor of v, $pred_{new}(v)$, if its current level is $l-1$ and if this move can already be found in the priority queue Q:

The case that $pred_{new}(v)$ loses v as its successor (v no longer can connect to $pred_{new}(v)$ if the new predecessor is moved to the same level) is new. So a

new predecessor for v has to be found, this information must be added to the list of predecessor and successor relationships stored for this move and the improvement Δ has to be updated accordingly.
Since the decrement move of v can only reduce the gain of shifting $pred_{\text{new}}(v)$ to level l this move has only to be checked if it was profitable before, i.e., an increment move of $pred_{\text{new}}(v)$ was already stored in Q.

If the improvement Δ becomes negative after the update, the corresponding move can be removed from the priority queue. This operation has to be performed in all following cases whenever Δ becomes less zero and will no longer be explicitly mentioned.

❸. Decrement move of all old successors of v, $succ_{\text{old}}(v)$, at level $l + 2$:

These nodes can now move one level towards the center without losing their cheapest available predecessor v. If such a move is already in the queue only the predecessor of the corresponding $succ_{\text{old}}(v)$ and so the improvement Δ have to be updated. Otherwise, the move has to be newly evaluated with the exception that there is no need to search for a new predecessor for $succ_{\text{old}}(v)$.

❹. Decrement move for each node u at level $l+2$ having its predecessor $pred(u) \neq v$ at level $l + 1$:

When performing such a move node u would lose its current predecessor and v might now become its new one which was not possible before. If such a move can be found in the priority queue Q only the predecessor of u stored in this move has to be checked against v and in case v would be the better choice the move (predecessor of u and Δ) has to be updated accordingly. Otherwise, the decrement move of u has to be evaluated completely new.

❺. Decrement move for each new successor of v, $succ_{\text{new}}(v)$, if such a move is already filed in the priority queue.

Before the move of v to level l a node $succ_{\text{new}}(v)$ had another predecessor p, so the calculated Δ of this move was based on the connection between $succ_{\text{new}}(v)$ and p. Now $succ_{\text{new}}(v)$ has v as its predecessor (the costs $c_{succ_{\text{new}}(v),v}$ has to be less than $c_{succ_{\text{new}}(v),p}$). Furthermore, the stored move can contain in L the information that $succ_{\text{new}}(v)$ would have become the new predecessor of v at its old level $l + 1$, after moving v to level l this is no longer possible. Finally, the move can contain new successors for $succ_{\text{new}}(v)$ now connected to v which have to be checked if still $succ_{\text{new}}(v)$ would be the cheaper choice. So there are three possible impacts on Δ and the list L of predecessor and successor relationships of the stored move record which have to be updated accordingly.

❻. Decrement move for each node u at level $l + 1$ not being a new successor of v if there is already a corresponding move in Q:

This is more or less case 5 with the restriction that the predecessor of u did not change after moving v to level l. So there are only two possible impacts on Δ and L to be considered.

❼. Increment move for each new successor of v, $succ_{\text{new}}(v)$, if such a move can already be found in the priority queue.

Since the only possibility for an increment move to be profitable is that afterwards itself can be connected much cheaper, two cases have to be distinguished:

1. In the move stored in Q for $succ_{\text{new}}(v)$ node v should have become the new predecessor for $succ_{\text{new}}(v)$. After moving v to level l this connection already has been established so the move can be deleted from Q without any further investigations (after incrementing the level of $succ_{\text{new}}(v)$ v was the cheapest available predecessor, so there cannot be another opportunity to make this increment move valuable).
2. Otherwise, $succ_{\text{new}}(v)$ is now connected cheaper than it was before when the move was evaluated so Δ has to be updated accordingly.

❽. Decrement move for each node u at level l, excluding v, if u would be a cheaper predecessor for v than its current one, i.e., $c_{v,u} < c_{v,pred_{\text{new}}(v)}$:

If such a decrement move for u is already filed in Q only the new connection between u and v has to be added (including an update of Δ), otherwise the move has to be computed from scratch.

❾. Increment move for each node u at level l, excluding v:

Node v now could act as new predecessor for u and for all old successors of u at level $l+1$ losing their predecessor when this increment move is executed. If no such move exists in the priority queue it has to be evaluated completely new, otherwise the following checks and updates of the stored increment move for u are necessary:

1. The move contains another new predecessor w for u. If v would be a cheaper predecessor than w update the corresponding information in L and the improvement Δ.
2. The old successors of u at level $l+1$ will lose their predecessor. For all of these nodes there will be a new predecessor stored in L of the move record.
   - In the meantime (after performing the decrement move of v) it is possible that a former old successor s of u is now successor of v. In this case s will no longer lose its predecessor, the corresponding information stored in the move record of u must be removed and Δ has to be updated.

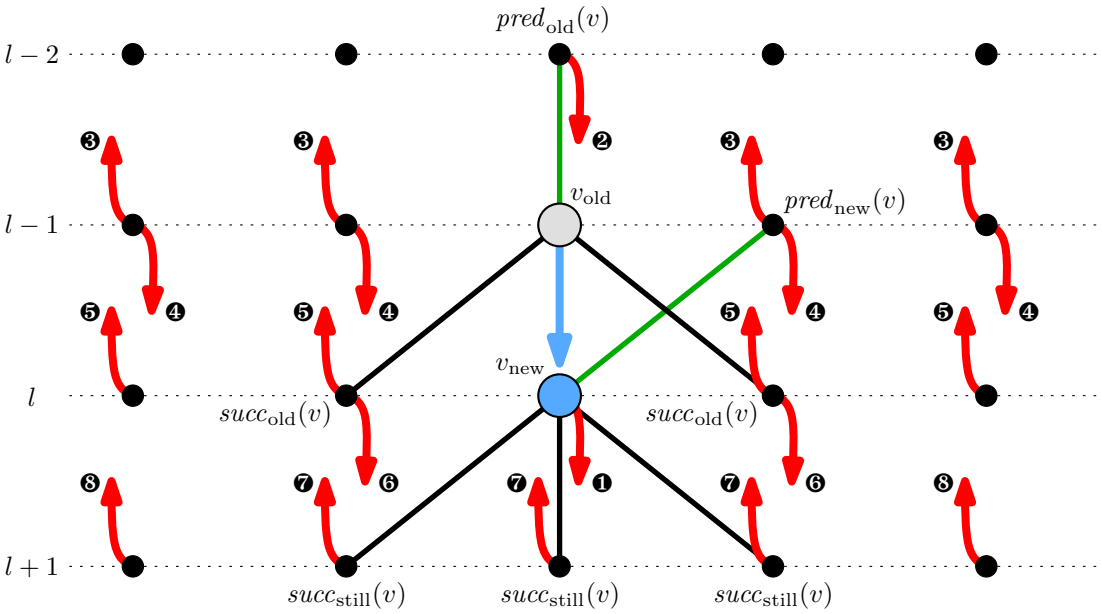


Figure 4.7: Incremental enumeration of the level change neighborhood: Moves that have to be reevaluated after an increment move of node v from level $l-1$ to level l. The numbers ❶,...,❽ denote the items in the corresponding description of the various moves.

- For all nodes still successors of u (after performing the move of v) the new predecessor stored in the move now has to be compared to v if v would be a better choice, and if so the corresponding information and Δ requires an update.
- In case u is the old predecessor of v ($pred_{\text{old}}(v)$) there will be a new predecessor for v stored in list L of the move record. This information can be removed and Δ has to be updated accordingly.

Case 2: Increment move. The performed move incremented the level of node v from level $l-1$ to l. If a decrement move for v can be found in Q it can be deleted instantly. In addition, again some tests and updates have to be performed, see Fig. 4.7:

❶. Increment move of v to level $l+1$:

Continue to move node v towards the leaves. This move was not possible before and therefore has to be computed from scratch.

❷. Increment move of the old predecessor of v, $\text{pred}_{\text{old}}(v)$, if its level equals $l-2$:

Node v will now not lose its predecessor. If the move can already be found in the priority queue only the objective value gain Δ will change and the new connection of v to another predecessor can be deleted from L. Otherwise, the move has to be evaluated completely new.

❸. Decrement move for each node u at level $l-1$ if such a move is already stored in the priority queue Q and $c_{v,u} < c_{v,pred_{\text{old}}(v)}$:

If this move is not in the queue it still has not to be considered, nothing changed to make this move profitable. Contrarily, if such a move can be found in Q then the objective value gain Δ includes the benefit of changing the predecessor of v from $pred_{\text{old}}(v)$ to u (note the precondition given above). Therefore, Δ and the fact that v is already connected to u have to be updated in the decrement move record of u.

❹. Increment move for each node u at level $l-1$ if such a move is already part of the priority queue Q:

Shifting node v to level l has various effects on already exiting increment moves of other nodes u from level $l-1$ to l:

1. Node v is no longer available as predecessor for u and for all old successors of u at level l.
2. Some old successors of v, $succ_{\text{old}}(v)$, as well as v itself can have u as their new predecessor and so would require to find a new node they can connect to.

All these have to be considered and appropriate updates have to be performed on the stored move record.

❺. Decrement move for each node u at level l, excluding v:

In case such a move is found in the priority queue Q two things have to be checked:

1. Node v could become a new successor of u if the condition $c_{u,v} < c_{pred_{\text{new}}(v),v}$ holds true.
2. Old successors of v at level l have lost their predecessor. Node u could now be attractive as new predecessor if it was not already before.

In both cases the objective value gain Δ and the list L of predecessor and successor relationships of the move record have to be updated accordingly.

If such a move was not stored in Q it must be reevaluated. For efficiency reasons the computation can be split into two steps: The only changes are the two cases described above (u possibly a better predecessor for v and old successors of v). If the evaluation of these changes do not lead to a positive Δ

the move could not have become valuable after moving v from level $l-1$ to l and so the computation can already be stopped at this point.

❻. Increment move for all old successors of v, $succ_{\text{old}}(v)$, at level l, if no such move is stored in Q:

After shifting node v from level $l-1$ to l the old successors of v were forced to connect to new predecessors at higher costs. Incrementing the level of an old successor $succ_{\text{old}}(v)$ from l to $l+1$ allows it to get back its old and cheaper predecessor v.

If such a move cannot be found in the priority queue it has to be evaluated completely new. Otherwise, nothing has to be done, since an increment move can only be profitable if the shifting node can connect to a cheaper predecessor. Consequently, because such a move already exists in Q there have to be a better predecessor for $succ_{\text{old}}(v)$ at level l than v was before. In addition, all successors of $succ_{\text{old}}(v)$ at level $l+1$ now losing its predecessor still can connect to v, so the situation not substantially changed since evaluating the move, no updates are required.

❼. Decrement move for all successors of v, $succ_{\text{old}}(v)$, at level $l+1$ if such a move already can be found in the priority queue:

Before moving v one level up to l a successor node at $l+1$ could keep v as its predecessor when moving one level down. Now this is no longer possible, a new predecessor has to be found leading to a different objective value gain Δ. The new predecessor for $succ_{\text{old}}(v)$ also has to be inserted into L of the corresponding move record in Q.

❽. Decrement move for each node u at level $l+1$, excluding all successors of v, but with a predecessor at level l if such a move is stored in the priority queue Q and u would have become successor of v:

When executing this decrement move u no longer can connect to v, therefore a new predecessor has to be found for u, the improvement Δ and L have to be updated accordingly.

After giving these detailed rules to incrementally enumerate the level change neighborhood it has to be noted that it does not always make sense to implement all of them. Depending on various factors like the actual implementation of the used data structures (e.g., the priority queue and its direct access operators) it could be sometimes more time consuming to test all conditions than simply evaluating the complete move from scratch. Thus appropriate experiments have to be performed, and also a compromise – from a software engineers' point of view – between highly complex and simple, reliable, and maintainable code has to be found.

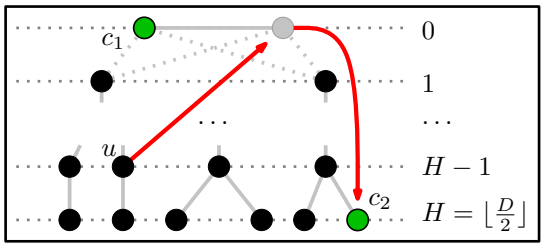


Figure 4.8: Center exchange level neighborhood: Replacing a center node and reconnecting the tree according to a level neighborhood.

4.3.2 Center Exchange Level Neighborhood

The level change neighborhood never affects the center node(s). In order to fill this gap, we use the center exchange level neighborhood. It replaces exactly one center node by any non-center node u. The replaced center node is set to level H, maximizing the number of potential predecessors, see Fig. 4.8.

To better exploit the potential of such a center exchange a local improvement step is immediately appended: As long as there exists a node w whose predecessor v has level $lev(v) < lev(w) - 1$, we assign node w to level $lev(v) + 1$. Following such a reduction, node w can serve as potential predecessor for a larger number of other nodes and – as a consequence – cheaper connections might be enabled.

Restricting the exchange to exactly one center node leads to a neighborhood size of $\mathcal{O}(n)$. A local improvement step requires in the worst-case time $\mathcal{O}(n^2)$, yielding a total time complexity of $\mathcal{O}(n^3)$ for evaluating the whole neighborhood.

4.4 Clustering-Based Neighborhood

As already mentioned in Chapter 3 when discussing greedy construction heuristics a well structured BDMST – especially in case of Euclidean instances – usually has a few but long edges near the center, the backbone, so the majority of the nodes can connect using relatively short edges. A height restricted, agglomerative, hierarchical clustering of all instance nodes can act as basis for such a structured tree, where the actual tree is derived by selecting for each cluster a good root for the subtree represented by it. Since a valid BDMST solution describes such an height restricted clustering in a natural way, it can easily be transformed into this representation and a new, probably better tree can be determined from the hierarchical clustering.

Deriving a tree of high quality from a given clustering is a multi-stage process. It is discussed in full detail in Chapter 8 where also a construction heuristic based on clustering is presented.

Chapter 5

Level-Based Integer Linear Programming Approach

5.1 Introduction

Prior exact approaches for determining a diameter constraint minimum spanning tree mostly rely on Miller-Tucker-Zemlin (MTZ) inequalities [43] and network flow-based integer linear programming formulations [3, 4]. In particular hop-indexed multi-commodity flow formulations by Gouveia et al. [59, 60] have been very successful. The lower bounds achieved with the linear programming (LP) relaxations of their models are extremely tight, however, at the expense of a large number of flow variables of order $\mathcal{O}(|E| \cdot |V| \cdot D))$.

In this chapter we present a new 0–1 integer linear programming model involving only $\Theta(|E| + |V| \cdot D)$ variables and $\Theta(|E| \cdot D)$ constraints. It is strengthened by dynamically adding violated connection and cycle elimination constraints within a Branch&Cut environment. The proposed approach is empirically compared to two state-of-the-art models based on network flows and the MTZ formulation, respectively.

5.2 A Compact 0–1 ILP Formulation

Like most of the existing formulations, we view the task of finding a BDMST as a directed graph problem. However, we do not introduce any artificial nodes. Let A be the arc set derived from E by including for each undirected edge $(i, j) \in E$ two oppositely directed arcs (i, j) and (j, i) with the same costs $c_{i,j} = c_{j,i}$. In our formulation of the BDMST only the two center nodes in case of an odd diameter are connected by an undirected edge. All other edges are directed in such a way that there exists a directed path from the center to any other node. Let the *depth* $d(v)$ of a node $v \in V$ be the number of arcs on the path from the center to node v. The center node(s) have depth $d(v) = 0$. The height of this rooted tree is then at most $H = \lfloor \frac{D}{2} \rfloor$. The BDMST can be seen as a variation of the hop constrained minimum spanning tree problem with a root not fixed in advance [57].

We formulate the BDMST problem by assigning each node v a depth $0 \leq d(v) \leq H$. Furthermore, each node of depth $d(v) > 0$ has to have one *predecessor* denoted by $pred(v)$. In order to get a feasible spanning tree, $d(pred(v)) = d(v) - 1$ must hold for all nodes $v \in V \mid d(v) > 0$.

The ILP uses the following variables:

- $u_{i,l} \in \{0, 1\}$; $i \in V$, $l = 0, \ldots, H$: $u_{i,l} = 1 \ \leftrightarrow d(i) = l$.
- $p_{i,j} \in \{0, 1\}$; $(i, j) \in A$: $p_{i,j} = 1 \ \leftrightarrow \ pred(j) = i$, i.e., $(i, j) \in E_T$.
- if D is odd:
  $r_{i,j} \in \{0, 1\}$; $(i, j) \in E$: $r_{i,j} = 1 \ \leftrightarrow$ edge $(i, j) \in E_T$ and connects the center nodes.

5.2.1 The Even Diameter Case

In the simplest form, the even diameter case can be modeled as follows:

$$\text{minimize} \qquad \sum_{(i,j) \in A} p_{i,j} \cdot c_{i,j} \tag{5.1}$$

$$\text{subject to} \quad \sum_{l=0}^{H} u_{i,l} = 1 \quad \forall i \in V \tag{5.2}$$

$$\sum_{i \in V} u_{i,0} = 1 \tag{5.3}$$

$$\sum_{i|(i,j)\in A} p_{i,j} = 1 - u_{j,0} \quad \forall j \in V \tag{5.4}$$

$$p_{i,j} \leq 1 - u_{j,l} + u_{i,l-1} \quad \forall (i,j) \in A,\ \forall l = 1, \ldots, H. \tag{5.5}$$

The objective (5.1) is to minimize the total costs of all used arcs ($p_{i,j} = 1$). Equations (5.2) ensure that each node i is assigned to exactly one depth. Depth zero must be assigned to exactly one node, the center (5.3). Constraints (5.4) guarantee that each node except the center always has exactly one predecessor. Finally, we have to model the fact that node i can only be predecessor of node j if $d(i) = d(j) - 1$, or in other words $p_{i,j} = 1 \rightarrow \exists\, l \mid u_{j,l} = 1 \wedge u_{i,l-1} = 1$. This is assured by inequalities (5.5).

To strengthen the LP-relaxation we further add constraints which explicitly express that a node assigned to depth zero cannot have a predecessor, nor can a node assigned to depth H be predecessor of any other node:

$$u_{j,0} \leq 1 - p_{i,j} \quad \text{and} \quad u_{i,H} \leq 1 - p_{i,j} \quad \forall (i,j) \in A. \tag{5.6}$$

5.2.2 The Odd Diameter Case

In case the diameter D is odd, we can formulate the BDMST problem as follows:

$$\text{minimize} \quad \sum_{(i,j)\in A} p_{i,j} \cdot c_{i,j} + \sum_{(i,j)\in E} r_{i,j} \cdot c_{i,j} \tag{5.7}$$

$$\text{subject to} \quad \sum_{i \in V} u_{i,0} = 2 \tag{5.8}$$

$$\sum_{j|(i,j)\in E} r_{i,j} = u_{i,0} \quad \forall i \in V \tag{5.9}$$

Constraints (5.2), (5.4), (5.5) and (5.6) are adopted unchanged.

Starting from the ILP (5.1) to (5.6) we have to adapt the objective function (5.1) to include the costs of the edge connecting the two center nodes (5.7) as well as equation (5.3) to reflect the fact that now there are two nodes assigned to depth zero (5.8). In addition, constraints (5.9) are required in order to get the edge connecting the two center nodes at depth zero.

Equations (5.9) together with (5.8) imply the following constraint that does not actually strengthen the LP-relaxation, but nevertheless speeds up the integer optimization in practice as our experiments indicate:

$$\sum_{e \in E} r_e = 1. \tag{5.10}$$

Finally, Gouveia and Magnanti [59] suggested to exploit the fact that in a BDMST with odd D, the nodes of depth one are always connected to the nearer one of the two center nodes. We consider this aspect in the following way.

For each arc $(i, j) \in A$, let $L(i, j)$ be the set of all potential center edges when i is one of the center nodes, j appears at depth one, and i is j's predecessor:

$$L(i, j) = \{(i, i') \in E \mid j \neq i' \wedge (c_{i,j} < c_{i',j} \vee c_{i,j} = c_{i',j} \wedge i < i')\}. \tag{5.11}$$

If $u_{j,1} = 1$, $p_{i,j}$ may only be set to one if the center edge appears in $L(i, j)$:

$$p_{i,j} \leq 1 - u_{j,1} + \sum_{e \in L(i,j)} r_e \quad \forall (i, j) \in A. \tag{5.12}$$

5.3 Branch&Cut

Achuthan et al. [4] already described the possibility to include constraints for explicitly avoiding cycles in order to strengthen the LP-relaxation. Unfortunately, considering each possible cycle and adding an appropriate inequality from the very beginning – as Achuthan et al. did – leads to a huge number of constraints which increases exponentially with D. This limits the applicability to very small diameters and/or instances.

We suggest a Branch&Cut approach, in which connectivity and cycle elimination constraints violated in the solution of the LP-relaxation are iteratively determined and added as cuts throughout the optimization process. These sorts of cuts are often used in cutting plane algorithms for various network design problems, such as the traveling salesman problem [102].

5.3.1 Connection Cuts

A spanning tree always has to be connected, i.e., each non-empty subset S of all nodes V has to induce a cut $\delta(S)$ of size greater than or equal to one. More formally,

$$\sum_{(i,j)\in\delta(S)} (p_{i,j} + r_{i,j}) \geq 1 \quad \forall S \subset V,\, S \neq \emptyset, \tag{5.13}$$

where variables $r_{i,j}$ are only required in case the diameter is odd. Using a max-flow/min-cut algorithm, it is easy to identify a subset of nodes for which the above constraint is violated in the solution of the LP-relaxation or to prove that no such set exists.

5.3.2 Cycle Elimination Cuts

In a tree no cycle may appear, and therefore we can consider the constraints

$$\sum_{(i,j)\in C} p_{i,j} + p_{j,i} + r_{i,j} \leq |C| - 1 \quad \forall \text{ cycles } C \subset E, \tag{5.14}$$

where again variables $r_{i,j}$ only have to be considered in the odd diameter case. We separate such cuts by constructing an undirected graph from the solution of the LP-relaxation with node set V, edge set $E'_T = \{(i,j) \in E \mid p_{i,j} + p_{j,i} + r_{i,j} > 0\}$, and with associated costs $\max(0, 1 - p_{i,j} - p_{j,i} - r_{i,j})$. A cycle violating the corresponding constraint has total costs less than 1 and can be identified by calculating the shortest path for each pair of nodes i and j with $(i,j) \in E'_T$.

From a theoretical point of view, connection and cycle elimination cuts are both covered by the more general sub-tour elimination cuts, where C is replaced by the set of all edges induced by a subset of vertices from V [104]. However, finding general sub-tour elimination cuts is much more time demanding than identifying violated connection and cycle constraints. Therefore, it is usually a good idea to first separate connection and cycle elimination constraints, before considering general sub-tour elimination cuts. We have not yet implemented the latter and leave them for future research. Since the separation of connection cuts is faster than looking for cycle elimination cuts, we first separate all violated connection constraints and only then calculate cycle elimination cuts when using both.

5.4 Computational Results

We compare our approach to the MTZ-based formulation of Santos et al. [43] (variant (B) and (C) for even and odd diameters, respectively) and the MCF formulation of Gouveia and Magnanti [59] (HopDMCF and Enh-HopMCF, respectively). The same test instances as in these previous works are used, which have been created the following way: The integral edge costs for random instances have been chosen from the interval $[1, 100]$ using a uniform distribution. For Euclidean instances, the integer coordinates of the nodes have been computed randomly within a 100×100 square grid, whereas for the edge weights just the integer parts of the Euclidean distances have been utilized. From these complete graphs, Gouveia and Magnanti only used the $|E|$ cheapest edges for their tests. To ensure that their sparse instance graphs have feasible solutions, Santos et al. built them in a special way by first computing a minimum spanning tree with a diameter bound of $D = 2$ (a star), and afterwards augmenting this tree with the least cost edges until reaching the desired number of edges for the instance. Our experiments were performed on a Pentium®4 2.8 GHz system with 2 GB of RAM using Linux 2.4.21 and CPLEX 8.1 under default parameters as MIP solver.

Table 5.1 lists CPU times for finding optimal solutions and proving optimality on complete and sparse instances for our reimplementation of Santos et al. and our ILP formulation. Variants marked by '+' apply connection cuts, whereas those marked by '$*$' use cycle elimination cuts. Enclosed in parentheses we list the percentage gap between the optimal solution (opt) and the LP-relaxation (lb) at the first node within the Branch&Cut tree after performing all separations of our own and the standard cuts of CPLEX: $gap = (opt - lb)/opt \cdot 100\%$. In case there are two values listed the first one depicts the gap at the very beginning only depending on the model used and so invariant in terms of applying cuts.

In general, no approach dominates any other. However, it can be observed that the ILP formulation performs better on tighter constrained instances with rather small diameters, whereas the model of Santos et al. becomes faster than the ILP for looser diameter constraints.

Applying our cuts to the different formulations lead to ambivalent results, except on sparse instances where connection cuts show significantly better performance. It also turns out that the ILP model in general benefits more from cycle elimination cuts, in particular on bigger instances with larger diameters.

Concerning the listed LP-relaxation gaps we further remark that in addition to our connection and cycle elimination cuts, CPLEX sometimes adds additional general-purpose cuts, which further reduce the LP-bound. In a few cases, this leads to the

Table 5.1: CPU times (in seconds) and LP-relaxation gaps on Euclidean instances from Santos et al. [43].

$\|V\|$	$\|E\|$	D	Santos	Santos+	Santos∗	Santos+∗	ILP	ILP+	ILP∗	ILP+∗
15	105	4	4.7 (37.4/23.9)	9.8 (22.4)	18.88 (21.0)	29.9 (20.1)	**0.7** (36.4/22.0)	0.9 (25.1)	1.1 (19.7)	1.7 (20.1)
15	105	5	22.8 (33.8/28.9)	36.9 (25.5)	44.1 (20.8)	78.7 (18.2)	**3.0** (33.6/29.2)	3.2 (25.3)	5.8 (19.2)	7.0 (18.1)
15	105	6	21.1 (31.5/19.7)	18.6 (16.4)	52.6 (16.9)	48.6 (13.7)	**8.1** (38.9/32.8)	24.1 (25.0)	15.6 (16.3)	33.6 (13.7)
15	105	7	44.8 (28.1/23.7)	29.3 (18.0)	38.8 (13.9)	26.9 (10.6)	26.4 (33.6/28.3)	37.2 (28.3)	20.5 (14.0)	**20.0** (10.6)
15	105	9	18.4 (24.8/21.1)	16.5 (15.1)	15.8 (10.4)	**6.2** (6.6)	150.6 (32.5/26.8)	47.0 (21.1)	20.7 (10.3)	10.7 (6.6)
15	105	10	1.5 (24.8/12.1)	**1.0** (8.3)	5.1 (9.2)	2.4 (5.3)	65.8 (36.1/29.1)	43.3 (20.6)	15.8 (9.2)	4.3 (5.3)
20	190	4	562.9 (29.7/24.4)	1,063.6 (21.6)	1,232.5 (24.1)	4,315.1 (21.4)	**2.5** (28.8/25.0)	3.3 (21.3)	3.7 (22.3)	5.0 (20.2)
20	190	5	436.7 (25.2/20.9)	662.5 (20.3)	572.9 (17.4)	1,260.6 (17.3)	**8.1** (27.4/17.8)	10.2 (17.8)	9.3 (15.2)	12.8 (15.6)
20	190	6	577.0 (20.3/13.8)	489.3 (13.3)	455.2 (12.5)	531.9 (12.1)	**95.0** (28.2/21.7)	210.5 (19.7)	396.4 (12.5)	382.8 (12.5)
20	190	7	8.1 (13.4/9.7)	5.1 (7.7)	7.8 (5.9)	10.2 (5.1)	10.0 (19.3/11.3)	12.9 (8.3)	**4.5** (5.0)	7.2 (5.0)
20	190	9	241.9 (22.2/18.5)	105.7 (14.4)	244.3 (11.2)	73.4 (8.9)	1,209.1 (30.0/24.4)	923.7 (18.6)	194.6 (11.0)	**66.7** (8.8)
20	190	10	64.6 (21.9/15.4)	41.9 (10.6)	205.1 (11.2)	**29.7** (8.1)	13,755.5 (34.0/26.1)	13,972.1 (18.4)	226.4 (11.3)	101.0 (8.1)
25	300	4	15,203.7 (30.3/26.6)	> 20,000.0 (25.9)	> 20,000.0 (26.1)	> 20,000.0 (25.1)	**12.0** (31.2/23.2)	14.2 (22.9)	16.5 (22.2)	20.3 (21.7)
25	300	5	> 20,000.0 (32.4/28.7)	> 20,000.0 (25.1)	> 20,000.0 (25.5)	> 20,000.0 (22.6)	76.3 (33.5/27.4)	**64.3** (23.1)	102.7 (23.7)	127.3 (20.9)
25	300	6	1,282.5 (18.6/12.4)	826.7 (11.4)	1,241.7 (10.9)	1,151.3 (10.1)	**26.4** (28.4/17.0)	166.7 (13.9)	75.8 (10.6)	146.1 (10.1)
25	300	7	11,521.3 (18.7/15.9)	> 20,000.0 (14.0)	> 20,000.0 (13.3)	17,014.1 (11.4)	**770.5** (26.5/17.6)	2,719.3 (14.9)	1,090.3 (13.0)	989.6 (11.6)
25	300	9	> 20,000.0 (22.7/18.4)	**246.0** (8.0)	13,677.6 (15.5)	429.7 (5.3)	> 20,000.0 (31.3/22.7)	2160.1 (11.9)	4,143.3 (15.4)	295.4 (5.2)
25	300	10	278.2 (10/8.6)	**254.8** (5.9)	401.4 (7.0)	327.0 (5.3)	3,666.1 (24.0/12.6)	2,904.2 (9.2)	1,127.9 (7.2)	404.9 (5.3)
20	50	4	1.0 (32.9/19.1)	1.0 (16.5)	4.7 (18.8)	3.8 (16.4)	**0.2** (29.0/13.9)	**0.2** (13.4)	0.3 (13.8)	0.4 (13.2)
20	50	5	4.6 (60.7/57.8)	9.0 (52.5)	15.6 (54.6)	23.7 (51.2)	**1.0** (62.1/58.8)	2.5 (53.6)	3.5 (53.8)	4.9 (50.6)
20	50	6	34.6 (28.9/20.8)	**0.8** (9.5)	45.6 (15.2)	1.5 (8.8)	8.7 (35.1/27.2)	5.1 (11.6)	19.9 (15.2)	6.0 (8.8)
20	50	7	13.3 (26.1/22.4)	**0.8** (8.4)	13.3 (20.8)	1.2 (7.9)	1.2 (27.4/25.5)	1.7 (7.8)	2.5 (19.5)	3.0 (7.4)
20	50	9	76.2 (24.3/19.5)	**0.7** (7.7)	108.5 (14.5)	0.8 (4.6)	42.5 (31.8/24.9)	2.8 (10.4)	25.6 (14.4)	3.7 (4.6)
20	50	10	98.5 (29.5/21.7)	**0.2** (3.9)	187.7 (16.8)	**0.2** (2.2)	505.3 (40.4/32.8)	1.8 (7.4)	79.1 (16.7)	1.3 (2.2)
40	100	4	43.7 (41.9/27.7)	82.4 (23.5)	516.5 (24.6)	2,352.1 (23.0)	**1.9** (39.6/20.4)	**1.9** (19.4)	8.5 (19.0)	10.0 (18.5)
40	100	5	471.0 (65.4/56.7)	291.7 (52.9)	1,646.5 (56.1)	893.2 (52.3)	**6.4** (62.9/47.5)	6.8 (46.9)	13.1 (46.9)	18.2 (46.7)
40	100	6	1,991.6 (29.9/25.7)	50.9 (4.6)	13,719.8 (23.3)	100.2 (4.1)	182.9 (44.8/30.2)	**13.2** (6.4)	449.6 (23.3)	37.4 (4.6)
40	100	7	> 20,000.0 (36.6/21.1)	459.4 (14.7)	3,731.7 (20.4)	10,224.2 (13.4)	**212.4** (45.5/34.5)	4,463.2 (15.2)	525.2 (20.9)	9455.3 (13.2)
40	100	9	14,882.5 (27.7/23.3)	1,565.0 (15.8)	13,078.0 (21.1)	4,262.7 (14.5)	3,828.7 (39.2/28.2)	9,974.3 (16.9)	**979.8** (21.1)	5137.2 (14.6)

Table 5.2: CPU times (in seconds) and LP-relaxation gaps on incomplete instances from Gouveia and Magnanti [59] (gaps for G&M at the beginning of the first Branch&Cut node without CPLEX generated cuts).

	$\|V\|$	$\|E\|$	D	G&M	ILP	ILP+	ILP∗	ILP+∗
Random	20	100	4	**0.5** (0.0)	0.9 (35.8/23.9)	0.9 (23.9)	1.5 (24.3)	1.6 (23.4)
	20	100	5	6.3 (0.0)	2.7 (35.5/27.0)	**2.4** (26.9)	**2.4** (25.6)	3.2 (25.6)
	20	100	6	5.8 (1.0)	**2.9** (34.7/21.0)	7.2 (20.6)	5.9 (16.3)	7.9 (18.0)
	20	100	7	94.0 (1.5)	10.6 (31.6/17.1)	5.7 (16.2)	**2.6** (13.1)	15.8 (13.2)
	20	100	8	**1.3** (0.0)	4.4 (28.1/10.0)	4.3 (9.4)	3.4 (5.8)	5.7 (5.8)
Random	30	200	4	**0.8** (0.0)	3.5 (39.8/30.1)	6.1 (29.7)	6.3 (29.8)	8.7 (29.5)
	30	200	5	**58.6** (0.0)	377.8 (48.1/41.9)	283.8 (41.9)	428.1 (42.1)	482.4 (41.6)
	30	200	6	**2.9** (0.0)	5.7 (28.4/19.4)	20.4 (14.9)	11.5 (14.6)	39.6 (13.5)
	30	200	7	529.4 (0.0)	112.4 (21.3/16.6)	261.1 (16.3)	**53.9** (14.3)	100.8 (14.2)
	30	200	8	**2.3** (0.0)	10.6 (18.3/14.3)	13.9 (5.4)	3.67 (8.3)	4.9 (2.2)
Euclidian	20	100	4	**0.1** (0.0)	1.1 (20.3/17.5)	1.4 (17.4)	2.3 (17.2)	2.2 (17.1)
	20	100	5	5.3 (0.0)	**1.7** (16.8/13.2)	2.0 (13.0)	2.5 (12.8)	3.5 (12.6)
	20	100	6	**3.1** (0.2)	7.3 (16.3/10.4)	9.5 (9.9)	15.5 (8.0)	21.2 (8.5)
	20	100	7	49.5 (0)	**10.0** (14.1/8.3)	24.3 (8.2)	11.2 (7.0)	31.3 (7.6)
	20	100	8	**1.1** (0.0)	10.7 (14.2/10.0)	12.9 (7.7)	18.9 (6.8)	31.2 (5.2)
Euclidian	30	200	4	130.8 (1.7)	148.6 (35.2/25.8)	84.9 (26.0)	**59.5** (26.3)	107.0 (25.3)
	30	200	5	**25.1** (0.1)	36.1 (33.9/25.7)	42.2 (26.1)	65.3 (24.1)	61.5 (25.6)
	30	200	6	1,381.9 (0.8)	**348.0** (28.8/19.4)	4,022.7 (17.6)	7,224.2 (16.9)	17,144.3 (15.0)
	30	200	7	6,912.1 (1.2)	1,339.7 (23.5/17.5)	3,713.8 (16.3)	**1,014.4** (13.6)	4,205.4 (13.4)
	30	200	8	**1,111.0** (0.8)	2,864.7 (22.6/16.2)	9,298.8 (12.6)	2,430.6 (11.5)	> 20,000.0 (9.4)

effect that the LP-bound of a variant where only one type of our cuts is used is slightly smaller than when applying both of them.

Table 5.2 shows running times and LP-bounds for the ILP variants and Gouveia and Magnanti's (G&M) approach. The times listed for G&M are adopted from [59] and scaled by a factor of 1/8 to account for different hardware. This factor has been determined by considering the widely used floating point benchmarks published at `http://www.spec.org`. Nevertheless, we remark that this scaling is only a rough estimation and running time comparisons should be taken with care.

Concerning the LP-relaxation gaps it can easily be seen that the ILP model cannot compete with the flow formulations of G&M. When looking at the computation times no single variant dominates any other. The different ILP approaches outperform G&M on several occasions. However, no real pattern can be observed for conditions under which a certain method performs best; speed-up factors are in general not as large as those of the first series of experiments (Table 5.1).

In this context we want to point out that the above results are not sufficient to draw more general conclusions on expected running times for specific classes of

instances. During our tests we experienced highly varying computation times for instances randomly generated all the same way. For example, when running ILP+ on 10 different complete Euclidean graphs with 20 randomly distributed nodes, CPU times ranged from 51 to 54,400 seconds. A similar behavior was observed for the model of Santos et al. and the flow formulations as well.

In addition to the presented results we also made first experiments on instances with 40 nodes and more. As expected due to the observed LP-relaxation gaps the size of the Branch&Cut tree grows fast and so does the computation time. Without further improvements the ILP results will not be competitive to state-of-the-art flow formulations.

We also tried to reduce the initialization overhead always involved when calculating connection or cycle elimination cuts. A number of two to three cuts generated at once (i.e., cuts added to the model before a new LP is solved) has proven to be a good choice.

5.5 Additional Constraints

In this section we present some additional constraints we made experiments with, some of them strengthening the LP-relaxation of the ILP model. However, exhaustive benchmarks led to the conclusion not to include them into the model in general, because over all tested instances the running time behavior was not significantly better.

$$\sum_{(i,j)\in A} p_{i,j} = n - 1 - (D \bmod 2) \tag{5.15}$$

A spanning tree of n nodes contains $n-1$ edges. In case of an odd diameter we have to subtract the center edge encoded separately using the variables $r_{i,j}$. When including this equation to the model the calculation times are undifferentiated; they range from three times faster and more to about two times slower.

$$p_{i,j} \leq 1 - u_{i,l-1} + u_{j,l} \quad \forall (i,j) \in A,\ \forall l = 1, \ldots, H. \tag{5.16}$$

These constraints are a modification of the inequalities (5.5). Replacing (5.5) by them yields a poorer LP-relaxation. Using both types of constraints strengthens the LP-relaxation noticeable, but in either case the running times are substantially higher.

$$p_{i,j} + p_{j,i} \leq 1 \quad \forall (i,j) \in A \tag{5.17}$$

A node i cannot be predecessor and successor of a node j at the same time. These inequalities strengthen the LP-relaxation significantly and often speed up computation for instances with larger diameter (approx. 8 and above) and sparse graphs. For other instances, however, running times are usually increased.

$$p_{i,j} \leq 1 - u_{j,l} + \sum_{l'=0}^{l-1} u_{i,l'} \quad \forall (i,j) \in A,\ \forall l = 1, \ldots, H \tag{5.18}$$

A relaxed alternative for inequalities (5.5), where an edge from a node at depth l may be attached to any node at depth $\leq l-1$, not just $l-1$. This formulation leads to weaker LP-bounds as well as a poorer running time behavior.

$$\delta_{i,j} + (H+1)p_{i,j} + (H-1)p_{j,i} \leq H \qquad \forall (i,j) \in A \qquad \text{with} \tag{5.19}$$

$$\delta_{i,j} = \sum_{l=0}^{H} l \cdot u_{i,l} - l \cdot u_{j,l} \qquad \forall (i,j) \in A \tag{5.20}$$

being the difference of the depths of nodes i and j. These lifted MTZ constraints from [43] strengthen the LP-relaxation, but the computational results do not exhibit a conclusive pattern.

5.6 Conclusions

A compact 0–1 ILP for the BDMST problem has been introduced. When solving this ILP with the general purpose MIP-solver CPLEX, the performance can be improved by separating violated connectivity and/or cycle constraints and adding them in a Branch&Cut manner at each node of the Branch&Bound tree. The new approach has been compared to two state-of-the-art formulations. It turns out that in particular dense instances with small to moderate diameter bounds are solved significantly faster than with the approach from Santos et al. [43]. This MTZ-base model, however, exhibits its strengths on instances where D approaches the diameter of the unconstrained minimum spanning tree.

In [60], Gouveia et al. introduced a specific improvement for the odd diameter case we did not consider so far. Further theoretical investigations and experiments on

more and larger benchmark instances are necessary to get a closer insight into the specific assets and drawbacks of the different formulations.

Besides the implementation of tighter sub-tour elimination cuts as a substitute for the cycle elimination cuts another future objective will be to combine exact algorithms like those discussed here with (meta-)heuristic approaches. This will be done not just in the classical sense – for example by heuristically determining a good starting solution for an exact algorithm – but also by running different algorithms in parallel and letting them exchange information relevant for the optimization in order to benefit from synergy [134].

Chapter 6

Integer Linear Programming Approach Based on Jump Inequalities

6.1 Introduction

The level-based ILP discussed in Chapter 6 reduces significantly the number of required variables in contrast to the multi-commodity hop-indexed flow formulations, but at the expense of weak LP relaxation bounds. Therefore, the gain in runtime and memory usage of having a small, compact model is in general undone by the size of the Branch&Bound tree required to solve a BDMST instance to proven optimality. Strengthening the formulation by applying standard cutting planes like connectivity and cycle elimination cuts within a Branch&Cut environment enhances the situation only marginally.

In [29] Dahl et al. proposed a strong formulation for the HCMST problem based on so-called *jump inequalities* to ensure the hop constraint. Unfortunately, their number grows exponentially with the number of nodes in the instance graph n, and the problem of separating them in a cutting plane algorithm is conjectured to be $\mathcal{NP}$-hard. Therefore, they exploited them in a relax-and-cut algorithm where violated jump inequalities only need to be identified for integer solutions, which is straightforward using depth first search.

In this chapter we present a strong integer linear programming formulation based on these jump inequalities solved by a Branch&Cut algorithm. As the separation subproblem of identifying currently violated jump inequalities is difficult, we approach it heuristically by two alternative construction heuristics, local search, and optionally tabu search. We also introduce a new type of cuts, the center connection cuts, to strengthen the formulation in the more difficult to solve odd diameter case. In addition, primal heuristics are used to compute initial solutions and to locally improve incumbent solutions identified during Branch&Cut.

6.2 The Jump Model

Our ILP model is defined on a directed graph $G^+ = (V^+, A^+)$, with the arc set A^+ being derived from E by including for each undirected edge $(u, v) \in E$ two oppositely directed arcs (u, v) and (v, u) with the same costs $c_{u,v} = c_{v,u}$. In addition, we introduce an artificial root node r that is connected to every other node with zero costs, i.e., $V^+ = V \cup \{r\}$ and $\{(r, v) \mid v \in V\} \subset A^+$. This artificial root allows us to model the BDMST problem as a special directed outgoing HCMST problem on G^+ with root r, hop limit (i.e., maximum height) $H = \lfloor \frac{D}{2} \rfloor + 1$, and the additional constraint that the artificial root must have exactly one outgoing arc in the case of even diameter D and two outgoing arcs in the case D is odd. From a feasible HCMST $T^+ = (V^+, A_T^+)$, the associated BDMST T on G is derived by choosing all edges for which a corresponding arc is contained in A_T^+. In the odd diameter case, an additional *center edge* connecting the two nodes adjacent to the artificial root is further included.

We make use of the following variables: Arc variables $x_{u,v} \in \{0, 1\}$, $\forall (u, v) \in A^+$, which are set to one iff $(u, v) \in T^+$, and center edge variables $z_{u,v} \in \{0, 1\}$, $\forall (u, v) \in E$, which are only relevant for the odd diameter case and are set to one iff (u, v) forms the center of the BDMST.

The even diameter case is formulated as follows:

$$\text{minimize} \qquad \sum_{(u,v) \in A} c_{u,v} \cdot x_{u,v} \tag{6.1}$$

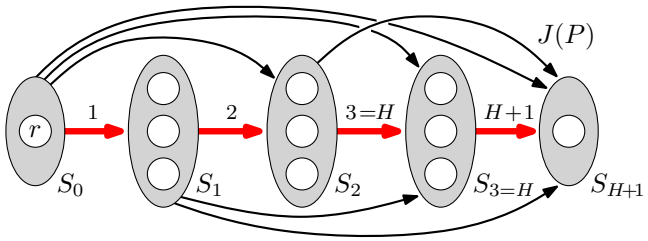


Figure 6.1: Partitioning P of the nodes in V^+ into $H+2$ nonempty sets $S_0, \ldots, S_{H+1}$. The jump $J(P)$ contains all arcs leading from a partition to a higher indexed one skipping at least one in-between (curved arcs). A path connecting the artificial root node r with nodes in S_{H+1} without any arc from $J(P)$ would consist of at least $H+1$ arcs and thus violate the hop constraint H.

$$\text{subject to} \quad \sum_{u|(u,v)\in A^+} x_{u,v} = 1 \qquad \forall\, v \in V \tag{6.2}$$

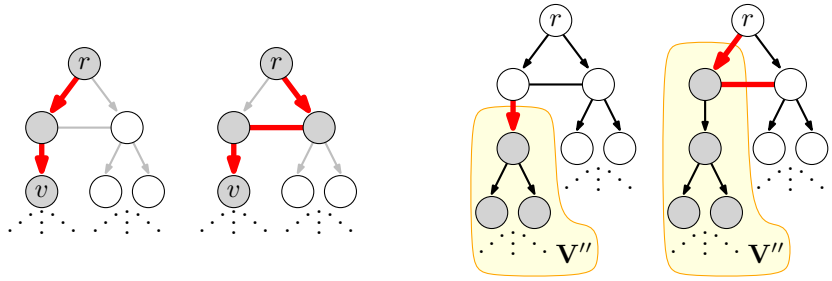
$$\sum_{v\in V} x_{r,v} = 1 \tag{6.3}$$

$$\sum_{(u,v)\in\delta^+(V')} x_{u,v} \geq 1 \qquad \forall\, V' \subset V^+ \mid r \in V' \tag{6.4}$$

$$\sum_{(u,v)\in J(P)} x_{u,v} \geq 1 \qquad \forall\, P \in P(V^+) \mid r \in S_0. \tag{6.5}$$

The objective is to minimize the total costs of all selected arcs (6.1). All nodes of the original graph (without artificial root node r) have exactly one predecessor (6.2), and just one node is successor of r (6.3). To achieve a connected, cycle free solution we include the widely used directed connection cuts (6.4), where $\delta^+(V')$ denotes all arcs (u,v) with $u \in V'$ and $v \in V^+ \setminus V'$, see also [94].

The diameter restriction is enforced by the jump inequalities (6.5) from [29] as follows. Consider a partitioning P of V^+ into $H+2$ pairwise disjoint nonempty sets S_0 to S_{H+1} with $S_0 = \{r\}$. Let $\sigma(v)$ denote the index of the partition a node v is assigned to. Jump $J(P)$ is defined as the set of arcs $(u,v) \in A^+$ with $\sigma(u) < \sigma(v) - 1$, i.e., $J(P)$ contains all arcs leading from a partition to a higher indexed one and skipping at least one in-between, see Fig. 6.1. The jump inequality associated with this partitioning states that in a feasible HCMST T^+ at least one of these arcs in $J(P)$ must appear. Otherwise, there would be a path connecting the root contained in S_0 to a node in S_{H+1} with length at least $H+1$ violating the hop constraint. Such jump inequalities must hold for all possible partitionings $P(V^+)$ of V^+ with r being element of set S_0.



(a) Two different paths from r to v. (b) Center connection inequalities.

Figure 6.2: Triangle tree: In the odd diameter case there are two paths connecting r with any node $v \in V$. This leads to the center connection inequalities involving the center edge.

The odd diameter case additionally makes use of the center edge variables $z_{u,v}$:

$$\text{minimize} \quad \sum_{(u,v)\in A} c_{u,v} \cdot x_{u,v} + \sum_{(u,v)\in E} c_{u,v} \cdot z_{u,v} \tag{6.6}$$

$$\text{subject to} \quad \sum_{v\in V} x_{r,v} = 2 \tag{6.7}$$

$$\sum_{v|(u,v)\in E} z_{u,v} = x_{r,u} \qquad \forall\, u \in V \tag{6.8}$$

$$2 \cdot \sum_{(u,v)\in\delta^+(V\setminus V'')} x_{u,v} + \sum_{v\in V''} x_{r,v} + \sum_{(u,v)\in\delta(V'')} z_{u,v} \geq 2 \qquad \forall\, \emptyset \neq V'' \subset V \tag{6.9}$$

(6.2), (6.4), and (6.5) are adopted unchanged.

Now, two nodes are to be connected to the artificial root node r (6.7), and they are interlinked via the center edge (6.8). The costs of this edge are also accounted for in the extended objective function (6.6).

The new connection inequalities (6.9), which we call *center connection inequalities*, are not necessary for the validity of the model but strengthen it considerably. They are essentially derived from observations in [60]: The HCMST T^+ together with the center edge linking the two center nodes connected to r forms a special structure, a so-called *triangle tree*. In such a tree every node $v \in V$ can be reached from r by two different – not necessarily completely arc disjoint – directed paths: The first path directly connects r with v via one center node, whereas the second one visits the second center node first and crosses the center edge, see Fig. 6.2. This idea is

captured in these inequalities: Two paths from r have to reach each subset V'' of nodes of V, either from other non-center nodes (first term) or – in case a center node v is contained in V'' – directly from r and via the center edge (second and third terms).

As there are exponentially many directed and center connection inequalities (6.4, 6.9) and jump inequalities (6.5), directly solving these models is not a practical option. Instead, we start without these inequalities and apply Branch&Cut, thus, separating inequalities that are violated by optimal LP solutions on the fly. Directed connection cuts – including our special variants (6.9) – can efficiently be separated: In each LP solution $|V|$ max-flow/min-cut computations have to be performed between the artificial root r and any node of the instance graph. To compute these maximum flows in a directed graph we used the algorithm by Cherkassky and Goldberg [24]. Unfortunately, solving the separation problem for the jump inequalities is conjectured to be NP-hard [29].

6.3 Jump Cut Separation

In order to find a valid jump cut, we have to identify a node partitioning P and corresponding jump $J(P)$ for which the current LP solution $(x^{\mathrm{LP}}, z^{\mathrm{LP}})$ violates $\sum_{(u,v)\in J(P)} x^{\mathrm{LP}}_{u,v} \geq 1$.

6.3.1 Exact Separation Model

In a first attempt we formulate the separation problem as an ILP, making use of the following variables: $y_{v,i} \in \{0,1\}$, $\forall v \in V^+,\ i = 0,\ldots,H+1$, is set to one iff node v is assigned to partition S_i, and $x_{u,v} \in \{0,1\}$, $\forall (u,v) \in A^{\mathrm{LP}}$ is set to one iff arc (u,v) is contained in the jump $J(P)$; let $A^{\mathrm{LP}} = \{(u,v) \in A^+ \mid x^{\mathrm{LP}}_{u,v} > 0\}$. This leads to the following model:

$$\text{minimize} \quad \sum_{(u,v)\in A^{\mathrm{LP}}} x^{\mathrm{LP}}_{u,v} \cdot x_{u,v} \tag{6.10}$$

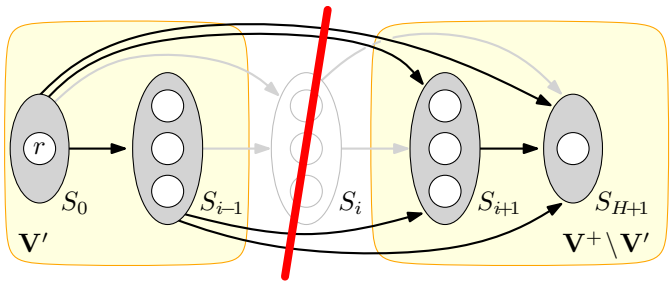


Figure 6.3: A partitioning P with $\sum_{J(P)} x^{\mathrm{LP}} < 1$ and an empty set S_i corresponds to a violated directed connection cut.

$$\text{subject to} \quad \sum_{i=1}^{H+1} y_{v,i} = 1 \qquad \forall\, v \in V \tag{6.11}$$

$$y_{r,0} = 1 \tag{6.12}$$

$$\sum_{v \in V} y_{v,H+1} = 1 \tag{6.13}$$

$$y_{u,i} - 1 + \sum_{j=i+2}^{H+1} y_{v,j} \leq x_{u,v} \qquad \forall\, i \in \{1, \ldots, H-1\},\ (u,v) \in A^{\mathrm{LP}} \tag{6.14}$$

$$\sum_{i=2}^{H+1} y_{v,i} \leq x_{r,v} \qquad \forall v \in V \mid (r,v) \in A^{\mathrm{LP}} \tag{6.15}$$

The objective is to minimize the total weight of the arcs in the jump $J(P)$ (6.10). Each node in V is assigned to exactly one of the sets S_1 to S_{H+1} (6.11), whereas the artificial root r is the only node in set S_0 (6.12). Exactly one node is assigned to set S_{H+1} (6.13), as Dahl et al. [29] showed that a jump inequality is facet-defining iff the last set is singleton. Finally, an arc (u,v) (6.14), respectively (r,v) (6.15), is part of the jump $J(P)$ iff it leads from a set S_i to a set S_j with $j \geq i+2$.

Note that, according to the following theorem, it is not necessary to explicitly address the condition that no partition may be empty:

Theorem 6 *In case all directed connection cuts are separated in advance, no partition S_i, $i \in \{1, \ldots, H\}$, will be empty in an optimal solution to the ILP model described by (6.10) to (6.15).*

Proof Assume S_i, $i \in \{1, \ldots, H\}$, is an empty set in an otherwise valid (according to the rules defined for jump inequalities) partitioning P, $\sum_{(u,v) \in J(P)} x^{\mathrm{LP}}_{u,v} < 1$. Then V^+ can be partitioned into two sets V' and $V^+ \setminus V'$, with $V' = \{v \in V^+ \mid \sigma(v) < i\}$

(including r). The sets V' and $V^+ \setminus V'$ define a cut where all arcs from V' to $V^+ \setminus V'$ belong to the jump $J(P)$; it follows that $\sum_{(u,v)\in\delta^+(V')} x^{\mathrm{LP}}_{u,v} < 1$. Consequently, every partitioning with $\sum_{(u,v)\in J(P)} x^{\mathrm{LP}}_{u,v} < 1$ and an empty set S_i, $i \in \{1,\ldots,H\}$, can be transformed into a violated directed connection inequality, see Fig. 6.3. Since such a violated directed connection inequality does not exist in the current LP solution by assumption, no set S_i can be empty. □

This observation reveals the possibility to avoid time-consuming max-flow/ min-cut computations to separate directed connection cuts. By not forcing the sets $S_1,\ldots,S_H$ to be nonempty, violated directed connection and jump constraints can be identified by only one single separation procedure, depending on whether the node partitioning P contains an empty partition S_i or not.

The exact jump cut separation model contains $O(H \cdot |V| + |A^{\mathrm{LP}}|)$ variables and $O(|V| + H \cdot |A^{\mathrm{LP}}|)$ constraints. Solving it by a general purpose solver each time when a jump cut should be separated is, however, only applicable for small problem instances as the computation times are high and increase dramatically with the problem size. According to our experiments, between about 85% and almost 100% of the total time for solving the BDMST problem is spent in this exact separation procedure for jump cuts.

To speed up computation, we developed heuristic procedures for this separation problem and apply them in a hierarchical fashion: Two alternative construction heuristics are used to find initial partitionings; they are improved by local search and – in case a violated jump inequality has not yet been encountered – finally by tabu search.

6.3.2 Simple Construction Heuristic C^A

Heuristic C^A greedily assigns the nodes V^+ to sets $S_1,\ldots,S_{H+1}$ trying to keep the *number* of arcs that become part of the jump $J(P)$ as small as possible, see Algorithm 11. An independent partitioning is computed for each node $v \in V$ initially placed in the last set S_{H+1}, and the overall best solution is returned. To derive one such partitioning, all nodes u connected to r via an arc $(r,u) \in A^{\mathrm{LP}}$ with $x^{\mathrm{LP}}_{r,u}$ exceeding a certain threshold (0.5 in our experiments) are assigned to set S_1. Then the algorithm iterates through partitions S_{H+1} down to S_3. For each of these sets S_i all arcs $(w,u) \in A^{\mathrm{LP}}$ with target node $u \in S_i$ are further examined. In case w is still *free* (i.e., not already assigned to a set), it is placed in S_{i-1}, in order to avoid (w,u) becoming part of $J(P)$. At the end, eventually remaining free nodes are assigned to set S_1.

Results achieved with heuristic C^A were encouraging, but also left room for improvement when compared to the exact separation. In particular, this heuristic does (almost) not consider differences in arc weights $x^{\mathrm{LP}}_{u,v}$ when deciding upon the assignment of nodes.

6.3.3 Constraint Graph Based Construction Heuristic C^B

To exploit arc weights in a better way, we developed the more sophisticated construction heuristic C^B which makes use of an additional *constraint graph* $G_C = (V^+, A_C)$. To avoid that an arc $(u, v) \in A^{\mathrm{LP}}$ becomes part of $J(P)$, the constraint $\sigma(u) \geq \sigma(v) - 1$ must hold in partitioning P. Heuristic C^B iterates through all arcs in A^{LP} in decreasing LP-value order (ties are broken arbitrarily) and checks for each arc whether or not its associated constraint on the partitioning can be realized, i.e., if it is compatible with previously accepted arcs and their induced constraints. Compatible arcs are accepted and collected within the constraint graph, while arcs raising contradictions w.r.t. previously accepted arcs in G_C are rejected and will be part of $J(P)$. After checking each arc in this way, a partitioning P respecting all constraints represented by G_C is derived. Algorithm 12 shows this heuristic in pseudo-code.

Algorithm 11: Simple Construction Heuristic C^A

input : V^+, A^{LP}
output: partitioning P of V^+

forall *nodes* $v \in V$ **do**
  $S_0 \leftarrow \{r\}$; $S_{H+1} \leftarrow \{v\}$; $\forall i = 1, \ldots, H: \ S_i \leftarrow \emptyset$
  forall *arcs* $(r, u) \mid u \neq v$ **do**
    if $x^{\mathrm{LP}}_{r,u} > 0.5$ **then** $S_1 \leftarrow S_1 \cup \{u\}$
  for $i = H + 1, \ldots, 3$ **do**
    foreach *node* $u \in S_i$ **do**
      foreach *arc* $(w, u) \in A^{\mathrm{LP}} \mid w$ *not already assigned* **do**
        $S_{i-1} \leftarrow S_{i-1} \cup \{w\}$
  forall *still unassigned nodes* $u \in V^+$ **do**
    $S_1 \leftarrow S_1 \cup \{u\}$
  derive jump $J(P)$ for current partitioning $P = (S_0, \ldots, S_{H+1})$
  evaluate $J(P)$ and store P if best so far
return best found partitioning

In more detail, graph G_C not only holds compatible arcs but for each node $u \in V^+$ also an integer *assignment interval* $b_u = [\alpha_u, \beta_u]$ indicating the feasible range of partitions; i.e., u may be assigned to one of the sets $\{S_i \mid i = \alpha_u, \ldots, \beta_u\}$. When an arc (u, v) is inserted into A_C, the implied new constraint $\sigma(u) \geq \sigma(v) - 1$ makes the following interval updates necessary:

$$b_u \leftarrow [\max(\alpha_u, \alpha_v - 1),\ \beta_u] \quad \text{and} \quad b_v \leftarrow [\alpha_v,\ \min(\beta_v, \beta_u + 1)]. \tag{6.16}$$

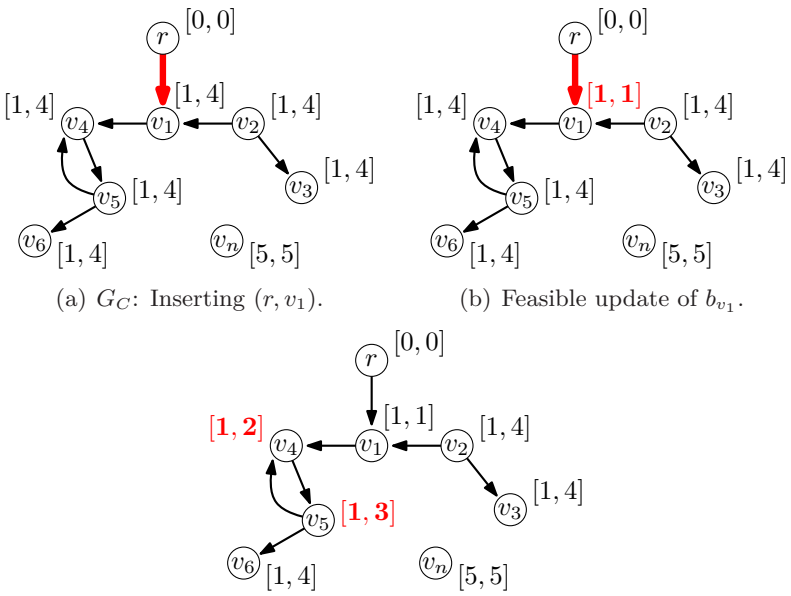
Changes of interval bounds must further be propagated through the constraint graph by recursively following adjacent arcs until all bounds are feasible again w.r.t. the constraints.

Figure 6.4 gives an example of such an update procedure after inserting an arc into the constraint graph. It visualizes the relevant part of G_C in an instance with a diameter constraint of six, including the artificial root node r assigned to S_0 ($b_r = [0, 0]$), node v_n in partition S_{H+1} ($b_{v_n} = [5, 5]$), six additional nodes v_1 to v_6 which still are allowed to be assigned to any partition S_i, $i = 1, \ldots, 4$, and already some compatible arcs. In Fig. 6.4(a) a new arc from r to v_1 should be inserted into the constraint graph. To prevent this arc to become part of the jump $J(P)$ we have to restrict the assignment interval of v_1 (r is already fixed to a single partition): If v_1

Algorithm 12: Constraint Graph Based Construction Heuristic C^B

input : V^+, A^{LP}
output: partitioning P of V^+

```
1  sort A^LP according to decreasing LP values
2  forall nodes v ∈ V do
3      S_0 ← {r}; S_{H+1} ← {v}; ∀i = 1,...,H : S_i ← ∅
4      b_r = [0,0]; b_v = [H+1, H+1]; ∀w ∈ V \ {v}: b_w ← [1, H]
5      initialize G_C: A_C ← ∅
6      initialize jump J(P) ← ∅
7      forall arcs (u,v) ∈ A^LP according to decreasing x^LP_{u,v} do
8          if A_C ∪ (u,v) allows for a feasible assignment of all nodes then
9              A_C ← A_C ∪ (u,v)
10             perform recursive update of bounds starting at b_u and b_v
11         else
12             J(P) ← J(P) ∪ (u,v)
13     assign nodes to partitions according to the constraints in G_C
14     evaluate jump J(P) and store P if best so far
15 return best found partitioning
```



(a) G_C: Inserting (r, v_1).

(b) Feasible update of b_{v_1}.

(c) Recursive update.

Figure 6.4: Insertion of arc (r, v_1) into the constraint graph G_C, including all necessary updates to the assignment intervals.

would be assigned to any partition S_i with $i \geq 2$, the arc (r, v_1) would skip at least S_1 making it a jump arc. Therefore, the upper bound β_{v_1} has to be decreased to one ($b_{v_1} = [1, \min(4, 0+1)]$), see Fig. 6.4(b). Now this update has to be propagated through the constraint graph as shown in Fig. 6.4(c). Nothing has to be done for node v_2 (and so for v_3), it still can be assigned to any of the partitions S_1 to S_4 since the arc (v_2, v_1) can no longer become part of $J(P)$ ($\sigma(v_2) \in [1, 4]$ will always be greater than or equal to $\sigma(v_1) - 1 = 1 - 1 = 0$). On the other hand, the upper interval bound of v_4 has to be set to two (to avoid that arc (v_1, v_4) skips at least partition S_2), and – analogously – β_{v_5} has to be set to three. After this recursive update procedure the constraint graph is in a valid state again, i.e., all nodes can be assigned to partitions without violating constraints implied by the collected arcs A_C.

An arc (u, v) can be feasibly added to the graph G_C without raising conflicts with any stored constraint as long as the assignment intervals b_u and b_v do not become empty, i.e., $\alpha_u \leq \beta_u \wedge \alpha_v \leq \beta_v$ must always hold. In Algorithm 12 this condition is tested in line 8, and the arc (u, v) is either accepted for A_C or added to $J(P)$, respectively.

Theorem 7 *The recursive update of the assignment interval bounds in G_C after inserting an arc (u, v) always terminates and cannot fail if it succeeded at nodes u and v.*

Proof Let G_C be valid, i.e., it contains no contradicting constraints, and it was possible to insert arc (u, v) into the graph without obtaining empty assignment intervals for nodes u and v. Let (s, t) be any other arc $\in G_C$, implying $\alpha_s \geq \alpha_t - 1$, and $\beta_t \leq \beta_s + 1$. Now, assume that α_t was updated, i.e., increased, to α'_t, with $\alpha'_t \leq \beta_t$. If the lower bound of s must be modified, it is set to $\alpha'_s = \alpha'_t - 1$ according to the update rules. To prove that the interval at s will not become empty we have to show that $\alpha'_s \leq \beta_s$:

$$\alpha'_s \overset{\text{(update rule)}}{=} \alpha'_t - 1 \overset{\alpha'_t \leq \beta_t}{\leq} \beta_t - 1 \overset{\beta_t \leq \beta_s + 1}{\leq} \beta_s \tag{6.17}$$

The feasibility of the upper bound propagation can be argued in an analogous way. This also proves that the recursive update procedure terminates, even when there are cycles in G_C (intervals cannot become empty, and updates increase respectively decrease lower and upper bounds by at least one). □

6.3.4 Local Search and Tabu Search

Although the construction heuristics usually find many violated jump inequalities, there is still room for improvement using local search. The neighborhood of a current partitioning P is in principle defined by moving one node to some other partition. As this neighborhood would be relatively large and costly to search, we restrict it as follows: Each arc $(u, v) \in J(P)$ induces two allowed moves to remove it from the associated jump $J(P)$: reassigning node u to set $S_{\sigma(v)-1}$ and reassigning node v to set $S_{\sigma(u)+1}$, respectively. Moves modifying S_0 or S_{H+1} are not allowed. The local search is performed in a first improvement manner until a local optimum is reached; see Algorithm 13.

In most cases, the construction heuristics followed by local search are able to identify a jump cut if one exists. In the remaining cases, we give tabu search a try to eventually detect still undiscovered violated jump inequalities. Algorithm 14 shows our tabu search procedure in pseudo-code.

The neighborhood structure as well as the valid moves are defined as in the local search, but now a best improvement strategy is applied. Having performed a movement of a node v, we file as tabu the node v in combination with its inverted direction of movement (to a lower or higher indexed set, respectively).

Algorithm 13: Local Search

input : V^+, A^{LP}, current partitioning P and implied jump $J(P)$
output: possibly improved partitioning P of V^+

```
repeat
    improved ← false
    forall arcs (u, v) ∈ J(P) do
        if moving u to S_{σ(v)−1} or v to S_{σ(u)+1} is valid and improves solution
        then
            perform move; update P and J(P) correspondingly
            improved ← true
            break
until improved = false
return partitioning P
```

Algorithm 14: Tabu Search

input : V^+, A^{LP}, current partitioning P and implied jump $J(P)$
output: possibly improved partitioning P of V^+

```
tabu list L ← ∅
repeat
    search neighborhood of P for best move m considering tabu list L
    perform move m; update P and J(P) correspondingly
    file move m^{-1} in tabu list: L ← L ∪ {m^{-1}}
    remove from L entries older than max(l_min, γ · |J(P)|) iterations
until no new best partitioning found during the last i_max iterations
return best encountered partitioning
```

The tabu tenure is dynamically controlled by the number of arcs in jump $J(P)$: Tabu entries older than $\max(l_{\min}, \gamma \cdot |J(P)|)$ iterations are discarded, where $l_{\min}$ and γ are strategy parameters.

We consider the following aspiration criterion: The tabu status of a move is ignored if the move leads to a new so far best node partitioning. Tabu search terminates when a predefined number $i_{\max}$ of iterations without improvement of the overall best partitioning is reached.

6.4 Primal Heuristics

In order to further improve the performance of our Branch&Cut approach we make use of additional fast heuristics to set an initial solution and to locally improve incumbent solutions.

To create good starting solutions the center based tree construction (CBTC) and the randomized tree construction (RTC) heuristics, proposed by Julstrom in [82], are utilized; for a detailed discussion of these construction heuristics see Section 3.2. While CBTC is used for BDMST instances with random edge costs, RTC is applied to Euclidean instances.

Both construction heuristics are designed to operate on complete graphs. Whereas CBTC can handle incomplete graphs easily we modified RTC to increase the possibility of identifying a valid BDMST also on sparse graphs in the following way: Every node of the permutation not feasibly connectable is stored within a queue. After the whole permutation of nodes has been processed each node in the queue is again checked if it could be connected to the tree without violating the height restriction. This procedure is stopped when either the queue becomes empty or none of the nodes in the queue can be added feasibly to the tree. In addition, in case the diameter is odd a permutation is only accepted if the first two nodes – which should form the center – are linked via an edge.

Solutions of both construction heuristics as well as all incumbent solutions found during the optimization are further improved by the variable neighborhood descent (VND) from [74] utilizing four of the neighborhood structures already discussed in Chapter 4: The arc exchange, the node swap, the level change, and the center exchange level neighborhood.

6.5 Computational Results

For our computational experiments we utilize Euclidean (TE) and random (TR) instances as described and used by Gouveia et al. [59, 60] as well as complete and sparse Euclidean instances of Santos et al. [43, 103]. The instance type, together with the number of nodes ($|V|$) and edges ($|E|$) and the diameter bound (D) is specified for each test case in the following results tables. All experiments have been performed on a dual-core AMD Opteron 2214 machine (2.2GHz), and CPLEX 11.1 has been used as ILP solver and framework for Branch&Cut. Since most of the heuristic components are not deterministic, the median and/or the mean value of at least 30 independent runs is listed for each experiment (when not otherwise

specified). To verify statistical significance Wilcoxon rank tests with an error level of 5% (if not indicated otherwise) have been performed.

The experiments were executed with modified jump cut heuristics to simultaneously identify violated directed connection cuts to avoid additional time-consuming max-flow/min-cut computations (see proof of Theorem 6). Although a polynomial time exact separation procedure is replaced by a heuristic approach, preliminary tests demonstrated a significant enhancement in running time. Violated directed connection cuts were only identified separately in case the exact ILP model was used to separate jump cuts.

Table 6.1 demonstrates the clear advantages of applying primal heuristics: For a set of small and medium-sized instances the running times in seconds are given (heuristic jump cut separation using construction heuristic C^B with local search), as well as the mean values (including the gaps to the optimal solutions) and the standard deviations of the initial solutions. For instances with random edge costs (TR) the CBTC construction heuristic was used to compute initial solutions, RTC for all others. Since CBTC gives deterministic results for a given center it was executed once for each node $\in V$ for even diameter bounds. Otherwise, both construction heuristics were iterated until no better solution could be found for 100 runs; the finally best solution was utilized as initial solution in Branch&Cut.

The results are clear: Primal heuristics boost the optimization noticeable, especially if D is even. Significantly better results are highlighted in gray, the error probability obtained by the Wilcoxon tests is always less than 0.01%, except for instance TR 60/600/7 (0.789%). The parts of the overall running times of CBTC/RTC and the VND to improve incumbent solutions are negligibly, much less than one second for all instances. Only in some rare cases the primal heuristics can mislead CPLEX, although the minimal running times achieved are still better or at least comparable.

The solutions computed by CBTC and RTC for these small instances are in general of high quality (average objective value less than 2% from the optimum) when the graph is complete or at least dense. On sparse graphs (Santos 40/100, TR 60/600) already finding a feasible solution is difficult. An interesting observation is that the running times are much more stable when no primal heuristics are used, so differences in the jump cuts identified by C^B plus local search have only a relatively small impact in this case. For all remaining experiments primal heuristics were activated.

For smaller instances where the exact ILP-based jump cut separation can also be applied, Table 6.2 lists success rates SR($\cdot$) for finding existing violated jump inequalities in LP solutions for the two construction heuristics (C^A and C^B), option-

Table 6.1: Optimization with and without primal heuristics, running times t (in seconds), and quality of solutions, compared to the optimum *(opt)*, obtained by the construction heuristics RTC (Euclidean instances TE and Santos) or CBTC (instances with random weights TR); significantly better results according to Wilcoxon tests are highlighted gray. Since not all of the applied heuristics are deterministic, 30 independent runs have been performed for each instance.

				t(primal heuristics)			t(no primal heuristics)			quality RTC/CBTC			
Instance	$\|V\|$	$\|E\|$	D	median	min	max	median	min	max	*opt*	mean	stddev	gap(mean)
TE	30	200	4	**11.78**	11.59	12.03	21.57	21.36	21.85	*599*	599.13	0.34	0.02%
			6	**8.92**	8.63	12.68	12.84	12.70	13.11	*482*	483.97	2.98	0.41%
			8	**1.99**	1.89	2.27	2.41	2.33	2.51	*437*	437.35	1.05	0.08%
TR	30	200	4	**1.37**	1.35	1.41	2.13	2.08	2.20	*234*	234.00	0.00	0.00%
			6	**0.61**	0.59	0.63	0.78	0.74	0.80	*157*	160.00	0.00	1.91%
			8	**0.12**	0.10	0.13	0.15	0.14	0.16	*135*	135.00	0.00	0.00%
Santos	25	300	4	**2.07**	2.02	2.12	4.06	3.98	4.12	*500*	500.00	0.00	0.00%
			6	**0.70**	0.66	0.93	1.07	1.05	1.11	*378*	378.55	1.15	0.15%
			10	**0.48**	0.40	0.56	0.59	0.55	0.62	*379*	383.06	2.13	1.07%
	40	100	4	**1.16**	1.10	1.29	1.34	1.27	1.38	*755*	759.26	11.45	0.56%
			6	**0.43**	0.40	0.45	**0.43**	0.41	0.44	*599*	621.32	2.87	3.73%
			10	**0.38**	0.36	0.41	0.39	0.37	0.41	*574*	589.42	5.58	2.69%
TE	40	400	4	**27.98**	27.18	46.24	91.98	91.23	93.60	*672*	674.32	3.35	0.35%
			6	**126.62**	93.23	243.96	182.59	181.73	189.06	*555*	558.97	1.96	0.71%
			8	**81.78**	42.37	98.84	154.92	154.01	162.29	*507*	514.94	3.05	1.57%
TR	60	600	4	**1739.10**	1647.47	1828.58	3494.98	3464.51	3645.16	*326*	368.00	0.00	12.88%
			6	**561.53**	537.10	607.79	901.11	894.57	937.41	*175*	179.00	0.00	2.29%
			8	**4.66**	4.53	4.89	4.74	4.67	4.89	*127*	148.00	0.00	16.54%
TE	30	200	5	67.50	45.67	69.34	**52.96**	52.54	53.74	*534*	534.29	0.90	0.05%
			7	28.98	24.91	31.95	**28.34**	27.92	28.91	*463*	464.68	1.58	0.36%
TR	30	200	5	2.67	2.36	3.64	**2.39**	2.35	2.44	*195*	196.52	3.11	0.78%
			7	**0.29**	0.27	0.34	0.32	0.31	0.33	*144*	145.26	3.20	0.87%
Santos	25	300	5	**10.42**	10.27	10.59	10.65	10.52	10.88	*429*	429.00	0.00	0.00%
			7	**2.13**	2.11	2.16	3.85	3.79	3.92	*408*	408.00	0.00	0.00%
			9	**1.11**	1.08	1.41	1.62	1.58	1.64	*336*	337.19	1.83	0.36%
	40	100	5	**0.93**	0.87	1.02	1.06	1.02	1.10	*729*	739.35	14.37	1.42%
			7	**3.38**	2.90	4.30	4.52	4.47	4.65	*667*	684.87	7.12	2.68%
			9	**3.44**	3.30	3.81	3.95	3.90	4.05	*552*	570.77	8.79	3.40%
TE	40	400	5	**348.51**	335.09	618.57	466.34	464.20	478.88	*612*	613.55	2.41	0.25%
			7	**463.89**	244.64	808.79	605.31	601.90	623.02	*527*	532.84	3.38	1.11%
			9	**181.40**	111.62	822.45	527.47	524.99	544.38	*495*	502.74	3.68	1.56%
TR	60	600	5	1286.76	652.53	2546.96	**811.16**	804.56	835.89	*256*	265.71	11.09	3.79%
			7	33.37	17.44	52.10	**27.31**	27.01	28.06	*150*	163.35	3.90	8.90%
			9	**5.99**	5.33	20.88	10.32	10.17	10.62	*124*	136.35	2.74	9.96%

ally followed by local search (L) and tabu search (T) with the strategy parameters $l_{\min} = 5$, $\gamma = 0.75$, and $i_{\max} = 25$. The number of cuts identified by the exact model is given in column "#exact". As can be seen, for even diameter already the simple construction heuristic C^A gives excellent results, in most cases further improved by

Table 6.2: Success rates SR (%) for separating jump cuts by construction heuristics C^A and C^B, optionally followed by local search L and tabu search T, in comparison to the exact separation approach on the same LP solutions.

Instance	$\|V\|$	$\|E\|$	D	#exact	SR(C^A)	SR(C^AL)	SR(C^B)	SR(C^BL)	SR(C^BLT)
TE	30	200	4	817	99.02%	**100.00%**	99.14%	99.39%	99.39%
			6	991	97.17%	**99.80%**	97.07%	97.58%	98.63%
			8	560	65.87%	92.94%	95.08%	95.42%	**96.35%**
TR	30	200	4	272	**100.00%**	**100.00%**	**100.00%**	**100.00%**	**100.00%**
			6	152	98.03%	**100.00%**	**100.00%**	**100.00%**	**100.00%**
			8	22	**100.00%**	**100.00%**	**100.00%**	**100.00%**	**100.00%**
Santos	25	300	4	316	**100.00%**	**100.00%**	**100.00%**	**100.00%**	**100.00%**
			6	126	99.21%	99.21%	**100.00%**	**100.00%**	**100.00%**
			10	77	**100.00%**	**100.00%**	**100.00%**	**100.00%**	**100.00%**
	40	100	4	204	**100.00%**	**100.00%**	**100.00%**	**100.00%**	**100.00%**
			6	112	**100.00%**	**100.00%**	**100.00%**	**100.00%**	**100.00%**
			10	85	64.71%	90.59%	**96.47%**	**96.47%**	**96.47%**
TE	30	200	5	2786	89.75%	**98.39%**	92.41%	95.36%	95.36%
			7	3353	64.04%	91.88%	94.06%	95.41%	**96.99%**
TR	30	200	5	377	79.05%	91.51%	96.55%	**97.35%**	**97.35%**
			7	89	80.90%	85.39%	92.13%	94.38%	**95.51%**
Santos	25	300	5	794	83.50%	97.10%	97.73%	98.36%	**99.46%**
			7	188	81.38%	88.83%	95.21%	95.74%	**96.81%**
			9	115	91.30%	93.91%	97.39%	97.39%	**98.26%**
	40	100	5	186	**100.00%**	**100.00%**	**100.00%**	**100.00%**	**100.00%**
			7	445	81.88%	93.82%	95.58%	96.15%	**96.16%**
			9	485	67.80%	73.35%	92.66%	93.04%	**94.02%**

local search. The statistically significantly better heuristic C^B (error level $< 0.01\%$) leaves not much room for local and tabu search to enhance the success rate. A more differentiated situation can be observed for odd diameter bounds. The number of jump cuts identified directly by C^B is significantly higher in contrast to C^A (error level $< 0.01\%$), whereas local search flattens the differences in the construction phase to a greater or lesser extent. On almost all test instances, tabu search further improves the success rate to more than 95%. In total, heuristic C^B followed by local search and tabu search was able to separate all existing jump cuts for 9 out of 22 instances.

The consequences of the success to reliably identify violated jump inequalities can be seen in Table 6.3, where for the various approaches CPU-times t$(\cdot)$ to identify proven optimal integer solutions are listed. It can clearly be seen that the excessive running times of the exact jump cut separation prohibit its usage on larger instances. Times of the overall optimization process are in general magnitudes higher as when using our heuristics for jump cut separation, sometimes even the given CPU-time limit of one hour is exceeded. Since tabu search is only executed in case the construction heuristic followed by local search fails to identify a violated jump inequality, running

Table 6.3: Optimal solution values, median running times t (in seconds) to find and prove these solutions when using different strategies for jump cut separation, and optimality gaps of the final LP relaxations in the root nodes of the Branch&Cut search trees when using heuristic C^B followed by local search and tabu search. The last column gives running times in case directed connection cuts (dc) are separated exactly using multiple max-flow/min-cut computations.

Instance	$\|V\|$	$\|E\|$	D	*opt*	t(exact)	t(C^AL)	t(C^BL)	t(C^BLT)	gap(C^BLT)	t(dc+C^BLT)
TE	30	200	4	*599*	3522.73	13.03	11.78	**11.39**	1.69%	18.73
			6	*482*	> 1h	32.06	**8.92**	9.09	2.59%	13.73
			8	*437*	> 1h	2.16	**1.99**	2.12	1.98%	3.25
TR	30	200	4	*234*	328.09	1.63	**1.37**	1.38	0.00%	3.28
			6	*157*	185.65	0.96	**0.61**	0.63	0.00%	1.16
			8	*135*	0.59	**0.11**	0.12	**0.11**	0.00%	0.30
Santos	25	300	4	*500*	809.86	7.03	**2.07**	2.10	0.00%	3.58
			6	*378*	215.30	1.04	**0.70**	0.71	0.53%	0.86
			10	*379*	419.03	0.58	**0.48**	**0.48**	0.00%	0.64
	40	100	4	*755*	105.34	**0.98**	1.16	1.18	0.00%	2.14
			6	*599*	41.07	**0.37**	0.43	0.43	0.00%	0.93
			10	*574*	440.55	**0.34**	0.38	0.36	0.13%	0.70
TE	30	200	5	*534*	> 1h	**57.85**	67.50	62.14	7.20%	148.88
			7	*463*	> 1h	28.87	28.98	**28.35**	6.63%	38.16
TR	30	200	5	*195*	831.31	2.86	**2.67**	2.85	9.40%	5.36
			7	*144*	139.08	**0.27**	0.29	0.30	4.56%	1.31
Santos	25	300	5	*429*	1122.52	7.20	10.42	**6.08**	8.87%	20.08
			7	*408*	2489.67	**1.69**	2.13	1.98	4.65%	6.10
			9	*336*	66.66	**1.01**	1.11	1.12	0.89%	1.28
	40	100	5	*729*	238.24	**0.79**	0.93	1.02	0.00%	2.98
			7	*667*	988.36	**2.47**	3.38	3.22	1.50%	5.32
			9	*552*	> 1h	7.47	**3.44**	3.98	3.22%	5.70

times of C^BL and C^BLT considerably differ only on few instances, especially when D is odd.

On these relatively small instances it is difficult to draw conclusions on the performance of the various heuristics, even though the time required to solve all instances to proven optimality is lowest for C^B with local search and tabu search (141.02s), followed by C^BL (150.86s) and C^AL (170.77s). The picture becomes more apparent when investigating slightly larger instances (sparse, dense, and complete graphs), see Table 6.4. Again, statistically significantly better results are highlighted gray; the error probability is always less than 0.01% except for instances TE 30/435/9 (0.5%), TR 40/480/7 (2.73%; C^AL is significantly faster although median(C^BL)<median(C^AL)), TR 40/480/9 (4.17%), and TR 40/780/7 (1.72%). With increasing instance size the higher success rates of C^BL in identifying jump cuts show a considerable impact on running times.

Table 6.4: Running times t (in seconds) on larger instances (sparse, dense, complete) when separating jump cuts using heuristics C^A and C^B including local search; statistically significantly better results are highlighted gray.

Instance		$\|V\|$	$\|E\|$	D	t(C^AL)	t(C^BL)	D	t(C^AL)	t(C^BL)
sparse	TE	30	175	4	9.40	**9.31**	5	112.39	**72.05**
				6	28.66	**6.62**	7	**23.07**	28.65
				8	2.09	**1.62**	9	**1.49**	**1.49**
dense			305	4	98.95	**27.08**	5	35.38	**33.51**
				6	24.01	**11.28**	7	**12.09**	27.10
				8	2.70	**2.01**	9	**1.47**	1.80
complete			435	4	98.68	**30.74**	5	54.49	**32.64**
				6	47.57	**13.18**	7	**13.00**	19.73
				8	2.68	**2.60**	9	**2.37**	2.64
sparse	TR	40	175	4	63.59	**24.27**	5	174.60	**20.03**
				6	10.28	**2.08**	7	3.82	**1.63**
				8	**0.46**	0.47	9	0.84	**0.72**
dense			480	4	173.81	**27.55**	5	24.63	**20.78**
				6	8.34	**2.71**	7	3.21	**3.09**
				8	0.77	**0.72**	9	1.15	**1.10**
complete			780	4	206.48	**27.75**	5	100.00	**68.67**
				6	7.60	**3.61**	7	**15.27**	15.50
				8	**1.08**	1.10	9	9.13	**8.96**
				sum:	787.15	**194.70**		588.40	**360.09**

To achieve a good runtime behavior using tabu search a lot of parameter tuning for $l_{\min}$, γ, and $i_{\max}$ is necessary. A parameter set working for all instance types and sizes very well does not exist. In addition, when the number of nodes and edges in the graph increases, the benefit of identifying more violated jump inequalities is increasingly undone. Especially this is true when D is odd since a lot of computational effort is invested into LP solutions in which no jump cuts exist. Therefore, we abstained from using tabu search on larger instances since the performance of the construction heuristics with local search is already excellent.

Table 6.3 also lists optimal solution values ("opt") as well as optimality gaps of the LP relaxations at the root nodes of the Branch&Cut search trees for C^BLT. Whereas our model is quite tight in the even diameter case, the gaps for odd diameters reveal potential for further investigations to strengthen the formulation. In the last column, Table 6.3 finally gives running times for C^BLT when directed connection cuts (dc) are separated for LP solutions before jump cuts using an exact max-flow/min-cut algorithm, which proved to be definitely much more time consuming by a factor of at least 1.2 up to 4 and more.

Last but not least, Table 6.5 compares our approach to the so far leading hop-indexed multi-commodity flow formulations from [59] (even diameter cases) and [60] (odd diameter cases) on larger instances. The columns list for each instance the optimal

Table 6.5: Optimal values resp. upper bounds, LP relaxation values, LP gaps (for C^BL and GMR, the tightest models from [59] and [60]), and running times on Euclidean and random instances with 40, 60, and 80 nodes.

								t(C^BL)		
Instance	$\|V\|$	$\|E\|$	D	*opt*/UB*	LP(C^BL)	gap(C^BL)	gap(GMR)	median	min	max
TE	40	400	4	*672*	672.00	0.00%	0.04%	27.98	27.18	46.24
			6	*555*	544.33	1.92%	0.60%	126.62	93.23	243.96
			8	*507*	500.14	1.35%	0.50%	81.78	42.37	98.84
	60	600	4	*1180*	1178.50	0.13%	0.10%	1062.03	673.11	1154.82
			6	*837*	816.85	2.41%	0.50%	9244.26	5331.65	16389.33
			8	**755**	736.60	2.44%		18844.98	15815.31	25913.07
	80	800	4	**infeasible**		infeasible		1871.81	1857.74	2098.96
			6	**1066**	1044.87	1.98%			> 10h	
			8	963*	925.32	*3.91%			> 10h	
TR	40	400	4	*309*	309.00	0.00%	0.00%	23.35	22.84	23.99
			6	*189*	189.00	0.00%	0.00%	2.82	2.78	2.90
			8	*161*	161.00	0.00%	0.00%	0.76	0.72	0.79
	60	600	4	*326*	323.49	0.77%	0.70%	1739.10	1647.47	1828.58
			6	*175*	171.16	2.19%	1.30%	561.53	537.10	607.79
			8	*127*	127.00	0.00%	0.00%	4.66	4.53	4.89
	80	800	4	*424*	399.67	5.74%	5.70%		> 10h	
			6	**210**	206.41	1.71%		1904.19	1891.74	2181.73
			8	**166**	164.33	1.00%		25.56	24.83	27.24
TE	40	400	5	*612*	578.42	5.49%	0.00%	348.51	335.09	618.57
			7	*527*	495.09	6.06%	0.30%	463.89	244.64	808.79
			9	*495*	468.08	5.44%	0.30%	181.40	111.62	822.45
	60	600	5	*965*	899.79	6.76%	0.00%	34288.91	31383.42	> 10h
			7	*789*	742.23	5.93%	0.00%		> 10h	
			9	*738*	690.88	6.38%	0.50%	> 10h	30869.08	> 10h
	80	800	5	**1313**	1205.82	8.16%			> 10h	
			7	**1010**	942.60	6.67%			> 10h	
			9	950*	871.90	*8.22%			> 10h	
TR	40	400	5	*253*	224.90	11.11%	1.00%	17.94	17.66	22.49
			7	*171*	169.11	1.10%	0.00%	2.16	2.00	2.26
			9	*154*	154.00	0.00%	0.00%	1.06	0.86	1.20
	60	600	5	*256*	217.14	15.18%	3.20%	1286.76	652.53	2546.96
			7	*150*	138.50	7.67%	0.30%	33.37	17.44	52.10
			9	*124*	119.84	3.35%	0.00%	5.99	5.33	20.88
	80	800	5	**323**	272.42	15.66%			> 10h	
			7	**185**	176.44	4.62%		153.57	126.16	300.28
			9	**158**	154.57	2.17%		15.97	13.81	133.14

objective value if known, otherwise an upper bound (opt/UB*), the LP relaxation value for construction heuristic C^B with local search (LP(C^BL)), the gaps for this approach and for the best model from [59] and [60] whenever the optimum is available resp. the corresponding values were published (gap(C^BL), gap(GMR)), as well as the running time to proven optimality (t(C^BL)); a time limit of 10 hours was used for these experiments.

We were able to discover and prove previously unknown optima (bold) and could show that instance TE 80/800/4 is infeasible. Concerning the LP gaps, the results are comparable on even diameter instances, while for odd diameters the flow models are significantly better. A fair runtime comparison to [59] and [60] is not possible since the used hardware is too different (dual-core AMD Opteron 2214 (2.2GHz) compared to an Intel Pentium II (450MHz)). A rough estimation indicates that the flow formulations have their strengths on small diameter bounds (4 to 6), whereas Branch&Cut dominates when the diameter bound is looser (6 and above). To give an example: In [60] Gouveia et al. report for their best odd diameter formulation, the Longest-Path model, on instance TE 40/400/5 a running time of 345 seconds to prove optimality, the Branch&Cut approach requires about the same time on a much faster machine (median: 348.51 seconds). On the same instance with a diameter bound of 9 the situation changes, Gouveia et al. list 44600 seconds for their model whereas Branch&Cut in general only requires about 181.40 seconds (median).

6.6 Conclusions and Future Work

In this work we presented a new ILP formulation for the BDMST problem utilizing jump inequalities to ensure the diameter constraint and solve it with Branch&Cut. The odd diameter case is further strengthened by new center connection inequalities. For the separation of jump inequalities we considered an exact ILP approach and two greedy construction heuristics followed by local and tabu search. While our exact separation prohibits its use in practice due to its excessive computation times, the heuristic methods are substantially faster and achieve convincing success rates in identifying violated jump inequalities; they lead to an excellent overall performance of the Branch&Cut.

The usage of primal heuristics for determining initial solutions and for locally improving new incumbent solutions enhances our approach significantly. The gain received by replacing an exact polynomial time separation procedure for directed connection cuts by fast (meta-)heuristics was surprisingly high and can be an interesting field for further research also for other types of cuts and problems. Having an exact algorithm at hand to solve BDMST instances of moderate size in reasonable time also opens up new opportunities in combining it with leading metaheuristics. Smaller subproblems arising can now be solved to proven optimality, or specially designed neighborhoods can be searched making use of the Branch&Cut approach.

Chapter 7

Metaheuristics for the BDMST Problem

7.1 Introduction

The applicability of exact approaches for the BDMST problem is limited to relatively small instances, namely considerable less than 100 nodes when dealing with complete graphs, cf. Chapters 5 and 6. To solve instances with up to 1000 nodes and more fast construction heuristics can be used, see Section 3.2 and Chapter 8, or metaheuristics in order to obtain better results at the expense of more computational effort. Different *evolutionary algorithms* (EAs) have been developed for this problem, mainly differing in their solution representation: an EA employing a direct edge-set encoding [116], a permutation-coded EA [83], or an EA of similar performance based on random-keys [81].

This chapter now describes a general *variable neighborhood search* (VNS) approach [75] for solving large instances heuristically. Furthermore, an EA based on a newly developed level encoding and an *ant colony optimization* (ACO) approach are proposed. All these metaheuristics make use of the neighborhood structures defined in Chapter 4 to locally improve candidate solutions. In comparison to previous work, the VNS as well as the EA and the ACO were able to obtain new, significantly better solutions on all investigated instances. In particular in long-term runs on larger instances, the ACO performs best with respect to solution quality, while the EA's results are better when the running time is strictly limited.

7.2 Variable Neighborhood Search

The framework follows the general VNS scheme as proposed by Hansen and Mladenović in [75] using *variable neighborhood decent* (VND) as local search strategy.

An initial solution is created by one of the fast greedy construction heuristics described in Section 3.2. In our implementation we repeatedly applied RTC until no new improved solution has been obtained within the last $n = |V|$ repetitions. Within VND we always use a best improvement strategy, i.e., each neighborhood is completely explored and the best move is performed as long as it yields an improvement. The following order of neighborhoods has proven to be successful: First, whole sub-trees are moved within the solution (arc exchange), afterwards the arrangement of nodes and their direct successors is considered (node swap). Then the usually more time consuming level based neighborhoods are applied: The best center with respect to the center exchange level neighborhood is determined, and finally the levels the non-center nodes are assigned to are refined by means of the level change neighborhood.

When performing a local search using a single neighborhood and following a best improvement strategy, it is sometimes possible to store information during the exploration of this neighborhood allowing a faster incremental search for the successive best move. We implemented such a scheme for the node swap and the level change neighborhoods, cf. the detailed description in Chapter 4. To benefit from this advantage and in contrast to standard VND, we do not switch back to the first neighborhood immediately after an improvement, but continue the local search within the same neighborhood until a local optimum is reached. Only then we restart our search with the first neighborhood in order to exploit further possible improvements.

Our general VNS framework is shown in Algorithm 15. Since VND always yields a solution that is locally optimal with respect to all used neighborhoods, it makes usually no sense to shake a solution in the VNS performing only a single random move in one of these neighborhoods. Therefore, shaking is performed by applying k random moves within one of the four neighborhoods chosen at random, with k running from $k_{\text{start}} \geq 2$ to $k_{\max}$.

In case the center exchange level neighborhood has been chosen for the shaking process we make an exception and use a combination of it and the level change neighborhood because iterated center exchange moves alone cannot gain the desired larger variation: The first $1 + D \bmod 2$ (the number of center nodes) moves are executed within the center exchange level neighborhood, and for the remaining $k - (1 + D \bmod 2)$ shaking moves we switch to the level change neighborhood.

Algorithm 15: VNS for the BDMST problem

```
1  create initial solution using RTC heuristic
2  number of shaking moves k ← k_start
3  while termination condition not met do
4      perform VND with best improvement strategy: begin
5          neighborhood l ← 1
6          while l ≤ 4 and time limit not reached do
7              switch l do perform local search to a local optimum using
               neighborhood
8                  case 1 : arc exchange
9                  case 2 : node swap
10                 case 3 : center exchange level
11                 case 4 : level change
12             if solution improved and l ≠ 1 then l ← 1 else l ← l + 1
13     end
14     if best solution improved or k ≥ k_max then k ← k_start else k ← k + 1
15     choose neighborhood at random and shake currently best known solution
       using k moves
```

7.3 Evolutionary Algorithm

The evolutionary algorithm uses a novel representation, based on the concepts of the level neighborhoods, to loosely represent solutions to the BDMST problem. It then relies on a decoding mechanism to transform this representation into a specific tree. A BDMST is represented as a string of levels, i.e., each gene represents the level of a corresponding node in the directed tree. This level information on itself does not completely encode a tree, and therefore, we use Algorithm 10 (Section 4.3, page 55) to derive the predecessor information, i.e., a specific tree. This predecessor information is also stored in the individual and local improvement is applied.

The local improvement consists of applying only the arc exchange neighborhood search from Section 4.2.1. If an improvement can be made, which is the case more often than not, the node level representation and fitness value are set according to the new improvement. The arc exchange neighborhood is chosen to perform the local improvement as it is computationally relatively cheap in practice and it complements the genetic operators, which provide new points in the search space by changing level information, and especially, by moving the centers around.

We decided not to include the other three neighborhood searches. This decision is based on preliminary experiments where we found including them almost always extremely increased the necessary running times for converging to high-quality solutions. These experiments consisted of including the three neighborhood searches with equal probability to the arc exchange neighborhood search, including each of them with the small probability of 0.10, and including each of them with a very small probability of 0.01.

An important aspect of the representation is that an individual is allowed a limited number of center nodes only. These nodes have a level of zero. If the BDMST problem has an even diameter, one node is allowed, if it is odd, two nodes form the center. When generating the initial population, genes are assigned levels of one to the maximum height of the tree $H = \lfloor \frac{D}{2} \rfloor$. Thereafter, one or two nodes are randomly assigned to level zero, depending on the parity of the problem.

The genetic operators are of a simple, straight-forward nature, with the exception where they cater for the center nodes. A uniform crossover is performed where for each gene a level is chosen with equal probability from the two parents. The locations of the center nodes of both parents are put into a pool. Then one or two are selected randomly to represent the center nodes of the offspring. The remaining locations are checked in the offspring, if they contain center nodes inherited from the uniform crossover, then these genes are overwritten with a random level between one and the maximum tree height. The mutation operator generates a new level for each gene with a probability of $1/n$, skipping center nodes. Afterwards, the center nodes are swapped with other nodes selected randomly with equal probability.

A steady-state evolutionary model with binary tournament selection for choosing the parents is used. A new candidate solution always replaces the worst solution in the population, with one exception: duplicate individuals are not allowed to enter the population, whereas duplicate elimination is based on the genotype (level encoding) and not the phenotype (predecessor information of the BDMST). Crossover and mutation are always applied, as well as local improvement.

7.4 Ant Colony Optimization

In our ant colony optimization approach (ACO, Dorigo and Gambardella [40]) we also exploit the idea that a solution to the BDMST problem can be derived from an assignment of nodes to levels $0 \ldots H$ by Algorithm 10.

Therefore, we make use of a $n \times (H + 1)$ pheromone matrix τ where each value $\tau_{i,l}$ denotes the pheromone value for a node i at level l. The pheromone matrix

is uniformly initialized with $\tau_{i,l} = (n \cdot T_0)^{-1}$, with T_0 being the objective value of a heuristic solution to the BDMST problem computed using one of the presented greedy construction heuristics, e.g., RTC for Euclidean instances (see Section 3.2).

To derive a valid solution we have to restrict the number of center nodes, so they are chosen first: A node i is selected in proportion to its pheromone value $\tau_{i,0}$ at level 0, i.e., with a probability

$$P_{i,0} = \frac{\tau_{i.0}}{\sum_{j \in V} \tau_{j.0}}.$$

In case the diameter is odd, the second center node is selected analogously from all remaining nodes.

After the center has been determined all other nodes are assigned to the available levels $1 \dots H$ independently from each other. The probability for a node i to be set to level l is defined similar as for the center nodes, namely

$$P_{i,l} = \frac{\tau_{i.l}}{\sum_{l'=1}^{H} \tau_{i.l'}}.$$

Note in particular that these probabilities do not include any local heuristic component.

After each node has been assigned to a level, a corresponding BDMST is derived using Algorithm 10, where every node at a level ≥ 1 is connected to the cheapest available predecessor at any smaller level. Afterwards, this tree is locally improved using a VND exploiting only the arc exchange and node swap neighborhoods. The ideas of the level based neighborhoods are already captured in the construction and decoding phase. In practice, the use of these neighborhoods does not lead to a further improvement of the solution quality, but only significantly increases running time.

After each ant has built a BDMST the pheromone evaporation $\tau_{i.l} \leftarrow (1 - \rho) \cdot \tau_{i.l}$, where $\rho \in [0, 1)$ represents the pheromone decay coefficient, is triggered on all entries of the pheromone matrix. Only the best ant of an iteration is allowed to deposit pheromone: if node i is assigned to level l we set $\tau_{i.l} \leftarrow \tau_{i.l} + \rho \cdot \frac{1}{T^+}$ with T^+ being the objective value of the best BDMST in the current iteration.

7.5 Computational Results

The experiments have been performed on a Pentium®4 2.8 GHz system using Linux 2.4.21 as operating system utilizing benchmark instances already used in the corresponding literature from Beasley's OR-Library [17, 16] originally proposed for the

Table 7.1: Long-term runs on Euclidean instances; results for the EAs are taken from [81].

Instance			permutation coded EA			random-key coded EA			VNS					
$\|V\|$	D	nr.	best	mean	stddev	best	mean	stddev	k_{start}	$k_{\max}$	best	mean	stddev	sec.
100	10	00	7.818	7.919	0.07	7.831	7.919	0.05	3	15	**7.759**	**7.819**	0.03	37.35
		01	7.873	8.017	0.08	7.853	8.043	0.09			**7.852**	**7.891**	0.03	41.52
		02	7.990	8.139	0.08	7.982	8.137	0.09			**7.904**	**7.962**	0.04	38.66
		03	8.009	8.143	0.07	7.996	8.122	0.06			**7.979**	**8.046**	0.03	34.27
		04	8.193	8.335	0.08	8.198	8.313	0.08			**8.165**	**8.203**	0.03	39.31
250	15	00	12.440	12.602	0.08	12.448	12.580	0.08	4	20	**12.301**	**12.430**	0.05	1584.31
		01	12.237	12.432	0.10	12.222	12.393	0.10			**12.024**	**12.171**	0.06	1678.90
		02	12.117	12.282	0.08	12.178	12.315	0.07			**12.041**	**12.112**	0.04	1309.21
		03	12.572	12.824	0.11	12.632	12.802	0.07			**12.507**	**12.615**	0.06	1572.39
		04	12.358	12.608	0.12	12.382	12.623	0.10			**12.281**	**12.423**	0.07	1525.39
500	20	00	17.216	17.476	0.10	17.156	17.429	0.10	5	25	**16.974**	**17.129**	0.07	3718.54
		01	17.085	17.311	0.11	17.097	17.291	0.10			**16.879**	**17.052**	0.07	3762.02
		02	17.173	17.449	0.11	17.164	17.369	0.11			**16.975**	**17.148**	0.07	3849.42
		03	17.215	17.484	0.13	17.266	17.432	0.09			**16.992**	**17.166**	0.06	3687.97
		04	16.939	17.137	0.11	16.872	17.092	0.11			**16.572**	**16.786**	0.07	3693.13

Euclidean Steiner tree problem. These complete instances contain point coordinates in the unit square, and the Euclidean distances between each pair of points are taken as edge costs. For our tests we used the first five instances of each size $|V| = 100$, 250, 500, and 1000. The maximum diameters were set to 10, 15, 20, and 25, respectively. Best results over all algorithms are printed in bold.

First we compare the results of the VNS to those of the at this point leading evolutionary algorithms from [81] based on permutation and random-key representations. VNS uses a least-cost tree identified in multiple runs of RTC as initial solution: This construction heuristic is repeatedly performed until no better solution was obtained during the last n iterations. As stopping condition for VNS we used a combination of a CPU time limit (2000, 3000, and 4000 seconds for the 100, 250, and 500 node instances, respectively) and a maximum of 1000 consecutive applications of shaking without further improvement of the best solution. Depending on the problem size we also used different values k_{start} and k_{max} for shaking as indicated in Table 7.1.

Table 7.1 further lists for each instance the number of nodes, the maximum diameter D, the instance number, and for each approach the best found solution, the mean, and the standard deviation of 50 (EAs) respectively 30 (VNS) independent runs. In addition, for VNS the mean times to find the best solutions are given.

Table 7.2: Short-term runs with strict time limit on Euclidean instances.

Instance			time	edge-set coded EA			random-key coded EA			VNS			
$\|V\|$	D	nr.	limit (sec.)	best	mean	stddev	best	mean	stddev	$k_{\max}$	best	mean	stddev
500	20	00	50	19.368	19.830	0.17	21.223	21.440	0.07	25	**17.753**	**18.108**	0.12
		01		19.156	19.522	0.13	20.836	21.097	0.09		**17.688**	**17.966**	0.10
		02		19.321	19.888	0.16	21.042	21.304	0.11		**17.799**	**18.114**	0.10
		03		19.464	19.866	0.19	21.129	21.432	0.09		**17.930**	**18.161**	0.11
		04		19.209	19.477	0.17	20.728	21.017	0.11		**17.464**	**17.863**	0.12
500	20	00	500	18.470	18.976	0.13	19.658	19.908	0.14	25	**17.290**	**17.460**	0.08
		01		18.442	18.810	0.22	19.332	19.651	0.13		**17.215**	**17.373**	0.08
		02		18.619	19.056	0.18	19.618	19.887	0.10		**17.252**	**17.464**	0.05
		03		18.745	19.116	0.17	19.654	19.905	0.11		**17.318**	**17.514**	0.07
		04		18.197	18.685	0.20	19.312	19.635	0.10		**16.932**	**17.139**	0.09
1000	25	00	100	28.721	29.265	0.16	30.996	31.288	0.11	50	**25.850**	**26.188**	0.13
		01		28.607	29.105	0.19	30.832	31.132	0.11		**25.501**	**25.981**	0.17
		02		28.410	28.905	0.17	30.515	30.856	0.12		**25.340**	**25.705**	0.09
		03		28.695	29.263	0.21	30.966	31.277	0.08		**25.562**	**26.128**	0.17
		04		28.396	28.882	0.19	30.633	31.010	0.10		**25.504**	**25.826**	0.15
1000	25	00	1000	26.494	26.936	0.14	30.097	30.401	0.13	50	**25.177**	**25.572**	0.14
		01		26.300	26.789	0.24	29.924	30.261	0.12		**25.015**	**25.342**	0.14
		02		25.762	26.556	0.21	29.586	29.981	0.12		**24.816**	**25.086**	0.11
		03		26.470	26.816	0.15	29.946	30.329	0.13		**25.289**	**25.572**	0.11
		04		26.117	26.606	0.19	29.782	30.151	0.12		**25.026**	**25.254**	0.12

The results are clear and consistent among all instances: VNS outperforms both EAs with respect to the best found solutions as well as with respect to the mean values. Sometimes, especially on larger instances, even the mean over all VNS runs is better than the overall best solutions identified by both EAs. For VNS the time limit was of no significance for the 100 and 250 nodes instances, whereas on graphs with 500 nodes the optimization was usually terminated due to the time constraint before 1000 successive applications of shaking without further improvement were achieved.

As there was no time information published for the EAs in [81] and since we were particularly interested in the short-term performance, we did additional experiments providing the algorithms the same, very limited amount of time. For this comparison we chose the random-key coded EA from [81] and the edge-set EA by Raidl and Julstrom [116]; the latter because it scales much better to larger instances since it derives new candidate solutions in almost linear time. For these experiments, we used instances with 500 and 1000 nodes and two different time limits for each instance size, namely 50 and 500 seconds for the 500 node graphs and 100 and 1000 seconds for the instance with 1000 nodes, respectively. For VNS k_{start} was set to 5

Table 7.3: Final objective values of long-term runs on Euclidean instances.

Instance		VNS				level-encoded EA				ACO			
$\|V\|$	D nr	best	mean	stdev	sec.	best	mean	stdev	sec.	best	mean	stdev	sec.
100	1000	**7.759**	7.819	0.03	37.35	7.760	7.785	0.03	678.70	**7.759**	**7.768**	0.02	27.78
	01	7.852	7.891	0.03	41.52	**7.849**	**7.860**	0.02	734.65	7.850	7.864	0.01	25.10
	02	**7.904**	7.962	0.04	38.66	**7.904**	7.964	0.04	897.58	7.907	**7.943**	0.04	28.48
	03	7.979	8.046	0.03	34.27	**7.977**	8.008	0.03	732.83	7.979	**8.000**	0.01	38.24
	04	8.165	8.203	0.03	39.31	**8.164**	8.176	0.03	410.17	**8.164**	**8.170**	0.00	25.45
250	1500	12.301	12.430	0.05	1584.31	12.280	12.377	0.05	1992.70	**12.231**	**12.280**	0.02	174.17
	01	12.024	12.171	0.06	1678.90	12.054	12.156	0.06	1969.42	**12.016**	**12.038**	0.01	156.71
	02	12.041	12.112	0.04	1309.21	12.026	12.095	0.04	1897.87	**12.004**	**12.021**	0.01	145.29
	03	12.507	12.615	0.06	1572.39	12.487	12.594	0.05	1742.48	**12.462**	**12.486**	0.01	159.41
	04	12.281	12.423	0.07	1525.39	12.319	12.423	0.06	1712.16	**12.233**	**12.288**	0.04	211.11
500	2000	16.974	17.129	0.07	3718.54	16.866	16.967	0.06	2609.28	**16.778**	**16.850**	0.03	906.17
	01	16.879	17.052	0.07	3762.02	16.764	16.858	0.05	2472.59	**16.626**	**16.699**	0.03	1012.91
	02	16.975	17.148	0.07	3849.42	16.856	16.977	0.05	2808.15	**16.792**	**16.844**	0.03	1069.84
	03	16.992	17.166	0.06	3687.97	16.943	17.040	0.06	2837.81	**16.796**	**16.923**	0.04	1010.91
	04	16.572	16.786	0.07	3693.13	16.501	16.590	0.05	2294.43	**16.421**	**16.456**	0.02	947.26

and k_{max} depending on the instance size. We performed 30 runs for each instance and time limit.

Table 7.2 lists the results. As easily can be seen VNS again performs consistently better than both EAs. The mean values of VNS with the tighter time limits are even always superior to the objective values of the overall best solutions found by both EAs with 10 times more time available. Comparing the performance of the EAs the complexity of the chromosome decoding procedure in the random-key EA becomes noticeable, and the edge-set EA always gives better results since it can perform much more iterations.

Finally, the VNS is experimentally compared with the level-based EA and ACO for the BDMST problem. For the different approaches the following parameters were used: For the EA a population size of 100 was chosen. The number of artificial ants in the ACO was 25, the value for the pheromone decay coefficient ρ was studied in depth by extensive preliminary tests and was set in dependence on the size of the instance, namely $\rho = 0.003$, 0.005, 0.006, and 0.008 for the 100, 250, 500, and 1000 node instances. For the VNS the parameters were chosen as described above for the comparison with the edge-set and random-key coded EAs.

Regarding the termination condition we again performed two different series of experiments: long-term and short-term runs with the same CPU time restrictions as used in Tables 7.1 and 7.2, respectively. In addition, in case of the VNS and ACO

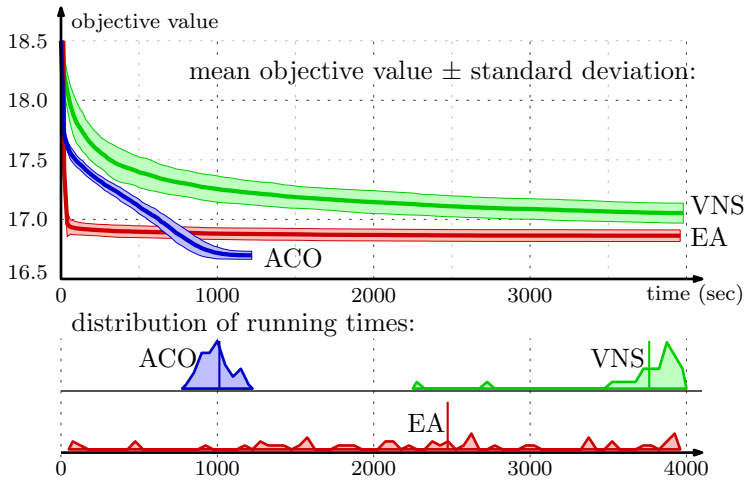


Figure 7.1: Objective value over time and running time distribution; long-term runs, $n = 500$, $D = 20$, instance 01.

a run was also terminated after 1000 iterations without further improvement of the best solution, since in this situation these two algorithms – in contrast to the EA – can be considered to have converged and further improvements are extremely unlikely. All statistical evaluations are based on 30 (VNS) respectively 50 (EA, ACO) independent runs for each instance.

Table 7.3 shows the results for long runs on instances with 100, 250, and 500 nodes, where the main focus lies on the quality of the built tree. Listed are best and mean objective values, the corresponding standard deviations and the average times to identify the finally best solutions for each instance and the three metaheuristics under consideration.

The instances with $n = 100$ seem too small to provide a proper comparison; each algorithm finds the best results for some of the instances. Furthermore, the objective values of the solutions over the three algorithms are rather similar and do not allow any conclusion to be drawn. On all larger instances with 250 and 500 nodes, the ACO clearly outperforms VNS and the EA. In fact, the ACO's observed mean objective values are never worse than the single best solutions identified by one of the other approaches. Furthermore, the ACO's standard deviations are smallest indicating a higher reliability of finding high-quality solutions.

Comparing VNS with the level-encoded EA on the 250 and 500 node instances, the mean values of the EA are always smaller than those of VNS with exception of the fifth instance (04) with 250 nodes, where they are equal.

Table 7.4: Final objective values of short-term runs on Euclidean instances.

Instance			limit	VNS				level-encoded EA				ACO			
n	D	nr	sec.	best	mean	stdev	sec.	best	mean	stdev	sec.	best	mean	stdev	sec.
500	20	00	50	17.753	18.108	0.12	46.41	**16.573**	**16.760**	0.16	37.94	17.594	17.751	0.06	41.29
		01		17.688	17.966	0.10	44.70	**16.826**	**17.014**	0.11	41.06	17.403	17.583	0.05	40.33
		02		17.799	18.114	0.10	46.23	**16.947**	**17.192**	0.13	43.15	17.653	17.756	0.05	39.66
		03		17.930	18.161	0.11	45.38	**16.957**	**17.085**	0.08	39.18	17.647	17.793	0.05	41.41
		04		17.464	17.863	0.12	45.94	**17.055**	**17.245**	0.13	39.54	17.331	17.438	0.05	40.95
500	20	00	500	17.290	17.460	0.08	476.22	**16.534**	**16.641**	0.07	340.34	17.017	17.150	0.07	485.57
		01		17.215	17.373	0.08	480.87	**16.808**	**16.902**	0.05	320.84	16.864	17.072	0.08	478.47
		02		17.252	17.464	0.05	476.33	**16.886**	**17.017**	0.06	319.09	17.094	17.259	0.07	479.17
		03		17.318	17.514	0.07	476.80	**16.923**	**17.036**	0.06	316.33	17.070	17.277	0.08	472.57
		04		16.932	17.139	0.09	473.82	17.007	17.105	0.06	288.66	**16.613**	**16.791**	0.08	479.93
1000	25	00	100	25.850	26.188	0.13	75.40	**24.831**	**25.019**	0.10	92.06	25.246	25.437	0.07	81.42
		01		25.501	25.981	0.17	68.30	**24.890**	**25.159**	0.10	89.29	25.092	25.239	0.07	80.17
		02		25.340	25.705	0.09	62.33	25.021	25.338	0.14	92.27	**24.870**	**25.007**	0.06	73.96
		03		25.562	26.128	0.17	73.89	**25.133**	25.524	0.12	92.17	25.329	**25.450**	0.06	76.56
		04		25.504	25.826	0.15	74.75	25.493	25.675	0.08	89.18	**24.884**	**25.153**	0.07	79.90
1000	25	00	1000	25.177	25.572	0.14	905.50	**23.434**	**23.573**	0.08	565.38	24.842	25.033	0.07	812.78
		01		25.015	25.342	0.14	930.04	**23.464**	**23.668**	0.08	561.49	24.634	24.834	0.06	847.79
		02		24.816	25.086	0.11	956.06	**23.635**	**23.793**	0.08	524.21	24.498	24.619	0.06	838.68
		03		25.289	25.572	0.11	928.97	**23.787**	**23.962**	0.09	602.30	24.993	25.091	0.06	793.41
		04		25.026	25.254	0.12	935.85	**23.837**	**23.982**	0.10	516.74	24.571	24.732	0.06	844.67

After verifying our data are normally distributed, we performed unpaired t-tests between each pair of algorithms for each problem instance. With a significance level of 1%, the difference in results between the EA and the VNS are all significant with the exceptions of instance 01 to 04 for $n = 250, D = 15$, and instance 02 for $n = 100, D = 10$. The differences between the ACO and the VNS are all significant, except for instance 02 for $n = 100, D = 10$. All differences between the EA and the ACO are significant, except for instances 01, 03, and 04 for $n = 100, D = 10$.

When looking at the average times until the best solutions have been found the ACO was in almost all cases substantially faster than VNS and the EA. Furthermore, on smaller instances the VNS found its final solutions in shorter time than the EA; on the largest considered instances the situation was vice versa.

Fig. 7.1 shows the mean objective value over time for multiple runs of the VNS, EA and ACO on instance number 01 with 500 nodes and a diameter of 20. The bottom of the chart displays the distributions of running times required to identify the best solution of a run, where mean running times are indicated by a vertical line each.

In our short-term experiments, we tested the approaches under CPU-time limits of 50 and 500 seconds for the 500 node instances, as well as 100 and 1000 seconds for

the 1000 node instances. Table 7.4 shows the results of these short-term runs. Here we see that roles are reversed, as in most cases the mean results of the EA are better than those of the ACO. Both, the EA and the ACO, almost always outperform the VNS. Interesting to note is that, with only a few exceptions, the mean results of the EA are already better than the best results found by the VNS, and this also holds true for the mean values of the ACO over the best of the VNS. Furthermore, when looking at the average computation times to identify the finally best solutions, the EA is usually faster than the ACO and VNS.

The objective value differences between all algorithms are statistically significant on an error-level of 1%, except for the EA and ACO on instances 02 and 04 for $n = 1000, D = 25$ with a time limit of 100 seconds.

In this comparison, we used for the short-term runs the same strategy parameters as for the high-quality experiments, which proved to be robust in excessive preliminary tests. However, there would be different possibilities to tune the algorithms to better perform when time is short. For example the VNS can omit the very time consuming center exchange level neighborhood; the idea of the level representation is still captured with the level change neighborhood. The population size of the EA can be reduced, as well as the number of ants in the ant colony. In addition, a higher pheromone decay coefficient ρ can be used to influence the convergence behavior of the ACO.

7.6 Conclusions

In this chapter new metaheuristic algorithms have been introduced for the Bounded Diameter Minimum Spanning tree problem, which make use of for neighborhood structures to locally improve solutions.

These neighborhoods have been combined within a general VNS/VND approach. The results on complete Euclidean instances have been compared with those of three so-far leading meta-heuristics for this problem, namely a permutation, a random-key, and an edge-set coded evolutionary algorithm. In both categories, solution quality as well as computation time, VNS exhibits results clearly superior to those of these EAs. In particular when the running time is strongly limited the solution quality of the VNS approach is substantially better.

In addition, an evolutionary algorithm has been presented that encodes candidate trees only incompletely by node levels and uses a decoding procedure to complement solutions. Similarly, an ant colony optimization algorithm is introduced where pheromone values are associated to the assignment of nodes to levels and the same

decoding procedure is applied. In both metaheuristics, candidate spanning trees are further improved by a choice of neighborhood searches.

Our results show that the level-based evolutionary algorithm with the arc exchange neighborhood search as local optimization leads to significantly better results in runs with a tight limit on execution time when compared to the variable neighborhood search and the ant colony optimization algorithm. When running time is less critical, the tables turn, and it is the ant colony optimization algorithm that improves on the previous best known optima.

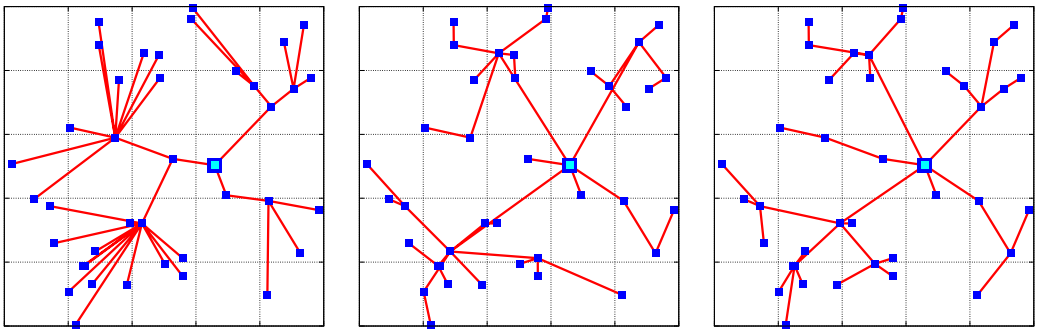
Chapter 8

Clustering-Based Construction Heuristic

8.1 Introduction

Simple and fast construction heuristics for the BDMST problem are primarily based on Prim's MST algorithm [109], for example the center based tree construction (CBTC) and randomized tree construction (RTC) [82] already discussed in detail in Section 3.2. To summarize, CBTC is well suited for problem instances with random edge weights, whereas it is much too greedy on Euclidean instances. It tends to create a backbone (the edges near the center) of relatively short edges and the majority of the nodes have to be connected to this backbone via rather long edges, see Fig. 8.1(a). A good solution, like this one shown in Fig. 8.1(c), only contains a few long edges in the backbone to span the whole area so the large number of remaining nodes can be connected as leaves by much cheaper edges. In RTC the nodes are connected to the tree in random order, so at least the possibility to include longer edges into the backbone at the beginning of the algorithm is increased, see Fig. 8.1(b). For Euclidean instances RTC has been so far the best choice to quickly create a first solution as basis for exact or metaheuristic approaches.

In the following we will introduce a new construction heuristic for the BDMST problem which is especially suited for very large Euclidean instances. It is based on a hierarchical clustering that guides the algorithm to find a good backbone.



(a) CBTC (8.284). (b) RTC (5.725). (c) Optimum (5.195).

Figure 8.1: A diameter-constrained tree with $D = 6$ constructed using (a) the CBTC heuristic, compared to (b) RTC (best solution from 100 runs) and (c) the optimal solution (complete, Euclidean graph with 40 nodes distributed randomly in the unit square, the corresponding objective values are given in parentheses).

8.2 Clustering-Based Construction Heuristic

The clustering-based construction heuristic can be divided into three steps, cf. Fig. 8.2: Creating a hierarchical clustering (*dendrogram*) of all instance nodes based on the edge costs, deriving a height-restricted clustering (HRC) from this dendrogram, and finding for each cluster in the HRC a good root (center) node. Different algorithms are used for the various steps, Fig. 8.3 shows the algorithmic point of view: After generating the hierarchical clustering, cuts through the dendrogram have to be found to get a height-restricted clustering. In a first step, initial cutting positions are computed analytically where a parameter is optimized using simple binary search. Afterwards, the determined cutting positions are further refined using a GRASP-like procedure [123]. To evaluate the cuts through the dendrogram, diameter-constrained trees have to be derived from the corresponding HRCs, and since this have to be done multiple times a fast greedy heuristic with additional local search is used. Finally, when a good set of cutting positions has been identified to create a height-restricted clustering, dynamic programming operating on a reduced search space is applied to get the approximately best BDMST from this HRC, again improved by local search.

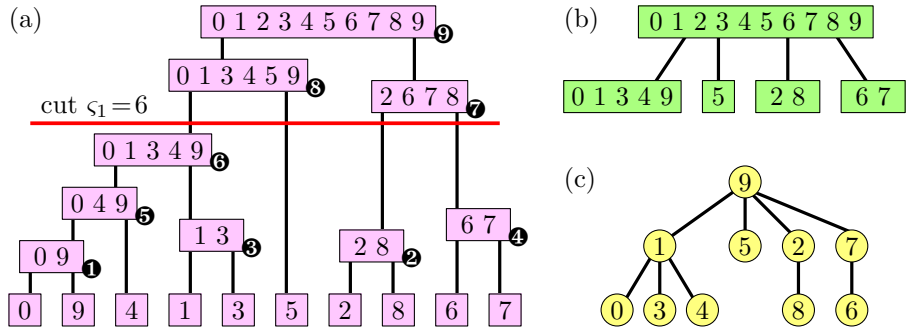


Figure 8.2: Hierarchical clustering (a), height restricted clustering (b), and the resulting diameter constrained tree with $D = 4$ (c) after choosing a root for each cluster in (b). In (a) ❶...❾ denote the merge numbers, i.e., at which time the two corresponding clusters have been merged during the agglomeration process.

8.2.1 Hierarchical Clustering

For the purpose of creating a good backbone especially for an Euclidean instance of the BDMST problem agglomerative hierarchical clustering provides a reasonable guidance. To get spatially confined areas, two clusters A and B are merged when $\max\{c_{a,b} : a \in A, b \in B\}$ is minimal over all pairs of clusters (complete linkage clustering [80]).

The agglomeration starts with each node being an individual cluster, and stops when all nodes are merged within one single cluster. The resulting hierarchical clustering can be illustrated as a binary tree, also referred to as a *dendrogram*, with $|V|-1$ inner nodes each representing one merging operation during clustering and $|V|$ leaves; see Fig. 8.2(a) for an example with $|V| = 10$. An inner node's distance from the leaves indicates when the two corresponding clusters – relative to each other – have been merged.

8.2.2 Height Restricted Clustering

After performing the agglomerative hierarchical clustering, the resulting dendrogram has to be transformed into a height-restricted clustering (HRC) for the BDMST, i.e., into a representation of the clustering respecting the diameter and thus the height condition. The dendrogram itself cannot directly act as HRC since in general it will violate this constraint, see Fig. 8.2(a). Therefore, some of the nodes in the

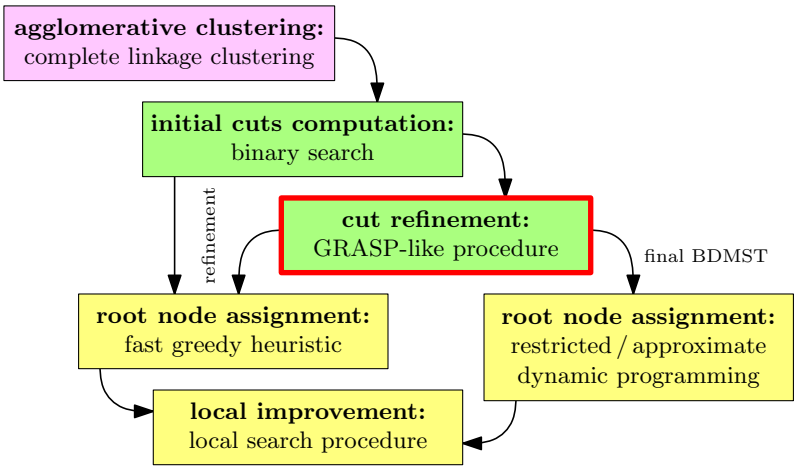


Figure 8.3: Algorithmic overview of the clustering-based construction heuristic.

dendrogram have to be merged to finally get a tree of height $H-1$, the HRC for the BDMST; see Fig. 8.2(b) for a diameter of $D=4$.

For the quality of the resulting tree this merging of dendrogram nodes is a crucial step, worth investing significant effort. It can be described by $H-1$ *cuts* through the dendrogram defining which nodes of it will also become part of the height-restricted clustering and which are merged with their parent clusters. As an example, starting at the root containing all instance nodes agglomerated within one single cluster the cut illustrated in Fig. 8.2(a) defines the dendrogram nodes $\{0,1,3,4,9\}$, $\{5\}$, $\{2,8\}$, and $\{6,7\}$ to become direct successors of the root cluster in the height-restricted clustering.

One fundamental question arising in this context is the way of defining the cutting positions through the dendrogram. This could be, for example, based on the height of the (sub-)dendrogram(s), but this choice would be problematic in two senses. On one hand, the cut illustrated in Fig. 8.2(a) cannot be described based on height information: The whole dendrogram has a height of five. With a cut at height three, i.e., in this example the first sub-dendrogram of height ≤ 3 on a path from the root to a leaf will become direct successor of the root cluster in the HRC, we are able to specify the clusters $\{0,1,3,4,9\}$ and $\{5\}$, but not $\{2,8\}$ and $\{6,7\}$ since their parent cluster already has a height of three. Reducing the cutting height to two is no solution since then the cluster $\{0,1,3,4,9\}$ will be split to $\{0,4,9\}$ and $\{1,3\}$. On the other hand, the height of the dendrogram will be, when the instance graph is not specifically structured, in general approximately $\log_2(2n)$. Preliminary tests showed that especially for larger diameter bounds, e.g., $D \geq 20$ for a 1000 node instance

Algorithm 16: buildHRC(p_d, p_p, ς, j)

input : reference to dendrogram node p_d; reference to parent cluster in the HRC p_p; cutting positions ς_i, $i = 1, \ldots, H-1$; current cut index j
output: height-restricted clustering for the BDMST

```
1 if p_d.merge# > ς_j then
2     forall children p_c of p_d do
3         buildHRC(p_c, p_p, ς, j)
4 else
5     create new HRC node p_n for instance nodes agglomerated in p_d
6     connect p_n to its parent cluster p_p within the HRC
7     if j < H−1 then
8         forall children p_c of p_d do
9             buildHRC(p_c, p_n, ς, j + 1)
```

where the height of the dendrogram would be near 11, the height is not flexible enough to obtain good solutions. Alternatively, the precise information at which iteration two clusters have been merged in the agglomeration process turns out to be a good criterion. This *merge number* (or *merge#*), which allows a finer grained control of the cutting positions, can be stored within each node of the dendrogram, where the leaves have a merge number of zero, and the root $|V| - 1$.

Based on the merge numbers cutting positions ς are computed as

$$\varsigma_i = (|V|-1) - 2^{i \cdot \frac{\log_2(x)}{H-1}} \qquad i = 1, \ldots, H-1, \tag{8.1}$$

where x is a strategy parameter. This formula is motivated by a perfectly balanced tree, where parameter x can be interpreted as the number of nodes that shall form the backbone.

These cutting positions can now be utilized to build the height-restricted clustering for the BDMST, as depicted in Algorithm 16. The recursion is started with the root of the dendrogram (p_d), a reference to the newly created root node of the HRC (p_p), the computed cutting positions (ς), and 1 for the current cut index j.

An experimental evaluation with a simple greedy construction heuristic described in the next section revealed that for $D \geq 6$ promising values for x can be found close to $|V|$; see Fig. 8.4. Only in case of the smallest possible even diameter of four the picture is inverted and x should be chosen near $\frac{|V|}{10}$. The rather continuous and mostly monotonic increase or decrease of the curve further suggests to apply

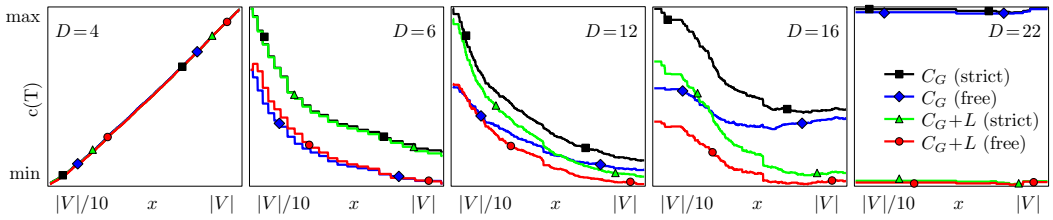


Figure 8.4: Obtained objective values (scaled to minimal and maximal values) over parameter x of Equation (8.1) (ranging from $\frac{|V|}{10}$ to $|V|$) for various diameter bounds on an Euclidean instance with 1000 nodes distributed randomly in the unit square. The trees were computed using the simple greedy construction heuristic C_G (Section 8.2.3) with and without local improvement (L), respectively with free leaf nodes or leaves strictly following the clustering (see Section 8.2.4 for details).

binary search to determine an approximately best value for x for a specific Euclidean instance and diameter bound, if time allows and the heuristic can be run multiple times.

8.2.3 Determining Good Root Nodes

Finally, from the height-restricted clustering a BDMST has to be derived by identifying for each (sub-)cluster an appropriate root; see Fig. 8.2(b) and (c). This can be done heuristically in a greedy fashion based on rough cost estimations for each cluster followed by a local improvement step, or by more sophisticated approaches based on dynamic programming [19].

In the following we will require a more formal and in some points augmented definition of a height-restricted hierarchical clustering. Let $C^0 = \{C_1^0, \ldots, C_{|V|}^0\}$ be the set of clusters at the lowest level 0, where each node of V forms an individual cluster. Moreover, let $C^k = \{C_1^k, \ldots, C_{i_k}^k\}$ be the clustering at the higher levels $k = 1, \ldots, H$. All C_i^k, $i = 1, \ldots, i_k$, are pairwise disjoint, and $C_1^k \cup C_2^k \cup \ldots \cup C_{i_k}^k = C^{k-1}$. C^H is the highest level, and it is singleton, i.e., $C^H = \{C_1^H\}$; it refers to all nodes in V aggregated within one cluster. Furthermore, by $V(C_i^k)$ we denote the set of nodes in V represented by the cluster C_i^k, i.e., the nodes part of this cluster and all its sub-clusters at lower levels; $V(C^k) = V(C_1^k) \cup \ldots \cup V(C_{i_k}^k) = V$, and $V(C_1^k) \cap \ldots \cap V(C_{i_k}^k) = \emptyset$, for all $k = 0, \ldots, H$.

This definition mainly corresponds to the simple height-restricted clustering previously presented in Fig. 8.2(b) and computed by Algorithm 16, with two exceptions:

Algorithm 17: greedyRoots(r)

input : root r of the HRC
output: a root node for each cluster, and if D is odd a center edge for the root cluster of the HRC

```
1  forall v ∈ V do available[v] ← true
2  if D is even then
3      assignRoot(r)
4  else
5      forall children r_c of r do
6          r_c.stars ← ( )
7          foreach node v ∈ V(r_c) do
8              compute star s_v: connect to v all nodes V(r_c) \ {v} of cluster r_c
9              r_c.stars.append(s_v)
10         sort r_c.stars ascending according to the costs of the stars
11         r_c.root ← v of least cost star s_v ∈ r_c.stars
12     r.root ← best center edge for the roots of the child clusters r_c
13     for both center nodes c_1 and c_2 do available[c_i] ← false
14     forall children r_c of r do
15         if not available[r_c.root] then
16             r_c.root ← next best root node based on r_c.stars
17         available[r_c.root] ← false
18         forall children p_c of r_c do assignRoot(p_c)
```

The clusters at level zero corresponding to the individual nodes have not been realized explicitly, and not all leaves of the HRC created by Algorithm 16 have to be found at level one. In the latter case such a leaf can only contain exactly one node $v \in V$, therefore the HRC can be augmented with (virtual) nodes to connect the corresponding cluster at level zero with a leaf at a level ≥ 2.

Greedy Heuristic with Local Improvement

A simple greedy heuristic to find an initial root for each cluster C_i^k can be based on so-called *stars*, i.e., trees with a diameter of two where a single node v of the cluster acts as center while the remaining nodes $V(C_i^k) \setminus \{v\}$ are connected directly to it. Such a star can be computed for every node belonging to the cluster, the

Algorithm 18: assignRoot(p)

input : reference p to a node of the HRC
output: for cluster p a sorted list *p.stars* of diameter 2 trees and a root *p.root*

```
1  p.root ← ∅
2  p.stars ← ( )
3  foreach node v ∈ V(p) do
4  |  compute star s_v: connect to v all nodes V(p) \ {v} of current cluster p
5  |  p.stars.append(s_v)
6  sort p.stars ascending according to the costs of the stars
7  scan p.stars from beginning for first s_v where available[v] = true
8  if such a star s_v could be found then
9  |  p.root ← v
10 |  available[v] ← false
11 |  forall children p_c of p do assignRoot(p_c)
```

center node v leading to a star of minimal costs for C_i^k is chosen as root for this cluster. The heuristic starts at cluster C^H and assigns roots to clusters top-down until reaching the leaves of the simple height-restricted clustering. Note that a node already selected as root at a level l no longer has to be considered in levels less than l, which can also cause an empty cluster in case all nodes of it are already used as roots at higher levels.

Algorithms 17 and 18 illustrate this heuristic in more detail. An array *available* is used to indicate whether a node still can be selected as root in a (sub-)cluster. The even-diameter case is much simpler to handle: For each cluster represented by a node in the simple HRC all possible stars are computed and gathered within a list. This list is sorted in ascending order according to the costs of the stars, and the still available node leading to the cheapest star is chosen as root for the current cluster.

While this procedure can also be used for the root cluster C_1^H of the height-restricted clustering when D is even, not a single node but an edge has to be selected as center of the BDMST in case the diameter bound is odd. The corresponding algorithm would be to compute the cheapest BDMST with a diameter of three for the whole instance and use the resulting center edge also as center for the root cluster. However, in general this would lead to a much too long center edge because in a good BDMST this edge becomes shorter with increasing diameter bound since it no longer has to span a larger area. To make a better choice in a first step a reasonable root node is computed for every cluster C_i^{H-1} at level $H-1$. These roots are the only nodes that

Algorithm 19: locallyImprove(r)

input : root r of the HRC
output: locally improved roots for each cluster of the HRC

```
1  costs* ← costs c(T) of the current BDMST T derived from the HRC
2  T* ← T
3  repeat
4      improved ← false
5      forall v ∈ V do available[v] ← true
6      r.root ← best center to connect current roots of all child clusters r_c of r
7      update available[·] accordingly;      // for center node/edge of BDMST
8      recursively find for each sub-cluster of r the currently local optimal root:
           • consider only nodes v_i with available[v_i] = true,
           • provide for connection costs to root of parent cluster,
           • if no leaf of the HRC:
             consider costs to connect the current roots of direct successor
       clusters,
           • always update available[·] accordingly
9      evaluate current BDMST T derived from the assigned roots in the HRC
10     if c(T) < costs* then
11         T* ← T
12         costs* ← c(T)
13         improved ← true
14 until improved = false
15 restore best tree T*
```

have to be linked directly to the center edge. Based on this observation now a more suitable center edge can be determined by computing a diameter three BDMST only considering the connection costs of the root nodes of the clusters C_i^{H-1}.

While this heuristic runs in $\mathcal{O}(H \cdot n^2)$ when D is even, the selection of the center edge in the odd-diameter case adds a term of $\mathcal{O}(\delta^r \cdot m)$, with δ^r being the branching factor of the root cluster in the HRC, leading to an overall runtime complexity of $\mathcal{O}(\delta^r \cdot m + H \cdot n^2)$.

In a following local improvement step the selection of root nodes (and the center edge) is refined. In case a cluster C_i^k with chosen root v is no leaf of the simple height-restricted clustering not all nodes of $V(C_i^k) \setminus \{v\}$ will straightly connect to v in the final tree but only the roots of the direct sub-clusters of C_i^k at level $k-1$, cf.

Fig. 8.2(c). This sub-cluster root information was not available in the greedy construction process since the assignment from root nodes to clusters was performed top-down but now can be used to adapt for each cluster the chosen root node iteratively, see Algorithm 19. This refinement of assigned roots to clusters requires for one iteration time $\mathcal{O}(H \cdot \delta^{\max} \cdot n)$ if D is even, where $\delta^{\max}$ is the maximal branching factor in the height-restricted clustering, and $\mathcal{O}(\delta^r \cdot m + H \cdot \delta^{\max} \cdot n)$ in the odd-diameter case.

Attention has to be payed to the fact that a local improving move (new root for a specific cluster C_i^k) not necessarily leads to an improvement of the overall BDMST. Choosing a node u instead of v as root node for C_i^k can have various effects on this part of the tree. E.g. u no longer can act as root for one of the clusters at a lower level; moreover, v now has to be connected as a new leaf to the BDMST if not chosen as a root within one of the sub-clusters of C_i^k. As a consequence, the stopping criterion is not based on the existence or absence of an local improvement move but on the costs of the whole derived BDMST.

Dynamic Programming

The multiple effects on the tree when choosing a specific node as root for a cluster increase the complexity of deriving an optimal BDMST for a given hierarchical clustering to such an extent that it is in general computationally unattractive. Nevertheless, when restricting the search space it is possible to formulate an efficient dynamic programming approach for this problem.

Let $c(C_i^k, v)$ denote the minimum costs of the subtree of the BDMST defined by the cluster C_i^k if it is rooted at node $v \in V(C_i^k)$, i.e., node v has been chosen as root for cluster C_i^k. These costs can now be recursively defined for each level and node of a cluster as follows:

$$c(C^0_{\mathrm{ord}(v)}, v) = 0 \qquad \forall v \in V \tag{8.2}$$

$$\phi(C_i^k, v) = \sum_{C_j^{k-1} \in C_i^k \setminus \{C_{j'}^{k-1}\}} \min_{u \in V(C_j^{k-1})} \Big(c_{v,u} + c(C_j^{k-1}, u) \Big)$$

$$c(C_i^k, v) = c(C_{j'}^{k-1}, v) + \phi(C_i^k, v)$$

$$\forall k = 1, \ldots, H, \quad \forall v \in V(C_i^k), \quad C_{j'}^{k-1} \in C_i^k \mid v \in V(C_{j'}^{k-1}) \tag{8.3}$$

At level zero each node is a single cluster. Therefore, in (8.2) the costs of the corresponding subtrees can be initialized with zero ($\mathrm{ord}(v)$ assigns each node $v \in V$

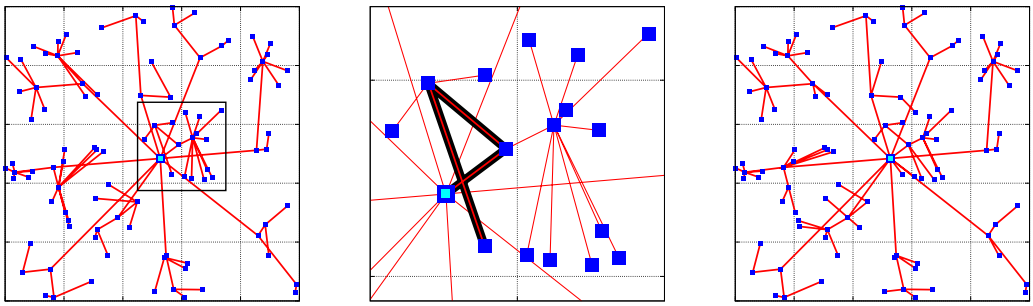
a unique index within 1 and $|V|$). In the remaining levels we restrict the root for a cluster C_i^k to nodes that are already roots in one of its direct sub-clusters C_j^{k-1}, $C_j^{k-1} \in C_i^k$. Then the costs $c(C_i^k, v)$ are composed of the costs of the subtree rooted at v at level $k-1$ plus – for all remaining direct sub-clusters – the minimal costs to connect a node u of a sub-cluster with its subtree to v, referred to as $\phi(C_i^k, v)$ in (8.3). After deriving all these costs in a bottom-up fashion, optimal root nodes leading to these costs can be chosen top-down in a second pass.

Limiting the potential roots of a cluster to root nodes within one of its sub-clusters obviously leads to suboptimal trees, especially when the diameter bound is loose and each root node only has very few connections. Moreover, using the whole subtree rooted at v from a cluster at level $k-1$ for a cluster at level k implies that this subtree is moved one edge towards the center of the BDMST and therefore does not exploit the full possible height, a problem arising for every cluster at every level $k \geq 2$.

Beside other implications one major point when choosing a node v as root is that it no longer has to be connected elsewhere in the tree. When computing $c(C_i^k, v)$ and selecting another node w from the same sub-cluster $C_{j'}^{k-1}$ that v is also part of, then the costs $c(C_{j'}^{k-1}, w)$ also contain the costs to connect node v (perhaps as root of one of the sub-clusters, more likely as a leaf of the BDMST). To exactly compute the contribution of v to the costs of $c(C_{j'}^{k-1}, w)$ is in general not worth the (huge) effort, in particular when considering the costs of edges between root nodes in relation to the costs of connecting a leaf to the tree via a short edge, which is the goal of the whole clustering heuristic.

This observation can be used to formulate an approximate dynamic programming approach utilizing a correction value κ_v for each node $v \in V$ which estimates the costs arising when v has to be connected as leaf to the BDMST. There are various possibilities to define these correction values, preliminary tests showed that a simple choice usually is sufficient: The subtrees computed at level one correspond to stars with diameter two. For each cluster at level one the cheapest star is determined, and for a node v of such a cluster, κ_v are the costs to connect it to the center of the best star. This now leads to the following reformulation of the recursion to compute the costs $c(C_i^k, v)$:

$$
\begin{aligned}
c(C_i^k, v) = \min \Big(c(C_{j'}^{k-1}, v),\ c_{v,w} + c(C_{j'}^{k-1}, w) - \kappa_v \Big) + \phi(C_i^k, v) \\
\forall k = 1, \ldots, H, \quad \forall v \in V(C_i^k), \quad C_{j'}^{k-1} \in C_i^k \mid v \in V(C_{j'}^{k-1}), \\
w \in V(C_{j'}^{k-1}) \mid w \neq v
\end{aligned}
\tag{8.4}
$$



(a) BDMST with leaves following the clustering.

(b) Problematic path in the solution (a).

(c) BDMST with free leaves.

Figure 8.5: Problem arising when strictly following the clustering on a complete Euclidean instance with 100 nodes and $D = 6$. In (b) the interesting area of (a) near the center of the BDMST is shown enlarged, a problematic path is highlighted. The solution can significantly be improved when leaf nodes are free to connect to any node of the backbone (c).

Both dynamic programming approaches compute roots for clusters within time $\mathcal{O}(H \cdot n^2)$ and $\mathcal{O}(n \cdot m + H \cdot n^2)$, respectively, for the even and odd-diameter case.

8.2.4 Inherent Problem of Clustering

One problem arising when strictly following the clustering to the leaves of the BDMST – in particular when the diameter bound is weak in comparison to the number of nodes – is illustrated in Fig. 8.5. This situation can be observed when a node of a (sub-)cluster is chosen to be part of the backbone (a root node), and other nodes close to it not. The node right below the center of the BDMST in Fig. 8.5(a) and the close-up (b) is connected to the root of the last sub-cluster following the hierarchy of the clustering neglecting the fact that there are much cheaper opportunities. The negative effect on the solution quality is noticeable especially for leaf nodes but is not restricted to them.

A possibility to deal with this problem is to *free* the leaves from their strict membership to a specific cluster and to allow them to connect to the root of any cluster in the cheapest possible way. This leads to – also visually observable – much better results as demonstrated in Fig. 8.5(c).

Algorithm 20: refineCuts(ς)

input : cutting positions ς_i, $i = 1, \ldots, H-1$
output: improved cutting positions

```
1  T* ← buildTree(ς) ;                                  // currently best BDMST T*
2  ς* ← ς ;                                  // currently best cutting positions ς*
3  clear cache for sets of cutting positions and insert ς
4  lwi ← 0 ;                                            // loops without improvement
5  repeat
6      for i = 1, ..., H − 1 do
7          if i = 1 then Δ ← (|V| − 1) − ς*_1 else Δ ← ς*_{i−1} − ς*_i
8          repeat ς_i ← ς*_i + Δ · N(μ, σ²) until check(ς_i) is ok
9      if ς ∈ cache then
10         lwi ← lwi + 1
11         continue
12     insert ς into cache
13     T ← buildTree(ς)
14     if c(T) < c(T*) then
15         T* ← T
16         ς* ← ς
17         lwi ← 0
18     else
19         lwi ← lwi + 1
20 until lwi ≥ l_max
```

8.3 Refinement of Cutting Positions

In Section 8.2.2 the computation of initial cutting positions ς_i, $i = 1, \ldots, H-1$, through the dendrogram to derive a height-restricted clustering has been presented. Since these ς_i have a formidable impact on solution quality we additionally implemented an approach similar to a greedy randomized adaptive search procedure (GRASP) [123] to further refine them, see Algorithm 20. In each iteration all cutting positions of the currently best solution are perturbated using the difference Δ to the next lower indexed cutting position (for ς_1 the value $(|V| - 1) - \varsigma_1$ is used), multiplied with a Gaussian distributed random value $N(\mu, \sigma^2)$.

To derive an actual BDMST from the cutting positions ς in `buildTree(ς)` a really fast construction heuristic should be applied like the greedy heuristic with local

search presented in the previous Section 8.2.3. To avoid redundant computations a cache is used to identify sets of cutting positions ς already evaluated. Furthermore, a new cutting position ς_i is only accepted if it lies within the interval $[|V|-2, 1]$ and if it differs from all ς_j, $j < i$, which is tested in `check(`ς_i`)`.

The whole refinement process is stopped when $l_{\max}$ iterations without improvement have been performed, or no sets of new cutting positions could be found, respectively.

8.4 The Odd-Diameter Case

In case the diameter bound D is odd no single node forms the center of a BDMST but an edge, the *center edge*, connecting the two center nodes. This increases – when considering complete instance graphs – the number of tests to determine the best center from $O(|V|^2)$ (when D is even) to $O(|V|^3)$. Due to the large number of trees that have to be derived from a height-restricted clustering in the various steps of the clustering heuristic the complexity increase has a substantial impact on the runtime.

An opportunity to speed-up computations in the odd-diameter case is to reduce in a preprocessing step the number of potential center edges that are considered. Given a height-restricted clustering the center edge (building the backbone at level H) will connect to every root node of the clusters at level $H-1$. In case the Euclidean instance is sufficiently large the set of plausible center edges can be restricted to edges where both endpoints are within the rectangle spanned by these root nodes. Since the optimal roots for the clusters at level $H-1$ are not known they have to be heuristically determined. This can easily be done by computing diameter two stars for all these clusters and then choosing for each cluster the center of the cheapest star as representative for its root node. With increasing diameter bound the rectangle spanned by these nodes at level $H-1$ becomes smaller, thus the number of potential center edges is further reduced.

When searching for a value of parameter x in Equation (8.1) to determine a good set of initial cutting positions through the dendrogram using binary search the charts as shown in Fig. 8.4 (objective value of the BDMST over x) are almost identical for an odd diameter D and the even diameter $D-1$. That is, a good value of x for an even D usually is also an appropriate choice for $D+1$ making it unnecessary to derive odd BDMSTs with an center edge in this step.

Unfortunately, this strong connection between the odd diameter D case and its even correspondent $D-1$ is no longer observable in the greedy randomized adaptive search

procedure to refine the initially calculated cutting positions. Since the runtime increases dramatically especially when the diameter bound is tight we implemented an additional variant: In general just BDMSTs with an even diameter $D-1$ are derived during the refinement process. Only in case a new best solution is found also a BDMST with the correct odd diameter D – managed separately – is computed based on the current HRC.

8.5 Local Search Neighborhood

The heuristics presented so far to derive a good tree from a height-restricted clustering can also be used to locally improve solutions obtained from another (meta-)heuristic since from each valid solution a corresponding hierarchical clustering of the nodes can be determined easily.

Creating a simple height-restricted clustering as shown in Fig. 8.2(b) from a diameter-constrained tree (c) is straight-forward, only root nodes not already part of the leaves of the HRC (just node 9 in the figure) have to be treated separately. Every such root has to be assigned to one of its direct sub-clusters where it additionally has to be propagated to further sub-clusters until a leaf of the HRC is reached. The decision which sub-cluster to choose can be based on the same criterion like in the agglomeration process the choice which clusters to merge, i.e., a root node is assigned to the sub-cluster where the maximum distance to a node of it is minimal.

8.6 Computational Results

The experiments have been performed on a dual-core AMD Opteron 2214 machine (2.2GHz) utilizing benchmark instances already used in the corresponding literature (e.g., [116, 74]) from Beasley's OR-Library [17, 16] originally proposed for the Euclidean Steiner tree problem. These complete instances contain point coordinates in the unit square, and the Euclidean distances between each pair of points are taken as edge costs. As performance differences are more significant on larger instances, we restricted for our experiments the set of test instances to the largest available ones with 1000 nodes. This class contains 15 different instances, but since the heuristics perform quite similar on all of them we list in the following detailed results only for the first three instances (average values of 30 independent runs, standard deviations in parentheses).

Tables 8.1 and 8.2 summarize the results obtained for the various heuristics when the diameter is even, while Tables 8.3 and 8.4 cover the odd-diameter case. Given

Table 8.1: Objective values of solutions obtained from CBTC, RTC, and the clustering heuristic C (binary search (b), cut refinement (r), local search (L), dynamic programming approaches d^A and d^B for assigning roots) on complete Euclidean instances with 1000 nodes, even diameter case. Additionally, the table lists the maximum times (sec) of one of the clustering heuristics (time limit for CBTC/RTC). Mean values over 30 runs, standard deviations are given parentheses; C_bL and CBTC (within time limit) are deterministic.

Instance					without VND												
#	D	CBTC	RTC		C_bL	$C_{br}L$		$C_{br}d^A$		$C_{br}d^B$		$C_{br}d^AL$		$C_{br}d^BL$		t[s]
00	4	333.1778	146.2846	(4.3087)	**68.4293**	**68.4293**	(0.0000)	68.7405	(0.0000)	**68.4293**	(0.0000)	**68.4293**	(0.0000)	**68.4293**	(0.0000)	2.55
	6	319.1373	80.9461	(2.2105)	57.3387	44.7858	(3.0416)	45.6181	(3.5626)	44.6655	(2.9398)	44.7195	(2.9398)	**44.6494**	(2.9624)	4.98
	8	298.7454	53.3496	(1.4402)	44.1488	37.0520	(1.5646)	37.9215	(1.8261)	36.9848	(1.6640)	37.0715	(1.6640)	**36.9725**	(1.6523)	5.86
	10	271.3976	41.2373	(0.5789)	37.7984	33.7021	(0.7689)	34.9211	(0.9216)	**33.5717**	(0.8710)	33.8305	(0.8710)	**33.5717**	(0.8880)	7.18
	12	261.6350	35.8323	(0.4741)	34.3425	32.4345	(0.3511)	33.8426	(0.5355)	**32.2621**	(0.4062)	32.7360	(0.4062)	**32.2621**	(0.3602)	6.69
	14	240.9920	33.4520	(0.2481)	32.1548	31.4093	(0.1791)	32.6319	(0.3028)	31.2290	(0.2537)	31.6736	(0.2537)	**31.2230**	(0.1733)	7.01
	16	223.4451	32.2899	(0.1421)	31.0876	30.8778	(0.0700)	32.2592	(0.3049)	**30.7887**	(0.1324)	31.2125	(0.1324)	**30.7887**	(0.1496)	6.72
	18	223.6044	31.6838	(0.1233)	30.8274	30.5872	(0.0328)	31.0783	(0.2674)	30.2669	(0.1847)	30.7697	(0.1847)	**30.2323**	(0.1274)	6.99
	20	205.1811	31.3145	(0.1156)	30.5581	30.4872	(0.0518)	30.5922	(0.1056)	30.2452	(0.1096)	30.3671	(0.1096)	**30.2020**	(0.0647)	7.68
	22	187.5576	31.1179	(0.1176)	30.5841	30.4394	(0.0542)	30.5380	(0.1588)	30.3451	(0.0693)	30.3139	(0.0693)	**30.2064**	(0.0605)	9.29
	24	176.7546	31.0607	(0.1122)	30.5997	30.4404	(0.0359)	30.1811	(0.0531)	30.4632	(0.0815)	**30.1446**	(0.0815)	30.1620	(0.0440)	9.08
01	4	319.8968	144.8287	(3.5643)	67.8938	**67.7157**	(0.0476)	67.8526	(0.0477)	67.8157	(0.0476)	**67.7157**	(0.0476)	**67.7157**	(0.0476)	2.56
	6	308.6494	81.2136	(2.3047)	56.6842	49.5692	(4.3732)	51.3624	(5.1963)	49.3140	(4.3217)	49.6271	(4.3217)	**49.3089**	(4.1812)	4.19
	8	283.7449	52.7770	(1.3545)	44.6115	37.6394	(1.3655)	38.5604	(1.4182)	37.6162	(1.4567)	37.6659	(1.4567)	**37.5570**	(1.3699)	5.86
	10	255.9973	41.1043	(0.5924)	37.1610	33.3137	(0.5323)	34.5437	(0.6932)	**33.0283**	(0.5995)	33.3021	(0.5995)	**33.0283**	(0.5810)	6.79
	12	232.2019	35.7376	(0.2761)	33.9504	32.1760	(0.2941)	33.4031	(0.4189)	**31.8839**	(0.2708)	32.2610	(0.2708)	**31.8839**	(0.3176)	7.40
	14	222.9918	33.2274	(0.2200)	32.3991	31.1289	(0.1255)	32.4471	(0.2406)	**30.9184**	(0.1211)	31.3850	(0.1211)	**30.9184**	(0.1743)	7.21
	16	216.4585	32.0792	(0.1378)	31.2347	30.6651	(0.1500)	31.7030	(0.4244)	30.5019	(0.0759)	30.8763	(0.0759)	**30.3795**	(0.2296)	6.91
	18	202.0703	31.5349	(0.1166)	30.4893	30.3661	(0.0784)	31.2821	(0.1868)	30.1496	(0.1114)	30.6215	(0.1114)	**30.0754**	(0.1472)	8.06
	20	201.3354	31.1884	(0.1171)	30.4513	30.3367	(0.0382)	30.6516	(0.2416)	30.0307	(0.1244)	30.4777	(0.1244)	**29.9864**	(0.1184)	7.60
	22	182.3676	31.0098	(0.1243)	30.5410	30.4810	(0.0229)	29.9266	(0.1115)	30.1627	(0.0764)	**29.9192**	(0.0764)	30.0288	(0.0711)	7.87
	24	173.4081	30.8659	(0.0793)	30.6320	30.5306	(0.0260)	**29.8572**	(0.0412)	30.2587	(0.0412)	**29.8572**	(0.0412)	30.0813	(0.1315)	8.70
02	4	323.8917	145.6452	(3.7690)	**68.9695**	**68.9695**	(0.0000)	69.2646	(0.0000)	**68.9695**	(0.0000)	**68.9695**	(0.0000)	**68.9695**	(0.0000)	2.53
	6	295.9416	80.6684	(2.0948)	57.0119	51.0036	(4.1721)	53.4263	(5.2519)	50.6211	(4.3343)	51.2829	(4.3343)	**50.6046**	(3.9952)	3.90
	8	287.6123	52.8909	(1.3366)	44.7386	35.9854	(0.0733)	36.9639	(0.1743)	**35.9041**	(0.0669)	36.0518	(0.0669)	**35.9041**	(0.0963)	6.42
	10	258.2323	40.8925	(0.6345)	37.3424	33.2714	(0.2865)	34.4700	(0.3700)	**33.2334**	(0.2805)	33.4197	(0.2805)	**33.2334**	(0.2819)	6.65
	12	245.4804	35.4809	(0.4291)	34.2121	31.9162	(0.2604)	33.2711	(0.3446)	31.8456	(0.3724)	32.1527	(0.3724)	**31.8363**	(0.3294)	6.64
	14	239.5847	33.0912	(0.1962)	31.9710	31.0000	(0.2179)	32.2365	(0.3541)	30.9223	(0.3121)	31.2276	(0.3121)	**30.8871**	(0.2742)	6.59
	16	231.7629	31.8738	(0.1564)	30.6676	30.5329	(0.1449)	31.6423	(0.3417)	30.5096	(0.2139)	30.7029	(0.2139)	**30.4300**	(0.3045)	6.93
	18	213.1380	31.2912	(0.0902)	30.4643	30.0813	(0.0940)	30.5786	(0.2185)	29.8776	(0.1196)	30.0950	(0.1196)	**29.7745**	(0.2144)	7.83
	20	200.7924	30.9771	(0.0778)	30.0950	30.0337	(0.0528)	30.3278	(0.1727)	29.7651	(0.0754)	29.8770	(0.0754)	**29.6536**	(0.0774)	7.59
	22	186.5278	30.7510	(0.1028)	30.1562	30.0266	(0.0528)	29.8146	(0.0941)	29.8711	(0.0952)	29.8131	(0.0952)	**29.6772**	(0.1013)	8.83
	24	181.6017	30.6342	(0.1381)	30.2402	29.9758	(0.0643)	29.7326	(0.0406)	29.8932	(0.0385)	29.7314	(0.0385)	**29.7242**	(0.0918)	10.64

Table 8.2: Objective values of solutions obtained from the same algorithms on instances as in Table 8.1, where the best solution is locally improved by a VND described in Section 7.2. For comparison the solution values of the ACO from Section 7.4 is given after one hour of computation.

Instance					with VND									
#	D	CBTC	RTC		C_bL	$C_{br}L$		$C_{br}d^AL$		$C_{br}d^BL$		ACO	
00	4	66.3860	66.0537	(0.2564)	**65.6725**	**65.6725**	(0.0000)	**65.6725**	(0.0000)	**65.6725**	(0.0000)	*66.5485*	*(0.1300)*
	6	41.5663	41.6762	(0.2597)	41.5787	**41.1946**	(0.4784)	41.2539	(0.2783)	41.2150	(0.3278)	*42.2666*	*(0.1203)*
	8	35.3437	35.2037	(0.2342)	34.6061	34.0972	(0.2710)	34.0504	(0.2874)	**33.9667**	(0.2960)	*34.8643*	*(0.0827)*
	10	32.2006	32.2369	(0.2736)	31.8518	31.2213	(0.1545)	31.1720	(0.1447)	**31.0408**	(0.1863)	*31.2158*	*(0.0870)*
	12	30.6040	30.3253	(0.2518)	29.6424	29.3811	(0.0824)	29.4594	(0.0952)	**29.3726**	(0.1192)	*28.8857*	*(0.1455)*
	14	28.8171	29.2421	(0.1887)	28.2135	**28.1360**	(0.1907)	28.2601	(0.2102)	28.1405	(0.1441)	*26.7998*	*(0.2710)*
	16	29.3602	28.4036	(0.2212)	**27.1433**	27.2165	(0.0614)	27.3989	(0.0904)	27.2841	(0.0677)	*25.6659*	*(0.0891)*
	18	27.5753	27.7371	(0.2576)	26.9135	26.7054	(0.0417)	26.7617	(0.0876)	**26.6980**	(0.0883)	*24.9721*	*(0.0485)*
	20	27.2770	27.2182	(0.1779)	**26.2943**	26.3903	(0.0955)	26.4811	(0.0710)	26.3564	(0.1031)	*24.5097*	*(0.0434)*
	22	26.7815	26.8119	(0.1741)	**25.9035**	26.1078	(0.0622)	26.0350	(0.0929)	25.9287	(0.1507)	*24.1486*	*(0.0896)*
	24	26.5512	26.4912	(0.2343)	**25.6094**	25.8073	(0.0386)	25.8281	(0.1095)	25.8046	(0.0972)	*23.9438*	*(0.1674)*
01	4	64.7319	**64.7081**	(0.1889)	64.8327	64.8327	(0.0000)	64.8327	(0.0000)	64.8327	(0.0000)	*65.3880*	*(0.1320)*
	6	41.6357	41.2002	(0.2930)	**40.7283**	41.1331	(0.0909)	41.1659	(0.0876)	41.2032	(0.0859)	*41.8799*	*(0.1185)*
	8	34.9962	34.8162	(0.2786)	34.2772	34.2077	(0.2897)	34.2428	(0.2514)	**34.1730**	(0.2379)	*34.4363*	*(0.0821)*
	10	31.8495	32.0129	(0.1959)	31.7840	30.8892	(0.1734)	30.9510	(0.1581)	**30.8128**	(0.1694)	*30.8802*	*(0.0964)*
	12	30.5000	30.1785	(0.2629)	29.6711	29.1753	(0.1506)	29.2489	(0.1836)	**29.1676**	(0.1566)	*28.5762*	*(0.1624)*
	14	29.5841	29.0310	(0.2469)	28.0571	**27.8590**	(0.1118)	28.0193	(0.0839)	27.8911	(0.1218)	*26.7925*	*(0.2662)*
	16	28.6480	28.1114	(0.2099)	27.1907	27.0579	(0.0383)	27.1706	(0.0654)	**27.0377**	(0.0997)	*25.6153*	*(0.1568)*
	18	27.6042	27.5220	(0.2179)	**26.5756**	26.5838	(0.0451)	26.7334	(0.1001)	26.6367	(0.0708)	*24.8529*	*(0.0696)*
	20	27.5157	26.9948	(0.2208)	26.1348	26.2519	(0.0526)	26.4150	(0.0873)	**26.1214**	(0.0723)	*24.3636*	*(0.0909)*
	22	27.0029	26.6309	(0.2395)	25.8205	25.8679	(0.0353)	25.9167	(0.0466)	**25.7924**	(0.0405)	*23.9628*	*(0.1291)*
	24	26.3927	26.3612	(0.2023)	**25.4279**	25.5766	(0.0857)	25.5088	(0.0347)	25.4857	(0.1436)	*23.6896*	*(0.1496)*
02	4	65.0116	64.7623	(0.1853)	**64.4312**	**64.4312**	(0.0000)	**64.4312**	(0.0000)	**64.4312**	(0.0000)	*65.5188*	*(0.1515)*
	6	**40.7349**	41.2438	(0.3294)	40.8028	41.0533	(0.1251)	41.2958	(0.2452)	41.1944	(0.1583)	*41.8806*	*(0.0830)*
	8	34.8123	34.8576	(0.3756)	34.1660	34.0139	(0.1236)	**34.0104**	(0.0630)	34.0569	(0.0399)	*34.4206*	*(0.0830)*
	10	32.5153	31.7946	(0.2564)	31.3344	30.7090	(0.1299)	30.7162	(0.1480)	**30.6506**	(0.1193)	*30.7889*	*(0.0927)*
	12	30.3992	30.0125	(0.2134)	29.4628	29.0580	(0.1258)	28.9847	(0.1344)	**28.9728**	(0.1697)	*28.1903*	*(0.1588)*
	14	28.8848	28.7565	(0.1979)	28.1242	27.8866	(0.1196)	28.0251	(0.1224)	**27.8472**	(0.1426)	*26.2120*	*(0.2075)*
	16	27.9989	27.9417	(0.1964)	**26.9242**	26.9744	(0.0682)	27.2007	(0.0828)	27.0769	(0.1136)	*25.2303*	*(0.0445)*
	18	27.5996	27.2587	(0.2280)	26.4321	26.3833	(0.0675)	26.4710	(0.0748)	**26.3214**	(0.0789)	*24.6526*	*(0.0948)*
	20	27.3748	26.7782	(0.1952)	25.9866	25.9405	(0.0776)	25.9985	(0.0606)	**25.8772**	(0.0879)	*24.1545*	*(0.1349)*
	22	26.6629	26.3459	(0.2105)	**25.4822**	25.5618	(0.0924)	25.7318	(0.0985)	25.5650	(0.0953)	*23.7834*	*(0.1287)*
	24	26.0186	26.1191	(0.2333)	**25.1808**	25.2422	(0.0738)	25.4261	(0.0788)	25.2403	(0.0859)	*23.5635*	*(0.1527)*

Table 8.3: Objective values of solutions obtained from CBTC, RTC, and the clustering heuristic C (binary search (b), fast cut refinement (r; based on even diameter case), local search (L), dynamic programming approaches d^A and d^B for assigning roots) on complete Euclidean instances with 1000 nodes, odd diameter case. Additionally, the table lists the maximum times (sec) of one of the clustering heuristics (time limit for CBTC/RTC). Mean values over 30 runs, standard deviations are given parentheses; C_bL is deterministic.

Instance		without VND																
#	D	CBTC		RTC		C_bL	$C_{br}L$		$C_{br}d^A$		$C_{br}d^B$		$C_{br}d^AL$		$C_{br}d^BL$		t[s]
00	5	244.1504	(2.9502)	117.3690	(2.1687)	62.5424	62.5424	(0.0000)	62.9691	(0.0000)	62.5132	(0.0000)	62.9691	(0.0000)	**62.4226**	(0.0000)	24.38
	7	224.5403	(2.4116)	67.2259	(1.5432)	55.0517	45.6928	(3.8075)	47.4495	(4.7193)	45.6770	(3.7782)	45.6899	(3.7640)	**45.6409**	(3.8140)	30.45
	9	209.8556	(3.1790)	47.3568	(0.7820)	43.6185	**37.6313**	(1.0293)	39.0107	(1.2652)	37.7189	(1.1080)	37.8234	(1.1224)	37.7189	(1.1080)	17.41
	11	189.6122	(3.3210)	38.6134	(0.4178)	37.6441	33.5533	(0.6729)	34.9134	(0.8441)	33.4510	(0.7488)	33.7331	(0.7071)	**33.4432**	(0.7238)	13.87
	13	178.7385	(2.4013)	34.6598	(0.2013)	34.2323	32.0684	(0.1658)	33.4588	(0.4533)	**31.8801**	(0.2219)	32.3622	(0.2856)	**31.8801**	(0.2219)	14.58
	15	164.5634	(3.4339)	32.7890	(0.1576)	32.0429	31.2568	(0.1176)	32.5559	(0.3009)	31.1224	(0.1459)	31.5329	(0.1677)	**31.1129**	(0.1480)	11.63
	17	152.3630	(3.4950)	31.9520	(0.1213)	30.9684	30.8059	(0.0663)	32.0331	(0.2396)	30.6794	(0.1833)	31.0627	(0.1409)	**30.6693**	(0.1792)	9.75
	19	139.3465	(3.3326)	31.4199	(0.1094)	30.8265	30.4726	(0.0549)	30.9834	(0.1925)	30.2495	(0.0872)	30.6617	(0.1116)	**30.1882**	(0.0739)	9.28
	21	125.5484	(3.9635)	31.2403	(0.1318)	30.5572	30.5038	(0.0504)	30.5628	(0.0858)	30.2344	(0.0872)	30.4095	(0.0745)	**30.1800**	(0.0577)	7.93
	23	117.4370	(2.7649)	31.1276	(0.1308)	30.5833	30.5667	(0.0197)	30.1877	(0.0204)	30.4051	(0.0454)	30.1877	(0.0204)	**30.1277**	(0.0626)	7.60
	25	111.2900	(3.8243)	30.9873	(0.1180)	30.5988	30.5532	(0.0159)	**30.0734**	(0.0153)	30.3180	(0.0244)	**30.0734**	(0.0153)	**30.2519**	(0.0442)	7.66
01	5	243.3573	(3.7621)	117.5966	(1.5584)	61.7430	61.5998	(0.0383)	61.8315	(0.0266)	61.7541	(0.0243)	61.5504	(0.0266)	**61.5483**	(0.0253)	29.95
	7	220.4474	(4.8040)	68.1388	(1.1929)	54.1575	46.8109	(3.1034)	49.1187	(4.0574)	**46.7287**	(3.0898)	46.7797	(3.0559)	**46.7287**	(3.0898)	27.82
	9	205.5397	(5.2434)	47.1454	(0.9493)	43.8505	37.3567	(1.3534)	38.1964	(1.4197)	37.2152	(1.3771)	37.3256	(1.4143)	**37.1840**	(1.3506)	15.76
	11	184.5198	(3.9264)	38.4400	(0.2788)	37.0529	33.0900	(0.5789)	34.2452	(0.7699)	**32.8505**	(0.6438)	32.9577	(0.6947)	**32.8505**	(0.6438)	14.29
	13	176.1962	(2.7340)	34.3933	(0.2577)	33.8801	32.1088	(0.2319)	33.3487	(0.3999)	31.7263	(0.2687)	32.0866	(0.2293)	**31.7065**	(0.2810)	12.68
	15	161.3017	(3.7868)	32.5977	(0.1852)	32.2819	31.2063	(0.1675)	32.4660	(0.2317)	30.9376	(0.1694)	31.3579	(0.1240)	**30.9263**	(0.1791)	10.04
	17	146.1453	(5.3256)	31.7749	(0.1406)	31.1285	30.6560	(0.0971)	31.5468	(0.2635)	30.3862	(0.1104)	30.8162	(0.1015)	**30.2677**	(0.1876)	8.99
	19	138.7341	(4.3218)	31.3275	(0.1109)	30.3917	30.3080	(0.0667)	31.3020	(0.1623)	30.1054	(0.1207)	30.7151	(0.0964)	**30.0958**	(0.1432)	7.94
	21	127.9913	(4.5085)	31.0876	(0.1094)	30.3344	30.3339	(0.0019)	30.3568	(0.0000)	29.9955	(0.0030)	30.3568	(0.0000)	**29.9930**	(0.0106)	7.29
	23	121.6571	(2.5004)	30.8813	(0.1560)	30.4241	30.4239	(0.0006)	**29.8576**	(0.0028)	30.4145	(0.0107)	**29.8576**	(0.0028)	30.0385	(0.0079)	7.14
	25	112.7985	(3.7336)	30.8831	(0.1375)	30.5151	30.5146	(0.0024)	**29.7934**	(0.0029)	30.5434	(0.0003)	**29.7934**	(0.0029)	30.3881	(0.0099)	6.37
02	5	243.1300	(3.1597)	116.5611	(2.1222)	**62.4420**	**62.4420**	(0.0000)	62.7556	(0.0000)	**62.4420**	(0.0000)	62.7556	(0.0000)	**62.4420**	(0.0000)	24.49
	7	225.2825	(1.8583)	67.3661	(1.4986)	54.5530	48.2742	(3.6418)	50.3754	(4.5401)	47.3495	(3.1588)	48.2633	(3.6447)	**47.3346**	(3.1856)	28.12
	9	210.1822	(3.2451)	47.1804	(0.7465)	43.9374	36.1488	(1.3038)	37.1451	(1.4016)	36.0873	(1.3106)	36.2697	(1.2741)	**36.0173**	(1.2930)	18.57
	11	191.1158	(3.0048)	38.2710	(0.4864)	37.1124	33.0853	(0.4668)	34.1593	(0.6691)	33.0039	(0.4844)	33.2310	(0.5164)	**32.9922**	(0.4547)	13.54
	13	178.8800	(3.1969)	34.2332	(0.2254)	34.1587	31.9303	(0.2754)	33.2779	(0.3346)	31.8708	(0.2618)	32.1677	(0.2874)	**31.8529**	(0.2843)	12.18
	15	166.8335	(2.5609)	32.4176	(0.1511)	31.9071	31.0055	(0.2245)	32.1462	(0.4317)	30.8390	(0.3012)	31.1466	(0.3387)	**30.8224**	(0.3058)	9.59
	17	153.2093	(3.2867)	31.5436	(0.1497)	30.6525	30.5276	(0.1441)	31.6632	(0.3333)	30.4901	(0.2799)	30.6511	(0.2043)	**30.4415**	(0.3387)	7.50
	19	139.1990	(2.7681)	31.0371	(0.1483)	30.4898	30.1239	(0.0829)	30.5731	(0.3010)	29.7799	(0.1663)	30.0967	(0.1221)	**29.6545**	(0.1794)	8.84
	21	127.6935	(3.8689)	30.8056	(0.1304)	30.1175	30.0757	(0.0433)	30.3077	(0.1264)	29.7750	(0.0696)	29.9120	(0.0398)	**29.5529**	(0.1027)	7.14
	23	117.5319	(3.6122)	30.6848	(0.1367)	30.1787	30.0851	(0.0558)	29.7391	(0.0227)	29.8834	(0.0785)	**29.6956**	(0.0382)	29.7227	(0.0584)	8.05
	25	109.0172	(2.5038)	30.5773	(0.1447)	30.2627	30.2409	(0.0237)	**29.6312**	(0.0398)	29.8293	(0.1253)	**29.6312**	(0.0398)	29.7407	(0.0416)	7.84

Table 8.4: Objective values of solutions obtained from the same algorithms on instances as in Table 8.3, where the best solution is locally improved by a VND described in Section 7.2. For comparison the solution values of the ACO from Section 7.4 is given after one hour of computation.

Instance		with VND											
#	D	CBTC		RTC		C_bL	$C_{br}L$		$C_{br}d^AL$		$C_{br}d^BL$		ACO	
00	5	**59.6221**	(0.1103)	59.7765	(0.2529)	59.8156	59.8156	(0.0413)	59.9559	(0.0706)	59.8042	(0.0477)	*60.3577*	*(0.1344)*
	7	39.6436	(0.1476)	39.6902	(0.2243)	39.4605	39.5537	(0.2961)	39.5395	(0.2729)	**39.3744**	(0.2200)	*40.2425*	*(0.0978)*
	9	34.1601	(0.3505)	34.0715	(0.2738)	33.6128	**33.1894**	(0.0688)	33.3052	(0.1306)	33.2355	(0.1397)	*33.7396*	*(0.0743)*
	11	31.4880	(0.2351)	31.4607	(0.2071)	30.9243	**30.5135**	(0.3328)	30.5305	(0.2379)	**30.4532**	(0.2617)	*30.5041*	*(0.0958)*
	13	30.0161	(0.2483)	29.8353	(0.1813)	29.2811	**28.8274**	(0.0936)	28.8512	(0.1499)	28.8525	(0.1066)	*28.3057*	*(0.1535)*
	15	28.9688	(0.2162)	28.7828	(0.2254)	27.9198	**27.7925**	(0.0337)	27.9081	(0.0587)	27.8019	(0.0998)	*26.4572*	*(0.2241)*
	17	28.0892	(0.3004)	28.0250	(0.1900)	27.0307	**26.9613**	(0.1244)	27.2969	(0.0736)	27.0643	(0.1312)	*25.5560*	*(0.1846)*
	19	27.5062	(0.2292)	27.4700	(0.2336)	26.7333	**26.4315**	(0.0794)	26.5478	(0.0752)	26.5503	(0.0807)	*24.8623*	*(0.1072)*
	21	26.9498	(0.2618)	27.0683	(0.2508)	**26.0460**	26.1122	(0.0501)	26.1258	(0.1049)	26.2238	(0.0741)	*24.3701*	*(0.1381)*
	23	26.4851	(0.2468)	26.6122	(0.1916)	25.7810	**25.7797**	(0.0670)	26.1194	(0.0106)	25.8932	(0.0456)	*23.9870*	*(0.1130)*
	25	26.1102	(0.2321)	26.3004	(0.1746)	**25.4557**	25.5545	(0.0531)	25.6581	(0.0000)	25.5949	(0.0977)	*23.8765*	*(0.2161)*
01	5	58.8320	(0.2001)	58.8361	(0.2913)	58.4245	58.4472	(0.0195)	**58.4230**	(0.0204)	**58.4230**	(0.0176)	*59.2314*	*(0.1045)*
	7	39.1542	(0.2811)	39.2198	(0.2558)	**38.9405**	39.3357	(0.3697)	39.3378	(0.5176)	39.2348	(0.3976)	*39.6842*	*(0.1040)*
	9	33.7809	(0.2224)	33.7917	(0.2481)	33.1571	33.1902	(0.2114)	**33.1230**	(0.2475)	33.1725	(0.2291)	*33.3331*	*(0.0792)*
	11	31.3595	(0.2545)	31.2630	(0.2017)	31.2530	30.3054	(0.2073)	30.3464	(0.1912)	**30.2346**	(0.2537)	*30.0771*	*(0.0735)*
	13	29.7690	(0.2772)	29.7246	(0.2649)	29.2360	**28.6266**	(0.1064)	28.7196	(0.1093)	28.6844	(0.1193)	*27.9802*	*(0.1502)*
	15	28.7330	(0.2448)	28.6167	(0.2367)	27.9772	**27.6316**	(0.1526)	27.8079	(0.1587)	27.6724	(0.1265)	*26.4729*	*(0.2407)*
	17	27.9237	(0.2089)	27.8762	(0.1928)	26.8635	26.8116	(0.0337)	26.9191	(0.0746)	**26.7986**	(0.0806)	*25.3262*	*(0.1523)*
	19	27.3373	(0.2535)	27.2496	(0.2195)	**26.4955**	26.5193	(0.0395)	26.5098	(0.0756)	26.4994	(0.0840)	*24.6858*	*(0.1394)*
	21	26.8815	(0.1982)	26.8427	(0.2405)	**26.0522**	**26.0522**	(0.0324)	26.0732	(0.0677)	26.0630	(0.0419)	*24.3830*	*(0.1736)*
	23	26.3993	(0.1980)	26.4637	(0.1784)	25.7646	25.7673	(0.0194)	25.8666	(0.0308)	**25.6814**	(0.0307)	*24.0122*	*(0.1667)*
	25	25.9669	(0.2171)	26.1550	(0.1929)	25.3941	25.3941	(0.0126)	25.6193	(0.0028)	**25.1329**	(0.0179)	*23.8536*	*(0.1912)*
02	5	58.8960	(0.2258)	58.7338	(0.2237)	**58.4183**	**58.4183**	(0.1058)	58.7459	(0.0534)	**58.4183**	(0.0838)	*59.3599*	*(0.1225)*
	7	39.1793	(0.2901)	39.2197	(0.2176)	38.8909	**38.7746**	(0.2032)	39.0378	(0.2531)	38.9682	(0.3107)	*39.6961*	*(0.1395)*
	9	33.6654	(0.2527)	33.7463	(0.2096)	33.2359	**32.9355**	(0.1276)	32.9426	(0.1121)	32.9709	(0.1751)	*33.2441*	*(0.0844)*
	11	31.0565	(0.1837)	31.0403	(0.2450)	30.2728	**30.1227**	(0.2090)	30.1662	(0.2186)	30.1451	(0.2571)	*30.0009*	*(0.0741)*
	13	29.6195	(0.2974)	29.4133	(0.1850)	29.3309	**28.5560**	(0.1009)	28.6452	(0.1307)	28.5969	(0.1632)	*27.8937*	*(0.0909)*
	15	28.4502	(0.2337)	28.3968	(0.1873)	27.9103	27.6218	(0.0627)	27.5936	(0.0616)	**27.5169**	(0.1012)	*26.1176*	*(0.2001)*
	17	27.8170	(0.2331)	27.5829	(0.1663)	26.8776	**26.8526**	(0.0597)	26.9479	(0.1273)	26.8676	(0.1548)	*25.1324*	*(0.1353)*
	19	27.1231	(0.2543)	27.0401	(0.2294)	26.1847	26.2344	(0.0791)	26.2541	(0.0464)	**26.1395**	(0.0746)	*24.6174*	*(0.1618)*
	21	26.4635	(0.2664)	26.6310	(0.1449)	25.8567	25.8101	(0.0768)	25.8430	(0.0714)	**25.6117**	(0.0542)	*24.1249*	*(0.1686)*
	23	26.0805	(0.2628)	26.2041	(0.1511)	**25.3178**	25.3434	(0.0492)	25.3645	(0.1673)	25.3746	(0.0578)	*23.7830*	*(0.1696)*
	25	25.7151	(0.2142)	25.8258	(0.2432)	25.0308	25.1299	(0.0007)	25.4112	(0.0437)	**24.9809**	(0.0044)	*23.5875*	*(0.1734)*

Table 8.5: Averaged objective values over all 15 Euclidean Steiner tree instances of Beasley's OR-Library with 1000 nodes for various diameter bounds and (meta-)heuristics, the standard deviations are given parentheses. In addition, the averaged maximum running times of the clustering heuristics that were used as time limit for CBTC and RTC are listed, whereas the time limit for the ACO was set to one hour. All results are statistically significant due to paired Wilcoxon signed rank tests.

	without VND									with VND								
D	CBTC		RTC		$C_{br}d^AL$		$C_{br}d^BL$		t[s]	RTC		$C_{br}d^BL$		ACO		t[s]	
4	329.0261	(6.0233)	146.4919	(3.8841)	68.3241	(0.7152)	**68.3226**	(0.7004)	2.54	(0.09)	65.2061	(0.5478)	**65.1598**	(0.5571)	65.8010	(0.4779)	5.56	(1.01)
6	306.2655	(9.0246)	80.8636	(2.3991)	47.4045	(4.8519)	**47.1702**	(4.6086)	4.55	(0.49)	41.4577	(0.3559)	**41.3127**	(0.5041)	42.1167	(0.2623)	9.94	(1.52)
8	288.3842	(7.5165)	53.2535	(1.3275)	37.0706	(1.3539)	**36.9408**	(1.3361)	5.92	(0.42)	35.0511	(0.3457)	**34.2171**	(0.2930)	34.7489	(0.2347)	11.61	(1.61)
10	266.3665	(9.0090)	41.1201	(0.6795)	33.5460	(0.6665)	**33.3408**	(0.6643)	6.79	(0.42)	32.1181	(0.3089)	**30.9704**	(0.2420)	31.0388	(0.2012)	13.43	(2.16)
12	250.0016	(8.0149)	35.7590	(0.4661)	32.2571	(0.4675)	**31.9561**	(0.4423)	7.11	(0.33)	30.2897	(0.2908)	29.1796	(0.2550)	**28.6356**	(0.2299)	14.68	(2.49)
14	237.1403	(6.2757)	33.3644	(0.2990)	31.3790	(0.3740)	**31.0176**	(0.3278)	7.00	(0.64)	29.0940	(0.2838)	28.0093	(0.2314)	**26.6524**	(0.3169)	15.05	(3.00)
16	224.3123	(5.7232)	32.1965	(0.2409)	30.7937	(0.3275)	**30.4287**	(0.2879)	7.20	(0.72)	28.2433	(0.2759)	27.1363	(0.1950)	**25.5760**	(0.1949)	15.63	(2.89)
18	210.9872	(7.6322)	31.5826	(0.2410)	30.5182	(0.2884)	**30.1348**	(0.2714)	7.32	(0.81)	27.6008	(0.2659)	26.5601	(0.1988)	**24.8811**	(0.1637)	16.78	(3.61)
20	197.1772	(7.9852)	31.2682	(0.2212)	30.3116	(0.3056)	**30.0384**	(0.2810)	7.57	(0.76)	27.1091	(0.2622)	26.1079	(0.2289)	**24.3698**	(0.1523)	18.54	(3.89)
22	183.0157	(8.0299)	31.0864	(0.2217)	30.2344	(0.2970)	**30.0739**	(0.2814)	8.56	(0.98)	26.6984	(0.2779)	25.8048	(0.2125)	**24.0129**	(0.1750)	21.39	(5.19)
24	172.8251	(10.5944)	30.9921	(0.2280)	**30.0202**	(0.2280)	30.1603	(0.2735)	8.28	(1.41)	26.3648	(0.2699)	25.4523	(0.2408)	**23.7723**	(0.2004)	21.36	(6.42)
5	241.3032	(5.0912)	117.3238	(2.2237)	62.2867	(0.7563)	**62.0646**	(0.6743)	24.59	(2.02)	58.9883	(0.5269)	**58.7930**	(0.5620)	59.5964	(0.4882)	30.82	(3.28)
7	222.1441	(4.5006)	67.7577	(1.3133)	46.7291	(3.9205)	**46.4112**	(3.7329)	27.94	(1.79)	39.4703	(0.3355)	**39.3817**	(0.4613)	39.9948	(0.2469)	38.79	(4.03)
9	204.6141	(6.0033)	47.3168	(0.8482)	37.0224	(1.2484)	**36.8904**	(1.2696)	18.27	(1.68)	33.9677	(0.2980)	**33.2142**	(0.2457)	33.5907	(0.2327)	32.51	(4.88)
11	189.7513	(4.6215)	38.4754	(0.4970)	33.4140	(0.6952)	**33.1749**	(0.6629)	13.97	(0.71)	31.3661	(0.2898)	30.3683	(0.2018)	**30.2701**	(0.1946)	29.47	(4.70)
13	175.7382	(4.2250)	34.5154	(0.3235)	32.1094	(0.4250)	**31.8041**	(0.4131)	12.79	(1.17)	29.7644	(0.2760)	28.7554	(0.2115)	**28.1224**	(0.2049)	29.94	(6.28)
15	163.1926	(4.3107)	32.7069	(0.2458)	31.2654	(0.3490)	**30.8941**	(0.3244)	11.03	(1.27)	28.6966	(0.2567)	27.6899	(0.2046)	**26.3893**	(0.2473)	28.54	(6.29)
17	149.9852	(5.1365)	31.8467	(0.2302)	30.7699	(0.3264)	**30.3664**	(0.3047)	8.93	(0.94)	27.9309	(0.2663)	26.9097	(0.1946)	**25.3794**	(0.2297)	28.47	(6.19)
19	139.9730	(4.3211)	31.4048	(0.2090)	30.5350	(0.2922)	**30.0837**	(0.2659)	7.91	(1.08)	27.3691	(0.2618)	26.3784	(0.2011)	**24.7705**	(0.1792)	29.67	(7.37)
21	128.1830	(4.8954)	31.1697	(0.2290)	30.3017	(0.3030)	**30.0384**	(0.2735)	7.60	(0.71)	26.9015	(0.2598)	25.9415	(0.1972)	**24.3128**	(0.1835)	30.05	(6.74)
23	119.5551	(4.4550)	31.0421	(0.2227)	**30.0627**	(0.2403)	30.1166	(0.3139)	6.96	(0.81)	26.5346	(0.2727)	25.6021	(0.2101)	**23.9719**	(0.2111)	28.55	(7.05)
25	110.6725	(4.3891)	30.9772	(0.2315)	**29.9450**	(0.2145)	30.1393	(0.2427)	6.68	(0.89)	26.2126	(0.2607)	25.2289	(0.2146)	**23.7773**	(0.2518)	25.59	(6.02)

are the objective values, each with and without a *variable neighborhood descend* (VND) based on four different neighborhoods as presented in Section 7.2 applied to the best found solution, of the two construction heuristics CBTC and RTC, as well as the results for the clustering heuristic C at various stages: After binary search to determine good initial cutting positions through the dendrogram (b) and refinement of these cuts (r), where the actual trees have been derived by the simple greedy heuristic with additional local search (L), and when utilizing the two dynamic programming approaches (d^A with restricted search space, and d^B approximating optimal cluster centers using a correction value κ) to assign each cluster a good root node.

Binary search to identify a good value for x was performed within $\frac{|V|}{2}$ and $|V|$, except when $D < 6$. In this latter case the interval bounds have been set to $\frac{|V|}{20}$ and $\frac{|V|}{8}$. In GRASP a mean μ of 0 and, after preliminary tests, a variance σ^2 of 0.25 was used, and the procedure was aborted after $l_{\max} = 100$ iterations without improvement. The time (in seconds) listed is the maximum running time of $\mathrm{C}_{br}\mathrm{d}^A\mathrm{L}$ and $\mathrm{C}_{br}\mathrm{d}^B\mathrm{L}$, which was also used as time limit for CBTC and RTC. Furthermore, in the tables with applied VND the results for the leading metaheuristic in this field, the *ant colony optimization* (ACO) presented in Section 7.4, are given after one hour of computation.

Clearly, CBTC is not suited for this type of instances. Its strength are problems with random edge costs. The clustering heuristic outperforms RTC for every diameter bound, where the gap in solution quality is huge when D is small and becomes less with increasing diameter bound. Since both dynamic programming approaches derive no optimal trees for a given clustering, local improvement can further enhance their solutions, thus $\mathrm{C}_{br}\mathrm{d}^B\mathrm{L}$ builds in most cases the best diameter-constrained tree of the construction heuristics. It can also be seen that the runtime of the clustering heuristic only increases moderately with the number of levels in the height-restricted clustering. However, when applying the VND the differences between the construction heuristics flatten, but still the BDMSTs derived from clustering heuristic solutions are in general of higher quality. On instances with small diameter bounds these trees – computed in a few seconds – can also compete with results from the ACO with computation times of one hour and more.

Table 8.5 summarizes the observed results by listing for various diameter bounds the averaged objective values over all 15 Euclidean Steiner tree problem instances with 1000 nodes of Beasley's OR-Library. To verify statistical significance paired Wilcoxon signed rank tests have been performed. $\mathrm{C}_{br}\mathrm{d}^B\mathrm{L}$ outperforms all other construction heuristics significantly with an error probability of less than $2.2 \cdot 10^{-16}$. Only when the diameter bound gets noticeably loose the first dynamic program-

Table 8.6: Objective values and runtimes for two different versions of GRASP to refine initial cutting positions through the dendrogram in the odd diameter case (see Section 8.4 for details).

Instance		GRASP (based on even BDMSTs)					GRASP (only odd BDMSTs)				
#	D	RTC	$C_{br}L$	$C_{br}d^BL$	$C_{br}d^BLV$	t[s]	RTC	$C_{br}L$	$C_{br}d^BL$	$C_{br}d^BLV$	t[s]
00	5	117.3691 (2.1687)	62.5424 (0.0000)	62.4227 (0.0000)	59.8042 (0.0000)	24.38	**115.0603** (2.1320)	**62.0675** (0.0741)	**62.0673** (0.0735)	**59.7178** (0.0477)	81.29
	7	67.2259 (1.5432)	45.6928 (3.8075)	45.6409 (3.8140)	39.3744 (0.3555)	30.45	**65.9653** (1.3729)	**41.8482** (1.3121)	**41.7991** (1.3700)	**39.1763** (0.2200)	152.85
	9	47.3568 (0.7820)	37.6313 (1.0293)	37.7189 (1.1080)	33.2355 (0.1867)	17.41	**46.7370** (0.5608)	**36.4715** (1.4370)	**36.4858** (1.4561)	**33.1628** (0.1397)	51.30
	11	38.6134 (0.4178)	33.5533 (0.6729)	33.4432 (0.7238)	30.4532 (0.1101)	13.87	**38.2935** (0.3907)	**33.2897** (0.6087)	**33.1521** (0.6115)	**30.3725** (0.2617)	31.88
	13	34.6598 (0.2013)	32.0684 (0.1658)	**31.8801** (0.2219)	**28.8525** (0.1331)	14.58	**34.5474** (0.2605)	**32.0303** (0.1286)	31.9302 (0.1796)	28.8576 (0.1066)	24.23
	15	32.7890 (0.1576)	31.2568 (0.1176)	31.1129 (0.1480)	**27.8019** (0.1140)	11.63	**32.7771** (0.1478)	**31.1149** (0.0391)	**31.0939** (0.1308)	27.8829 (0.0998)	21.80
	17	31.9520 (0.1213)	30.8059 (0.0663)	30.6693 (0.1792)	**27.0643** (0.1104)	9.75	**31.8768** (0.1125)	**30.7512** (0.0810)	**30.4830** (0.1805)	27.1374 (0.1312)	14.74
	19	**31.4199** (0.1094)	30.4726 (0.0549)	30.1882 (0.0739)	**26.5503** (0.0794)	9.28	31.4581 (0.1320)	**30.4489** (0.0481)	**30.1720** (0.0780)	26.5668 (0.0807)	10.03
	21	31.2403 (0.1318)	30.5038 (0.0504)	**30.1800** (0.0577)	26.2238 (0.1325)	7.93	**31.2124** (0.1096)	**30.4612** (0.0264)	30.2092 (0.1002)	**26.1652** (0.0741)	9.23
	23	31.1276 (0.1308)	30.5667 (0.0197)	**30.1277** (0.0626)	**25.8932** (0.0398)	7.60	**31.1043** (0.1065)	**30.4943** (0.0180)	30.3655 (0.0141)	25.9301 (0.0456)	7.13
	25	**30.9873** (0.1180)	30.5532 (0.0159)	**30.2519** (0.0442)	25.5949 (0.0423)	7.66	31.0492 (0.1172)	**30.4986** (0.0209)	30.2633 (0.0117)	**25.5366** (0.0977)	7.49
01	5	117.5966 (1.5584)	61.5998 (0.0383)	**61.5483** (0.0253)	**58.4230** (0.0004)	29.95	**115.0932** (2.1735)	**61.5091** (0.0021)	61.5586 (0.0315)	59.1128 (0.0176)	96.89
	7	68.1388 (1.1929)	46.8109 (3.1034)	46.7287 (3.0898)	**39.2348** (0.2341)	27.82	**65.9784** (1.1417)	**44.9667** (3.3749)	**45.0474** (3.4465)	39.2726 (0.3976)	123.74
	9	47.1454 (0.9493)	37.3567 (1.3534)	37.1840 (1.3506)	**33.1725** (0.1619)	15.76	**46.6096** (0.6025)	**36.0680** (0.9435)	**35.8380** (1.0060)	33.3254 (0.2291)	55.27
	11	38.4400 (0.2788)	33.0900 (0.5789)	32.8505 (0.6438)	**30.2346** (0.1627)	14.29	**37.9413** (0.4751)	**33.0719** (0.6094)	**32.8458** (0.6219)	30.2470 (0.2537)	34.58
	13	34.3933 (0.2577)	32.1088 (0.2319)	31.7065 (0.2810)	28.6844 (0.1284)	12.68	**34.2003** (0.3087)	**31.8353** (0.2689)	**31.5111** (0.3210)	**28.6077** (0.1193)	31.77
	15	32.5977 (0.1852)	31.2063 (0.1675)	30.9263 (0.1791)	27.6724 (0.1502)	10.04	**32.5164** (0.1617)	**31.1512** (0.1594)	**30.8229** (0.1880)	**27.6577** (0.1265)	18.21
	17	31.7749 (0.1406)	30.6560 (0.0971)	**30.2677** (0.1876)	**26.7986** (0.1064)	8.99	**31.6723** (0.1275)	**30.6329** (0.0968)	30.3059 (0.1489)	26.8031 (0.0806)	12.67
	19	31.3275 (0.1109)	30.3080 (0.0667)	**30.0958** (0.1432)	**26.4994** (0.0996)	7.94	**31.3165** (0.1373)	**30.2865** (0.0658)	30.1015 (0.1100)	26.5066 (0.0840)	7.01
	21	**31.0876** (0.1094)	30.3339 (0.0019)	29.9930 (0.0106)	26.0630 (0.0045)	7.29	31.1366 (0.1379)	**30.3145** (0.0216)	**29.9866** (0.0529)	**26.0620** (0.0419)	5.69
	23	**30.8813** (0.1560)	30.4239 (0.0006)	30.0385 (0.0079)	25.6814 (0.0107)	7.14	30.9327 (0.1654)	**30.4208** (0.0038)	**30.0190** (0.0121)	**25.6549** (0.0307)	5.32
	25	**30.8831** (0.1375)	30.5146 (0.0024)	**30.3881** (0.0099)	**25.1329** (0.0071)	6.37	30.8937 (0.1435)	**30.4967** (0.0093)	30.4331 (0.0245)	25.1675 (0.0179)	5.19
02	5	116.5611 (2.1222)	62.4420 (0.0000)	62.4420 (0.0000)	58.4183 (0.0000)	24.49	**114.6258** (1.8058)	**62.1294** (0.0080)	**62.1334** (0.0227)	**58.3889** (0.0838)	97.18
	7	67.3661 (1.4986)	48.2742 (3.6418)	47.3346 (3.1856)	38.9682 (0.2020)	28.12	**65.9766** (1.1568)	**45.0743** (4.1764)	**44.6519** (3.7817)	**38.9060** (0.3107)	110.12
	9	47.1804 (0.7465)	36.1488 (1.3038)	36.0173 (1.2930)	32.9709 (0.2365)	18.57	**46.5164** (0.7095)	**35.6095** (1.0858)	**35.3966** (1.0097)	**32.8876** (0.1751)	49.32
	11	38.2710 (0.4864)	33.0853 (0.4668)	32.9922 (0.4547)	**30.1451** (0.1226)	13.54	**37.9126** (0.4763)	**32.9404** (0.4897)	**32.8667** (0.5506)	30.1769 (0.2571)	31.78
	13	34.2332 (0.2254)	31.9303 (0.2754)	31.8529 (0.2843)	28.5969 (0.1753)	12.18	**34.1001** (0.2260)	**31.6198** (0.2464)	**31.5702** (0.3049)	**28.4633** (0.1632)	20.83
	15	32.4176 (0.1511)	31.0055 (0.2245)	30.8224 (0.3058)	27.5169 (0.0895)	9.59	**32.3443** (0.1604)	**30.7569** (0.1402)	**30.6213** (0.2016)	**27.3341** (0.1012)	21.26
	17	31.5436 (0.1497)	30.5276 (0.1441)	30.4415 (0.3387)	26.8676 (0.0816)	7.50	**31.4812** (0.1215)	**30.3853** (0.1211)	**30.1961** (0.3073)	**26.7540** (0.1548)	11.77
	19	**31.0371** (0.1483)	30.1239 (0.0829)	29.6545 (0.1794)	26.1395 (0.1227)	8.84	31.0742 (0.0890)	**30.1096** (0.0613)	**29.6050** (0.1276)	**26.1057** (0.0746)	9.04
	21	**30.8056** (0.1304)	30.0757 (0.0433)	**29.5529** (0.1027)	**25.6117** (0.0721)	7.14	30.8845 (0.1158)	**30.0601** (0.0335)	29.6016 (0.1152)	25.6669 (0.0542)	6.25
	23	30.6848 (0.1367)	**30.0851** (0.0558)	29.7227 (0.0584)	25.3746 (0.0370)	8.05	**30.6805** (0.1267)	30.0892 (0.0552)	**29.7079** (0.0732)	**25.3524** (0.0578)	6.26
	25	**30.5773** (0.1447)	30.2409 (0.0237)	**29.7407** (0.0416)	24.9809 (0.0654)	7.84	30.6665 (0.1282)	**30.2018** (0.0202)	29.7607 (0.0088)	**24.9281** (0.0044)	5.54

ming approach $C_{br}d^{A}L$ dominates $C_{br}d^{B}L$ (error probability always less than $2.13 \cdot 10^{-9}$).

In Tables 8.3, 8.4, and 8.5, a version of GRASP was used to speed-up the computation in the odd-diameter case where a tree with center edge was only derived from the actual height-restricted clustering if the corresponding even-diameter tree leads to the so far best solution, cf. Section 8.4. Table 8.6 now compares this approach to one where always an odd-diameter tree is used within the cut refinement step of the clustering heuristic. In this table the better of the two corresponding result entries is printed bold, the application of the VND is marked with V.

Especially if the diameter bound is tight, the runtime increases significantly when only using odd-diameter trees, however, also the solution quality of the clustering heuristic is in general higher in this case. Of course, RTC benefits from the enlarged time limit, too. Interesting is the fact that with increasing diameter bound the running times approach each other, and near a diameter of 21 the previously slower implementation even becomes fastest. There are two different reasons for this behavior. First, the number of potential center edges to be considered, determined in a preprocessing step, decreases with rising diameter bound, so computing an odd-diameter tree from a height-restricted clustering no longer requires significantly more time. Additionally, when directly using odd trees in the cut refinement step good solutions are found earlier; thus GRASP can be aborted after less iterations.

Finally, Tables 8.7 and 8.8 summarize results when using the clustering-based neighborhood in the VND from Section 7.2. In general, the ordering of the neighborhoods within the VND (V) is clear: First, whole subtrees are moved within the solution (arc exchange (a)), and afterwards, the arrangement of nodes and their direct successors is considered (node swap (n)). Then the usually more time consuming level based neighborhoods are applied: The best center with respect to the center exchange level neighborhood (c) is determined, and afterwards the levels of all non-center nodes are refined by means of the level change neighborhood (l). Since the clustering-based neighborhood (k) already requires a solution of some quality, and due to its high computational complexity (BDMSTs are derived using $C_{br}d^{B}L$), we performed experiments with the clustering neighborhood being executed before and after the two level neighborhoods (V^{ankcl} and V^{anclk}). The initial solutions are computed using RTC, whereas new trees are constructed as long as no better one could be found within the last 100 iterations; the best BDMST is returned as input for the different VNDs.

One conclusion is obvious: The application of the clustering neighborhood can improve the solution quality, but only in case the diameter bound D is less than (roughly) 14 on instances with 1000 nodes. Otherwise, the VND with the four standard neighborhoods is able to reach the same solution quality, and this typically in

Table 8.7: Objective values and runtimes for RTC solutions improved by VNDs (V) with different neighborhoods and neighborhood orders: arc exchange (a), node swap (n), center exchange (c), level (l), and cluster (k) neighborhood. Results for the even diameter case.

Instance		Objective Values								Running Times [s]							
#	D	RTC		V^{ancl}		V^{anclk}		V^{ankcl}		RTC		V^{ancl}		V^{anclk}		V^{ankcl}	
00	4	161.8783	(7.7021)	**65.9976**	(0.2532)	**65.9976**	(0.2532)	66.0038	(0.2374)	0.12	(0.0420)	6.39	(1.1476)	6.47	(1.1397)	6.32	(0.8977)
	6	89.6792	(4.0248)	41.7216	(0.2562)	**41.3942**	(0.3092)	41.4360	(0.2746)	0.21	(0.0607)	9.89	(1.2356)	13.27	(1.6011)	9.86	(1.4232)
	8	57.4663	(2.2666)	35.1672	(0.2996)	**34.7193**	(0.2321)	34.8042	(0.2760)	0.49	(0.1703)	12.17	(2.5145)	16.87	(2.8460)	11.08	(1.5076)
	10	42.9830	(1.1588)	32.2816	(0.2600)	**31.8786**	(0.3445)	31.9141	(0.3297)	0.94	(0.3361)	10.36	(2.7240)	13.02	(2.8968)	9.75	(2.8075)
	12	36.6975	(0.5287)	30.5246	(0.2490)	30.5049	(0.2529)	**30.3217**	(0.3627)	1.39	(0.5321)	8.16	(2.7625)	8.85	(3.1728)	8.28	(2.7584)
	14	33.9374	(0.2911)	**29.3455**	(0.2765)	**29.3455**	(0.2765)	**29.3455**	(0.2765)	1.75	(0.8572)	8.77	(3.0398)	8.85	(3.0625)	8.92	(3.0680)
	16	32.4497	(0.1821)	**28.4259**	(0.1911)	**28.4259**	(0.1911)	**28.4259**	(0.1911)	2.74	(0.8054)	6.75	(2.0797)	6.72	(2.0321)	7.02	(2.2064)
	18	31.8538	(0.1575)	**27.8196**	(0.2089)	**27.8196**	(0.2089)	**27.8196**	(0.2089)	2.55	(1.1343)	7.08	(2.2775)	6.97	(2.0670)	7.32	(2.2605)
	20	31.5229	(0.1123)	**27.2506**	(0.1961)	**27.2506**	(0.1961)	**27.2506**	(0.1961)	2.30	(0.8107)	7.89	(2.0367)	8.05	(2.4166)	8.20	(2.4373)
	22	31.3410	(0.0966)	**26.8635**	(0.2148)	**26.8635**	(0.2148)	**26.8635**	(0.2148)	2.71	(1.1098)	7.83	(2.5021)	7.81	(2.4104)	7.96	(2.4533)
	24	31.2421	(0.0955)	**26.5722**	(0.2371)	**26.5722**	(0.2371)	**26.5722**	(0.2371)	3.14	(1.0602)	6.70	(1.6292)	6.79	(1.6335)	6.91	(1.6746)
01	4	160.2108	(6.6749)	**64.6837**	(0.1907)	**64.6837**	(0.1907)	64.6871	(0.2058)	0.11	(0.0467)	6.41	(1.1866)	6.49	(1.1994)	6.34	(0.8594)
	6	87.1181	(4.7997)	41.1626	(0.2847)	**40.8945**	(0.2654)	40.9707	(0.2605)	0.23	(0.0729)	10.42	(1.8288)	13.52	(2.0571)	9.71	(1.0851)
	8	56.5508	(2.2608)	34.8626	(0.3786)	34.3725	(0.3231)	**34.3246**	(0.3642)	0.53	(0.1925)	11.98	(1.8597)	17.27	(2.8641)	10.78	(1.9101)
	10	43.0218	(0.9382)	32.0837	(0.2855)	**31.5869**	(0.3195)	31.6313	(0.3340)	0.99	(0.3593)	12.72	(3.1296)	17.67	(4.2997)	12.11	(3.3447)
	12	36.3884	(0.6051)	30.2910	(0.2407)	30.2311	(0.3345)	**30.1697**	(0.3300)	1.37	(0.4601)	10.18	(3.4561)	10.63	(3.8704)	10.14	(3.1478)
	14	33.6087	(0.3492)	**29.1399**	(0.2460)	**29.1399**	(0.2460)	**29.1399**	(0.2460)	1.87	(0.7503)	7.30	(2.6435)	7.43	(2.3950)	7.60	(2.3278)
	16	32.2474	(0.1823)	**28.2011**	(0.2295)	**28.2011**	(0.2295)	**28.2011**	(0.2295)	2.36	(1.0345)	7.16	(2.7799)	7.30	(3.0455)	7.53	(3.1344)
	18	31.6125	(0.1462)	**27.5964**	(0.2116)	**27.5964**	(0.2116)	**27.5964**	(0.2116)	2.55	(1.0174)	7.92	(2.6730)	7.89	(2.7209)	7.97	(2.7039)
	20	31.3158	(0.1275)	**27.0567**	(0.2680)	**27.0567**	(0.2680)	**27.0567**	(0.2680)	2.56	(0.9852)	8.48	(2.8963)	8.60	(3.1304)	8.80	(3.2607)
	22	31.1534	(0.1138)	**26.6877**	(0.2761)	**26.6877**	(0.2761)	**26.6877**	(0.2761)	2.56	(1.0313)	8.77	(3.3308)	8.87	(3.3486)	9.03	(3.4232)
	24	31.0694	(0.1041)	**26.4029**	(0.2776)	**26.4029**	(0.2776)	**26.4029**	(0.2776)	2.35	(0.6786)	7.71	(3.8639)	7.80	(3.8730)	7.93	(3.9411)
02	4	159.8350	(6.8385)	64.7563	(0.2695)	64.7509	(0.2741)	**64.7467**	(0.2826)	0.12	(0.0387)	6.45	(0.9301)	6.54	(0.9526)	6.34	(0.8144)
	6	90.7577	(3.7117)	41.2524	(0.2548)	40.9858	(0.2448)	**40.9321**	(0.2581)	0.25	(0.0866)	9.85	(1.2250)	12.79	(1.7230)	10.07	(1.1066)
	8	57.3699	(1.7375)	34.7815	(0.3493)	**34.3044**	(0.3319)	34.3536	(0.3844)	0.46	(0.1481)	11.83	(2.0127)	17.20	(2.9043)	10.92	(2.1376)
	10	42.7030	(1.0421)	31.8751	(0.2913)	**31.4591**	(0.2936)	31.5864	(0.3410)	0.81	(0.3673)	11.64	(2.1958)	14.73	(3.0356)	9.82	(1.8300)
	12	36.1223	(0.6098)	30.1287	(0.3228)	30.0212	(0.3862)	**29.9597**	(0.3444)	1.44	(0.4987)	8.54	(2.3660)	9.13	(3.1152)	8.48	(2.0374)
	14	33.2897	(0.3560)	28.8895	(0.2284)	28.8895	(0.2284)	**28.8550**	(0.2598)	2.04	(0.9013)	7.17	(1.7356)	7.25	(1.7475)	7.29	(1.7838)
	16	32.0007	(0.2255)	**27.9773**	(0.2202)	**27.9773**	(0.2202)	**27.9773**	(0.2202)	2.01	(0.5390)	6.63	(2.0342)	6.63	(2.0343)	6.82	(2.0856)
	18	31.3814	(0.2092)	**27.3952**	(0.1970)	**27.3952**	(0.1970)	**27.3952**	(0.1970)	2.17	(0.6694)	7.35	(2.0279)	7.44	(2.0334)	7.58	(2.0859)
	20	31.0562	(0.1500)	**26.8255**	(0.2003)	**26.8255**	(0.2003)	**26.8255**	(0.2003)	2.42	(0.7805)	6.78	(2.7186)	6.85	(2.7235)	6.99	(2.7852)
	22	30.8812	(0.1294)	**26.4152**	(0.2358)	**26.4152**	(0.2358)	**26.4152**	(0.2358)	2.55	(0.6151)	6.74	(2.5323)	6.82	(2.5366)	6.94	(2.5887)
	24	30.7914	(0.1507)	**26.1095**	(0.1946)	**26.1095**	(0.1946)	**26.1095**	(0.1946)	2.62	(0.8620)	7.32	(2.7724)	7.37	(2.7697)	7.48	(2.8288)

Table 8.8: Objective values and runtimes for RTC solutions improved by VNDs (V) with different neighborhoods and neighborhood orders: arc exchange (a), node swap (n), center exchange (c), level (l), and cluster (k) neighborhood. Results for the odd diameter case.

Instance		Objective Values								Running Times [s]							
		RTC		V^{ancl}		V^{anclk}		V^{ankcl}		RTC		V^{ancl}		V^{anclk}		V^{ankcl}	
#	D																
00	5	132.6017	(4.5827)	59.8303	(0.2386)	59.7996	(0.2399)	**59.7543**	(0.1924)	0.14	(0.0496)	12.41	(2.1681)	32.62	(8.3574)	139.69	(31.3942)
	7	74.2697	(2.5960)	39.7102	(0.2506)	39.3997	(0.2391)	**39.3925**	(0.2899)	0.32	(0.1046)	20.78	(3.2322)	53.66	(9.3268)	142.98	(26.2220)
	9	50.5489	(1.3972)	34.2026	(0.2169)	33.7336	(0.2147)	**33.6804**	(0.2512)	0.68	(0.2018)	20.56	(3.9809)	44.35	(6.3030)	87.85	(19.5879)
	11	39.9851	(0.8436)	31.4815	(0.2271)	**31.1696**	(0.3413)	31.1751	(0.2139)	1.15	(0.4205)	19.49	(4.6865)	33.41	(6.6998)	46.56	(15.6605)
	13	35.2966	(0.4075)	29.9683	(0.2638)	29.9683	(0.2638)	**29.9072**	(0.2729)	1.60	(0.5998)	17.74	(6.3709)	23.42	(6.3915)	44.69	(15.0085)
	15	33.0914	(0.2528)	**28.8962**	(0.2024)	**28.8962**	(0.2024)	**28.8962**	(0.2024)	2.06	(0.6189)	12.82	(3.9334)	17.63	(4.2317)	29.99	(11.6301)
	17	32.1307	(0.1828)	**28.1730**	(0.2012)	**28.1730**	(0.2012)	**28.1730**	(0.2012)	2.23	(0.7387)	15.53	(5.2475)	19.50	(5.8640)	31.27	(10.2896)
	19	31.6462	(0.1490)	**27.4792**	(0.2050)	**27.4792**	(0.2050)	**27.4792**	(0.2050)	2.58	(0.9212)	16.32	(6.1735)	19.85	(6.8190)	29.39	(11.7954)
	21	31.3840	(0.1628)	**27.0608**	(0.2209)	**27.0608**	(0.2209)	**27.0608**	(0.2209)	2.17	(0.8354)	14.29	(5.1124)	16.50	(4.8330)	24.04	(9.0761)
	23	31.1936	(0.1524)	**26.6370**	(0.2061)	**26.6370**	(0.2061)	**26.6370**	(0.2061)	2.82	(0.8530)	19.58	(8.0326)	22.08	(7.9138)	29.66	(10.6472)
	25	31.1080	(0.1523)	**26.3423**	(0.2287)	**26.3423**	(0.2287)	**26.3423**	(0.2287)	3.11	(1.0837)	14.32	(5.3013)	16.25	(5.3269)	20.62	(7.2469)
01	5	130.8494	(4.7108)	58.8675	(0.3998)	58.8503	(0.4005)	**58.8025**	(0.3927)	0.14	(0.0516)	12.81	(2.9093)	33.48	(8.9670)	141.79	(31.6241)
	7	74.2996	(2.5268)	39.2226	(0.2535)	**38.8871**	(0.2425)	38.9195	(0.2138)	0.35	(0.0935)	22.33	(4.9935)	60.58	(10.8401)	179.54	(44.8222)
	9	49.7791	(1.4454)	33.7868	(0.2304)	**33.3282**	(0.1944)	33.3965	(0.2700)	0.73	(0.2628)	22.47	(5.1301)	48.71	(9.4672)	86.16	(17.9768)
	11	39.7409	(0.6931)	31.3436	(0.2837)	**30.9745**	(0.4081)	31.0046	(0.3652)	1.13	(0.2893)	22.87	(5.9728)	37.64	(8.3419)	48.58	(12.3373)
	13	35.0445	(0.4010)	29.8522	(0.2012)	29.8522	(0.2012)	**29.8263**	(0.2236)	1.44	(0.4088)	18.29	(5.9684)	22.61	(6.0956)	40.84	(11.9887)
	15	32.9889	(0.2712)	**28.7303**	(0.1849)	**28.7303**	(0.1849)	**28.7303**	(0.1849)	2.05	(0.6516)	16.15	(6.2645)	20.94	(6.9014)	36.73	(14.3601)
	17	32.0092	(0.1572)	**27.9137**	(0.1839)	**27.9137**	(0.1839)	**27.9137**	(0.1839)	2.36	(0.7946)	15.97	(5.8454)	19.60	(5.5265)	32.30	(12.4138)
	19	31.5166	(0.1689)	**27.3046**	(0.2555)	**27.3046**	(0.2555)	**27.3046**	(0.2555)	2.38	(1.0861)	14.13	(4.2241)	16.93	(4.5415)	25.35	(9.4874)
	21	31.2039	(0.1640)	**26.8179**	(0.2614)	**26.8179**	(0.2614)	**26.8179**	(0.2614)	2.36	(0.7271)	18.22	(5.4003)	20.46	(5.3809)	27.89	(8.6915)
	23	31.0512	(0.1406)	**26.4495**	(0.2139)	**26.4495**	(0.2139)	**26.4495**	(0.2139)	2.41	(0.9568)	13.59	(4.7919)	15.49	(4.8347)	20.37	(7.4603)
	25	30.9905	(0.1483)	**26.1427**	(0.1972)	**26.1427**	(0.1972)	**26.1427**	(0.1972)	2.33	(0.7701)	15.01	(6.0368)	16.82	(5.8542)	20.86	(6.8823)
02	5	131.3578	(4.2625)	58.7303	(0.2589)	58.7069	(0.2557)	**58.7067**	(0.2383)	0.12	(0.0334)	12.38	(2.7238)	32.82	(9.0611)	152.80	(39.3996)
	7	74.1247	(3.1031)	39.1472	(0.2447)	**38.8581**	(0.2559)	38.8831	(0.2949)	0.34	(0.1192)	20.29	(3.1990)	51.01	(8.8465)	163.57	(41.6844)
	9	50.2263	(2.0036)	33.7039	(0.2716)	33.2137	(0.2986)	**33.2035**	(0.2496)	0.67	(0.2194)	20.98	(4.3867)	46.12	(8.0497)	94.49	(24.7316)
	11	39.3963	(0.8967)	31.1349	(0.2561)	30.7978	(0.3835)	**30.7735**	(0.2955)	1.22	(0.4649)	21.12	(5.3085)	34.44	(9.9576)	49.40	(12.6640)
	13	34.8063	(0.3678)	29.5023	(0.1467)	29.4584	(0.1795)	**29.3826**	(0.2473)	1.98	(0.5853)	15.89	(5.6456)	21.32	(6.2728)	38.55	(15.0540)
	15	32.7143	(0.2359)	**28.4464**	(0.2615)	**28.4464**	(0.2615)	**28.4464**	(0.2615)	1.97	(0.7709)	14.25	(4.1149)	18.58	(4.1961)	33.84	(10.2553)
	17	31.7155	(0.1834)	**27.6687**	(0.2027)	**27.6687**	(0.2027)	**27.6687**	(0.2027)	2.35	(1.0918)	13.15	(5.6503)	16.70	(5.5626)	26.79	(9.7479)
	19	31.2432	(0.1137)	**27.1137**	(0.1836)	**27.1137**	(0.1836)	**27.1137**	(0.1836)	2.36	(0.7516)	13.51	(3.9701)	16.35	(3.9731)	24.23	(7.2155)
	21	30.9612	(0.1285)	**26.6156**	(0.2307)	**26.6156**	(0.2307)	**26.6156**	(0.2307)	2.52	(0.7635)	14.53	(4.5662)	16.90	(4.7514)	23.51	(7.5798)
	23	30.8259	(0.1359)	**26.2015**	(0.2268)	**26.2015**	(0.2268)	**26.2015**	(0.2268)	2.51	(0.9871)	15.90	(5.7522)	17.98	(5.7360)	24.01	(8.2103)
	25	30.7651	(0.1364)	**25.8826**	(0.2378)	**25.8826**	(0.2378)	**25.8826**	(0.2378)	2.47	(0.8915)	15.42	(6.8132)	17.20	(6.6964)	21.51	(8.5251)

less time. When to execute the clustering neighborhood, before or after the level neighborhoods, is not that clear and depends mostly on the diameter: If D is even the clustering neighborhood should be the last one, whereas in case the diameter is odd it seems to be more promising to apply it before the center exchange level neighborhood. The bad runtime behavior for small odd diameters already discussed for the clustering-based construction heuristic can, of course, be observed in this context, too.

8.7 Conclusions and Future Work

On the more difficult to solve Euclidean BDMST instances fast construction heuristics proposed so far fail to compute a good backbone consisting of few but long edges to allow the majority of the nodes to connect to the tree via relatively short edges. In this work we presented a constructive heuristic that exploits a hierarchical clustering to guide the process of building a backbone. The clustering heuristic constructs diameter-constrained trees within three steps: building a hierarchical clustering, reducing the height of this clustering according to the diameter bound, and finally deriving a BDMST from this height-restricted clustering. Various techniques are used within the individual phases like GRASP to refine cutting positions through the dendrogram, or dynamic programming to assign each cluster a good root node.

In particular on large Euclidean instances with more than 500 nodes the BDMSTs obtained by the clustering heuristic are in general of high quality and outperform the other construction heuristics significantly, especially when the diameter bound is tight. When using a strong VND to further improve these solutions they can also compete with results from an ACO, currently the leading metaheuristic for this problem. The computation of our heuristic followed by VND, however, requires only a few seconds in comparison to one hour and more per run for the ACO.

The negative effects when strictly following the clustering as discussed in Section 8.2.4 may be further addressed in two different ways. One simple approach would be to let the clustering-based construction heuristic build only the first part (near the center of the BDMST) of the backbone and to use a Prim based algorithm (CBTC or RTC) for the remaining nodes. A more sophisticated version would allow a root u of a sub-cluster not only to connect to the root of its direct parent cluster v but to any node of the already built backbone on the path from the center of the BDMST to u. In case a cheaper connection is possible than (u, v) some clusters merged in the subtree rooted at u for the height-restricted clustering can again be split since now u is connected at least one edge closer to the center of the BDMST, and so this subtree would otherwise not fully exploit the available height.

Chapter 9

Conclusions

The bounded diameter minimum spanning tree problem is an $\mathcal{NP}$-hard combinatorial optimization problem in the area of network design when quality of services is a major concern, i.e., when the number of hops between any two communication partners in the network should be limited since they potentially introduce delays or noise. However, it also appears as a subproblem in other fields like data compression or distributed mutual exclusion algorithms.

In this thesis, a wide variety of different methods has been considered to deal with the BDMST problem: Integer linear programming embedded within Branch&Cut to solve moderate-sized instances to proven optimality, metaheuristics to handle large problem instances of several hundreds of nodes obtaining high quality solutions, and a new fast construction heuristic for particularly large Euclidean instances. Almost all of these approaches are hybrids in the sense that they make further use of other embedded exact and heuristic techniques to solve subordinate problems arising. These hybrid algorithms demonstrated their effectiveness in comparison to state-of-the-art approaches from the literature.

Five different neighborhood structures for the BDMST problem have been defined to locally improve solutions. They operate on different solution representations, namely the tree structure itself, the levels the nodes appear in within the tree, and a hierarchical clustering of all nodes. Special attention has been paid to an efficient implementation of the search procedures for the various neighborhoods. Not only the objective value of a solution is evaluated incrementally when an improvement move is executed but also neighboring solutions whenever possible: All improvement moves are stored within a cache and only those moves affected by executing the move

with the biggest gain are reevaluated. When following a best improvement strategy to search a neighborhood to a local optimum this approach reduces the required computational effort significantly.

Two different exact ILP formulations embedded within a Branch&Cut environment have been proposed to solve moderate-sized BDMST instances to proven optimality. Compared to the highly successful hop-index multi-commodity flow formulations the level-based ILP model is very compact and can be strengthened by additional cutting planes. Although giving comparable results on small instances, the relatively weak LP bounds of this formulation prohibit its application on larger ones. The jump model, which further reduced the number of required variables, makes use of so-called jump inequalities to ensure the diameter bound in a solution. Since the number of jump inequalities grows exponentially with the size of the problem instance and the problem of separating them within an LP solution is conjectured to be $\mathcal{NP}$-hard a hierarchy of heuristics including a tabu search metaheuristic is used to efficiently identify a high percentage of violated constraints. This approach leads to an excellent overall performance; it was able to discover so long unknown optima for various instances, and especially when the diameter bound is loose it is superior to the so far leading flow formulations according to runtime and memory usage.

For larger instances different metaheuristics have been developed, namely a VNS, an EA, and an ACO, making use of the local search neighborhoods defined for the BDMST problem. In contrast to other state-of-the-art metaheuristics this new EA and the very successful ACO operate on a special solutions representation optimizing not directly the connections within the tree but the levels the nodes appear in the final diameter-constrained tree. Till now, the ACO is the best choice to get solutions of high quality within reasonable time on complete instances with some hundreds of nodes.

Finally, for especially large and hard to solve Euclidean instances a new construction heuristic has been introduced. It is based on hierarchical clustering to guide the construction process and makes use of a multitude of heuristics to improve and refine the final BDMST. It outperforms standard heuristics from the literature significantly, and for tight diameter bounds, locally improved trees computed with this construction heuristic can also compete with solutions of the above mentioned ACO, although requiring only a fraction of time.

Advances of Gouveia et al. [66] in formulating the BDMST as a Steiner tree problem on layered graphs are encouraging to noticeably increase the size of instances that can be solved to optimality. The progress of exact approaches also allows to explore new opportunities in combining them with new and advanced (meta-)heuristics. Beside the definition of new neighborhoods that can be searched efficiently by exact algorithms, also the cooperative execution of different solving procedures in parallel

to benefit from synergy may lead to a significant boost of performance. Last but not least, the clustering-based construction heuristic still has potential to achieve better results when making the assignment of nodes to clusters more flexible, as well as heuristics based on Kruskal's instead of Prim's MST algorithm used with success for the related delay-constrained MST problem in recent time. Altogether, these are some interesting research challenges for the near future.

Bibliography

[1] A. Abdalla, N. Deo, and P. Gupta. Random-tree diameter and the diameter constrained MST. *Congressus Numerantium*, 144:161–182, 2000.

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

[3] N. R. Achuthan and L. Caccetta. Minimum weight spanning trees with bounded diameter. *Australasian Journal of Combinatorics*, 5:261–276, 1992.

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

[5] C. Aggarwal, J. Orlin, and R. Tai. Optimized crossover for the independent set problem. *Operations Research*, 45:226–234, 1997.

[6] O. Angel, A. D. Flaxman, and D. B. Wilson. A sharp threshold for minimum bounded-depth and bounded-diameter spanning trees and Steiner trees in random networks. arXiv:0810.4908v1 [math.PR] `http://arxiv.org/`, 2008.

[7] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook. On the solution of the traveling salesman problem. *Documenta Mathematica*, Extra Volume ICM III:645–656, 1998.

[8] P. Augerat, J. Belenguer, E. Benavent, A. Corberan, and D. Naddef. Separating capacity constraints in the CVRP using tabu search. *European Journal of Operational Research*, 106(2):546–557, 1999.

[9] T. Bäck, D. B. Fogel, and Z. Michalewicz. *Handbook of Evolutionary Computation*. Oxford University Press, New York, 1997.

[10] J. E. Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, pages 14–21, Hillsdale, NJ, USA, 1987. Lawrence Erlbaum Associates, Inc.

[11] K. Bala, K. Petropoulos, and T. E. Stern. Multicasting in a linear lightwave network. In *Proc. of the 12th IEEE Conference on Computer Communications*, pages 1350–1358. IEEE Press, 1993.

[12] E. Balas and E. Zemel. An algorithm for large zero-one knapsack problems. *Operations Research*, 28:1130–1154, 1980.

[13] J. Bar-Ilan, G. Kortsarz, and D. Peleg. Generalized submodular cover problems and applications. *Theoretical Computer Science*, 250(1-2):179–200, 2001.

[14] F. Barahona and R. Anbil. The volume algorithm: producing primal solutions with a subgradient method. *Mathematical Programming*, 87:385–399, 2000.

[15] R. Battiti and G. Tecchiolli. The reactive tabu search. *ORSA Journal on Computing*, 6:126–140, 1994.

[16] J. Beasley. OR-Library: Capacitated MST, 2005. `http://people.brunel.ac.uk/~mastjjb/jeb/orlib/capmstinfo.html`.

[17] J. E. Beasley. A heuristic for Euclidean and rectilinear Steiner problems. *European Journal of Operational Research*, 58:284–292, 1992.

[18] J. E. Beasley. *Lagrangean Relaxation*, pages 243–303. John Wiley & Sons, Inc. New York, NY, USA, 1993.

[19] R. E. Bellman. *Dynamic Programming*. Dover Publications Inc., 1957/2003.

[20] D. Bertsimas and J. N. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, 1997.

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

[22] K. Büdenbender, T. Grünert, and H.-J. Sebastian. A hybrid tabu search/branch-and-bound algorithm for the direct flight network design problem. *Transportation Science*, 34(4):364–380, 2000.

[23] L. Cassetta. Graph theory in network design and analysis. *Recent Studies in Graph Theory*, pages 29–63, 1989.

[24] B. V. Cherkassky and A. V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997.

code available at `http://www.avglab.com/andrew/CATS/maxflow_solvers.htm`.

[25] A. E. F. Clementi, M. D. Ianni, A. Monti, G. Rossi, and R. Silvestri. Experimental analysis of practically efficient algorithms for bounded-hop accumulation in ad-hoc wireless networks. In *Proc. of the 19th IEEE Int. Parallel and Distributed Processing Symposium (IPDPS'05), workshop 12*, volume 13, page 247.1, 2005.

[26] R. K. Congram, C. N. Potts, and S. L. van de Velde. An iterated Dynasearch algorithm for the single-machine total weighted tardiness scheduling problem. *INFORMS Journal on Computing*, 14(1):52–67, 2002.

[27] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[28] T. S. D. Whitley and D. Fuquay. Scheduling problems and traveling salesman: The genetic edge recombination operator. In *Proceedings on the Third International Conference on Genetic Algorithms*, pages 133–140, 1989.

[29] G. Dahl, T. Flatberg, N. Foldnes, and L. Gouveia. Hop-constrained spanning trees: The jump formulation and a relax-and-cut method. Technical report, University of Oslo, Centre of Mathematics for Applications (CMA), 2005.

[30] G. Dahl, L. Gouveia, and C. Requejo. On formulations and methods for the hop-constrained minimum spanning tree problem. In *Handbook of Optimization in Telecommunications*, chapter 19, pages 493–515. Springer Science + Business Media, 2006.

[31] E. Danna, E. Rothberg, and C. Le Pape. Exploring relaxation induced neighborhoods to improve MIP solutions. *Mathematical Programming, Series A*, 102:71–90, 2005.

[32] C. Darwin. *The Origin of Species*. John Murray, 1859.

[33] L. Davis. Applying adaptive algorithms to epistatic domains. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 162–164, 1985.

[34] M. P. de Aragão, E. Uchoa, and R. F. Werneck. Dual heuristics on the exact solution of large Steiner problems. *Electronic Notes in Discrete Mathematics*, 7:150–153, 2001.

[35] J. Denzinger and T. Offermann. On cooperation between evolutionary algorithms and other search paradigms. In W. Porto et al., editors, *Proceedings of the 1999 Congress on Evolutionary Computation (CEC)*, volume 3, pages 2317–2324. IEEE Press, 1999.

[36] N. Deo and A. Abdalla. Computing a diameter-constrained minimum spanning tree in parallel. In G. Bongiovanni, G. Gambosi, and R. Petreschi, editors, *Algorithms and Complexity*, number 1767 in LNCS, pages 17–31, Berlin, 2000. Springer-Verlag.

[37] I. Devarenne, H. Mabed1, and A. Caminada. Adaptive tabu tenure computation in local search. *Evolutionary Computation in Combinatorial Optimization*, 4972:1–12, 2008.

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

[39] M. Dorigo and G. D. Caro. *The Ant Colony Optimization Meta-Heuristic*, pages 11–32. McGraw-Hill, London, 1999.

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

[41] M. Dorigo, V. Maniezzo, and A. Colorni. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics-Part B*, 26:29–41, 1996.

[42] M. Dorigo and T. Stützle. *Ant Colony Optimization*. MIT Press, Cambridge, MA, 2004.

[43] A. C. dos Santos, A. Lucena, and C. C. Ribeiro. Solving diameter constrained minimum spanning tree problems in dense graphs. In *Proc. of the Int. Workshop on Experimental Algorithms*, volume 3059 of *LNCS*, pages 458–467. Springer, 2004.

[44] S. Dreyfus. Richard bellman on the birth of dynamic programming. *Operations Research*, 50(1):48–51, January 2002.

[45] A. Eiben and J. Smith. *Introduction to Evolutionary Computing*. Springer, Berlin Heidelberg, 2003.

[46] T. Feo and M. Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, 8:67–71, 1989.

[47] P. Festa, Mauricio, and G. C. Resende. Grasp: An annotated bibliography. In *Essays and Surveys in Metaheuristics*, pages 325–367. Kluwer Academic Publishers, 2002.

[48] G. R. Filho and L. A. N. Lorena. Constructive genetic algorithm and column generation: An application to graph coloring. In L. P. Chuen, editor, *Proceedings of APORS 2000, the Fifth Conference of the Association of Asian-Pacific Operations Research Societies within IFORS*, Singapore, 2000.

[49] M. Fischetti and A. Lodi. Local branching. *Mathematical Programming*, 98:23–47, 2002.

[50] A. P. French, A. C. Robinson, and J. M. Wilson. Using a hybrid genetic algorithm/branch and bound approach to solve feasibility and optimization integer programming problems. *Journal of Heuristics*, 7:551–564, 2001.

[51] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.

[52] F. Glover. Future paths for integer programming and links to artificial intelligence. *Decision Sciences*, 8:156–166, 1977.

[53] F. Glover and G. Kochenberger. *Handbook of Metaheuristics*, volume 57 of *International Series in Operations Research & Management Science*. Kluwer Academic Publishers, Norwell, MA, 2003.

[54] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, Boston, MA, 1997.

[55] F. Glover, M. Laguna, and R. Martí. Fundamentals of scatter search and path relinking. *Control and Cybernetics*, 39(3):653–684, 2000.

[56] D. E. Goldberg and R. Lingle. Alleles, loci and the traveling salesman problem. In *Proceedings of the First International Conference on Genetic Algorithms*, pages 154–159, 1985.

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

[58] L. Gouveia. Multicommodity flow models for spanning trees with hop constraints. *European Journal of Operational Research*, 95:178–190, 1996.

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

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

[61] L. Gouveia, T. L. Magnanti, and C. Requejo. An intersecting tree model for odd-diameter-constrained minimum spanning and Steiner trees. *Annals of Operations Research*, 146(1):19–39, 2006.

[62] L. Gouveia, T. L. Magnanti, and C. Requejo. Tight models for special cases of the diameter-constrained minimum spanning tree problem. In *Proc. of the Int. Network Optimization Conference*, Spa, Belgium, 2007.

[63] L. Gouveia, A. Paias, and D. Sharma. Modeling and solving the rooted distance-constrained minimum spanning tree problem. *Computers and Operations Research*, 35(2):600–613, 2008.

[64] L. Gouveia and C. Requejo. A new Lagrangean relaxation approach for the hop-constrained minimum spanning tree problem. *European Journal of Operational Research*, 132:539–552, 2001.

[65] L. Gouveia, L. Simonetti, and E. Uchoa. Modelling the hop-constrained minimum spanning tree problem over a layered graph. In *Proc. of the Int. Network Optimization Conference*, pages 1–6, Spa, Belgium, 2007.

[66] L. Gouveia, L. Simonetti, and E. Uchoa. Modelling hop-constrained and diameter-constrained minimum spanning tree problems as Steiner tree problems over layered graphs. Optimization Online, `http://www.optimization-online.org/DB_HTML/2008/06/2013.html`, 2008.

[67] M. Gruber and G. Raidl. A new 0–1 ILP approach for the bounded diameter minimum spanning tree problem. In L. Gouveia and C. Mourão, editors, *Proc. of the Int. Network Optimization Conference*, volume 1, pages 178–185, Lisbon, Portugal, 2005.

[68] M. Gruber and G. R. Raidl. Variable neighborhood search for the bounded diameter minimum spanning tree problem. In P. Hansen et al., editors, *Proc. of the 18th Mini Euro Conference on Variable Neighborhood Search*, Tenerife, Spain, 2005.

[69] M. Gruber and G. R. Raidl. Heuristic cut separation in a branch&cut approach for the bounded diameter minimum spanning tree problem. In *Proceedings of the 2008 International Symposium on Applications and the Internet, SAINT 2008*, pages 261–264, Turku, Finland, 2008. IEEE Computer Society.

[70] M. Gruber and G. R. Raidl. (Meta-)heuristic separation of jump cuts for the bounded diameter minimum spanning tree problem. In P. Hansen et al., editors, *Proceedings of Matheuristics 2008: Second International Workshop on Model Based Metaheuristics*, Bertinoro, Italy, 2008.

[71] M. Gruber and G. R. Raidl. Exploiting hierarchical clustering for finding bounded diameter minimum spanning trees on Euclidean instances. In G. R. Raidl et al., editors, *Proc. of the Genetic and Evolutionary Computation Conference 2009*, Montréal, Québec, Canada, to appear 2009. ACM.

[72] M. Gruber and G. R. Raidl. (Meta-)heuristic separation of jump cuts in a branch&cut approach for the bounded diameter minimum spanning tree problem, to appear 2009. special issue on Matheuristics of Operations Research/Computer Science Interface Series, Springer.

[73] M. Gruber and G. R. Raidl. Solving the Euclidean bounded diameter minimum spanning tree problem by clustering-based (meta-)heuristics. In A. Quesada-Arencibia et al., editors, *Twelfth International Conference on Computer Aided Systems Theory (EUROCAST 2009)*, Gran Canaria, Spain, to appear 2009. Springer LNCS.

[74] M. Gruber, J. van Hemert, and G. R. Raidl. Neighborhood searches for the bounded diameter minimum spanning tree problem embedded in a VNS, EA, and ACO. In M. Keijzer et al., editors, *Proc. of the Genetic and Evolutionary Computation Conference 2006*, volume 2, pages 1187–1194, Seattle, USA, 2006. ACM.

[75] P. Hansen and N. Mladenović. An introduction to variable neighborhood search. In S. Voß, S. Martello, I. Osman, and C. Roucairol, editors, *Metaheuristics, Advances and Trends in Local Search Paradigms for Optimization*, pages 433–458. Kluwer Academic Publishers, 1999.

[76] M. Haouari and J. C. Siala. A hybrid Lagrangian genetic algorithm for the prize collecting Steiner tree problem. *Computers & Operations Research*, 33(5):1274–1288, 2006.

[77] J. Holland. *Adaptation In Natural and Artificial Systems*. University of Michigan Press, 1975.

[78] H. Hoos and T. Stützle. *Stochastic Local Search – Foundations and Applications*. Morgan Kaufmann, San Francisco, CA, 2004.

[79] B. Hu and G. R. Raidl. Variable neighborhood descent with self-adaptive neighborhood-ordering. In C. Cotta, A. J. Fernandez, and J. E. Gallardo, editors, *Proceedings of the 7th EU/MEeting on Adaptive, Self-Adaptive, and Multi-Level Metaheuristics*, Malaga, Spain, 2006.

[80] A. Jain, M. Murty, and P.J.Flynn. Data clustering: a review. *ACM Computing Surveys (CSUR)*, 31:264–323, 1999.

[81] B. A. Julstrom. Encoding bounded-diameter minimum spanning trees with permutations and with random keys. In K. Deb et al., editors, *Genetic and Evolutionary Computation Conference – GECCO 2004*, volume 3102 of *LNCS*, pages 1282–1281. Springer, 2004.

[82] B. A. Julstrom. Greedy heuristics for the bounded diameter minimum spanning tree problem. *Journal of Experimental Algorithmics (JEA)*, 14:1.1:1–1.1:14, February 2009.

[83] B. A. Julstrom and G. R. Raidl. A permutation-coded evolutionary algorithm for the bounded-diameter minimum spanning tree problem. In A. Barry, F. Rothlauf, D. Thierens, et al., editors, *in 2003 Genetic and Evolutionary*

Computation Conference's Workshops Proceedings, Workshop on Analysis and Design of Representations, pages 2–7, 2003.

[84] N. Karmakar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[85] R. M. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

[86] L. Khachiyan. A polynomial algorithm in linear programming (english translation). *Soviet Mathematics Doklady*, 20:191–194, 1979.

[87] S. Kirkpatrick, C. Gellat, and M. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[88] G. Kortsarz and D. Peleg. Approximating shallow-light trees. In *SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 103–110, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[89] G. Kortsarz and D. Peleg. Approximating the weight of shallow Steiner trees. *Discrete Applied Mathemathics*, 93(2-3):265–285, 1999.

[90] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. of the American Mathematics Society*, 7(1):48–50, 1956.

[91] H. R. Lourenço, O. Martin, and T. Stützle. Iterated local search. In *Handbook of Metaheuristics* [53], pages 321–353.

[92] F. Maffioli. On constrained diameter and medium optimal spanning trees. In *5th Conference on Optimization Techniques Part II*, volume 4, pages 110–117. Springer, LNCS, 1973.

[93] T. Magnanti and R. Wong. Network design and transportation planning: Models and algorithms. *Transportation Science*, 18(1), 1984.

[94] T. L. Magnanti and L. A. Wolsey. *Handbooks in Operations Research and Management Science: Network Models*, chapter 9. Optimal Trees. Massachusetts Institute of Technology, Operations Research Center, 1994.

[95] P. Manyem and M. F. M. Stallmann. Some approximation results in multicasting. Technical Report TR-96-03, North Carolina State University at Raleigh, NC, USA, 1996.

[96] G. Mendel. Versuche über Pflanzen-Hybriden (experiments on plant hybridization). *Verhandlungen des naturforschenden Vereins Brünn (Proceedings of the Natural History Society of Brünn)*, 4:3–47, 1866.

[97] R. Montemanni and D. H. Smith. A tabu search algorithm with a dynamic tabu list for the frequency assignment problem. Technical report, University of Glamorgan, UK, 2001.

[98] B. M. E. Moret and H. D. Shapiro. An empirical analysis of algorithms for constructing a minimum spanning tree. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 400–411. Springer, 1991.

[99] P. Moscato. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. Technical Report C3P 826, California Institute of Technology, Pasadena, CA, 1989.

[100] P. Moscato. Memetic algorithms: A short introduction. In D. Corne et al., editors, *New Ideas in Optimization*, pages 219–234. McGraw Hill, 1999.

[101] S. B. Needlemana and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[102] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. Wiley-Interscience, 1988.

[103] T. F. Noronha, A. C. Santos, and C. C. Ribeiro. Constraint programming for the diameter constrained minimum spanning tree problem. *Electronic Notes in Discrete Mathematics*, 30:93–98, 2008.

[104] M. W. Padberg and L. A. Wolsey. Trees and cuts. *Annals of Discrete Mathematics*, 17:511–517, 1983.

[105] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982.

[106] S. Pirkwieser, G. R. Raidl, and J. Puchinger. Combining Lagrangian decomposition with an evolutionary algorithm for the knapsack constrained maximum spanning tree problem. In C. Cotta and J. van Hemert, editors, *Evolutionary Computation in Combinatorial Optimization – EvoCOP 2007*, volume 4446 of *Lecture Notes in Computer Science*, pages 176–187. Springer, 2007.

[107] A. Plateau, D. Tachat, and P. Tolla. A hybrid search combining interior point methods and metaheuristics for 0–1 programming. *International Transactions in Operational Research*, 9:731–746, 2002.

[108] M. Prandtstetter and G. R. Raidl. A variable neighborhood search approach for solving the car sequencing problem. In P. Hansen et al., editors, *Proceedings of the 18th Mini Euro Conference on Variable Neighborhood Search*, Tenerife, Spain, 2005.

[109] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[110] J. Puchinger and G. R. Raidl. An evolutionary algorithm for column generation in integer programming: An effective approach for 2D bin packing. In X. Yao et al., editors, *Parallel Problem Solving from Nature – PPSN VIII*, volume 3242 of *Lecture Notes in Computer Science*, pages 642–651. Springer, 2004.

[111] J. Puchinger and G. R. Raidl. Combining metaheuristics and exact algorithms in combinatorial optimization: A survey and classification. In *Proceedings of the First International Work-Conference on the Interplay Between Natural and Artificial Computation*, volume 3562 of *LNCS*, pages 41–53. Springer, 2005.

[112] J. Puchinger and G. R. Raidl. Relaxation guided variable neighborhood search. In *Proceedings of the XVIII Mini EURO Conference on VNS*, Tenerife, Spain, 2005.

[113] J. Puchinger, G. R. Raidl, and M. Gruber. Cooperating memetic and branch-and-cut algorithms for solving the multidimensional knapsack problem. In *Proc. of the 6th Metaheuristics Int. Conference*, pages 775–780, Vienna, Austria, 2005.

[114] J. Puchinger, G. R. Raidl, and U. Pferschy. The core concept for the multidimensional knapsack problem. In J. Gottlieb and G. R. Raidl, editors, *Evolutionary Computation in Combinatorial Optimization – EvoCOP 2006*, volume 3906 of *Lecture Notes in Computer Science*, pages 195–208. Springer, 2006.

[115] G. R. Raidl and H. Feltl. An improved hybrid genetic algorithm for the generalized assignment problem. In H. M. Haddadd et al., editors, *Proceedings of the 2003 ACM Symposium on Applied Computing*, pages 990–995. ACM Press, 2004.

[116] G. R. Raidl and B. A. Julstrom. Greedy heuristics and an evolutionary algorithm for the bounded-diameter minimum spanning tree problem. In G. Lamont et al., editors, *Proc. of the ACM Symposium on Applied Computing*, pages 747–752. ACM Press, 2003.

[117] G. R. Raidl and J. Puchinger. Combining (integer) linear programming techniques and metaheuristics for combinatorial optimization. In C. Blum, M. J. B. Augilera, A. Roli, and M. Sampels, editors, *Hybrid Metaheuristics – An Emergent Approach for Combinatorial Optimization*, volume 114 of *Studies in Computational Intelligence*, pages 31–62. Springer, 2008.

[118] G. R. Raidl, J. Puchinger, and C. Blum. Metaheuristic hybrids. In M. Gendreau and J. Y. Potvin, editors, *Handbook of Metaheuristics*. Springer, 2nd edition, submitted 2008 (invited).

[119] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, 1989.

[120] I. Rechenberg. *Evolutionsstrategie, Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution*. Frommann-Holzboog Verlag, 1973.

[121] W. Rei, J.-F. Cordeau, M. Gendreau, and P. Soriano. Accelerating Benders decomposition by local branching. *INFORMS Journal on Computing*, 2008. In press.

[122] A. Rényi and G. Szekeres. On the height of trees. *Journal of the Australian Mathematical Society*, 7:497–507, 1967.

[123] M. Resende and C. Ribeiro. Greedy randomized adaptive search procedures. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, pages 219–249. Kluwer Academic Publishers, 2003.

[124] J. Riordan. The enumeration of trees by height and diameter. *IBM Journal of Research and Development*, 4(5):473–478, 1960.

[125] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. ELSEVIER, 2006.

[126] M. Ruthmair and G. R. Raidl. A Kruskal-based heuristic for the rooted delay-constrained minimum spanning tree problem. In A. Quesada-Arencibia et al., editors, *Twelfth International Conference on Computer Aided Systems Theory (EUROCAST 2009)*, Gran Canaria, Spain, to appear 2009. Springer LNCS.

[127] H. F. Salama, D. S. Reeves, and Y. Viniotis. An efficient delay constrained minimum spanning tree heuristic, 1996. Submitted to the Fifth Int. Conference on Computer Communications and Networks, October 1996.

[128] H.-P. Schwefel. *Evolution and Optimum Seeking*. Wiley, New York, 1995.

[129] A. Singh and A. K. Gupta. Improved heuristics for the bounded-diameter minimum spanning tree problem. *Soft Computing – A Fusion of Foundations, Methodologies and Applications*, 11(10):911–921, 2007.

[130] A. T. Staggemeier, A. R. Clark, U. Aickelin, and J. Smith. A hybrid genetic algorithm to solve a lot-sizing and scheduling problem. In B. Lev, editor, *Proceedings of the 16th triannual Conference of the International Federation of Operational Research Societies*, Edinburgh, U.K., 2002.

[131] T. Stützle and H. Hoos. The $\mathcal{MAX}$–$\mathcal{MIN}$ ant system and local search for combinatorial optimization problems: Towards adaptive tools for combinatorial global optimisation. *Meta-Heuristic, Advances and Trends in Local Search Paradigm for Optimization*, pages 313–329, 1998.

[132] G. Syswerda. Schedule optimization using genetic algorithms. In L. Davis, editor, *Handbook of Genetic Algorithms*, pages 332–349. Int. Thomson Computer Press, 1991.

[133] G. Szekeres. *Combinatorial Mathematics X*, volume 1036 of *Lecture Notes in Mathematics*, chapter Distribution of Labeled Trees by Diameter, pages 392–397. Springer-Verlag, 1983.

[134] S. Talukdar, S. Murthy, and R. Akkiraju. Asynchronous teams. In *Handbook of Metaheuristics*, pages 537–556. Kluwer Academic Publishers, 2003.

[135] M. Vasquez and Y. Vimont. Improved results on the 0–1 multidimensional knapsack problem. *European Journal of Operational Research*, 165(1):70–81, 2005.

[136] V. V. Vazirani. *Approximation algorithms*. Springer, 2001.

[137] S. Voß. The Steiner tree problem with hop constraints. *Annals of Operations Research*, 86:321–345, 1999.

[138] L. A. Wolsey. *Integer Programming*. Wiley-Interscience, 1998.

[139] D. L. Woodruff. A chunking based selection strategy for integrating metaheuristics with branch and bound. In S. Voss et al., editors, *Metaheuristics: Advances and Trends in Local Search Paradigms for Optimization*, pages 499–511. Kluwer Academic Publishers, 1999.

[140] K. A. Woolston and S. L. Albin. The design of centralized networks with reliability and availability constraints. *Computers and Operations Research*, 15(3):207–217, 1988.

Appendix A

Curriculum Vitae

Personal Information

- Name: Martin Gruber
- Date of birth: September 3rd, 1971
- Place of birth: Vienna, Austria

Education

- since 06/2004: PhD student at Vienna University of Technology. Main research: "Network Design: The Bounded Diameter Minimum Spanning Tree Problem", supervised by Günther Raidl
- 2006 – 09/2007: Computer Management studies at Vienna University of Technology with graduation to "Magister rer.soc.oec." (MSc) with distinction.
- 1990 – 11/2003: Computer Science studies at Vienna University of Technology with graduation to "Diplom Ingenieur" (MSc) with distinction. Diploma thesis: "Effiziente Gestaltung der Wortanalyse in SiSiSi (efficient implementation of the word analysis for a reliable and sense-conveying German hyphenation system)", supervised by Wilhelm Barth and Gabriele Koller
- 09/1982 – 06/1990: Comprehensive secondary school (Bundesrealgymnasium) BG/BRG 15 in Vienna, Austria.
- 09/1978 – 06/1982: Primary school (Volksschule) in Vienna, Austria.

Work Experience (Academic)

- since 03/2005: Research and teaching assistant, Algorithms and Data Structures Group, Institute of Computer Graphics and Algorithms, Vienna University of Technology
- 06/2004 – 01/2006: Employed in the FWF project *Combining Memetic Algorithms with Branch&Cut&Price for Some Network Design Problem* under grant P16263-N04, Algorithms and Data Structures Group, Institute of Computer Graphics and Algorithms, Vienna University of Technology
- 1995 – 2000:
  Tutor ("Studienassistent") for the course "Algorithms and Datastructures 2", Institute of Computer Graphics and Algorithms, Vienna University of Technology, Austria.
  Tutor ("Studienassistent") for the course "Introduction to Programming for Electrical Engineering Technician", Institute of Computer Graphics and Algorithms, Vienna University of Technology, Austria.
  Tutor ("Studienassistent") for the course "Process Automation", Institute of Computer Aided Automation, Vienna University of Technology, Austria.
  Tutor ("Studienassistent") for the course "Socio-scientifical Fundamentals of Computer Science", Institute of Design and Assessment of Technology, Vienna University of Technology, Austria.

Publications

Refereed Journal Articles

- Martin Gruber and Günther R. Raidl. (Meta-)heuristic separation of jump cuts in a Branch&Cut approach for the bounded diameter minimum spanning tree problem. To appear in *Matheuristics of Operations Research*, Computer Science Interface Series, Springer, 2009.

Refereed Conference Papers

- Martin Gruber and Günther R. Raidl. Exploiting Hierarchical Clustering for Finding Bounded Diameter Minimum Spanning Trees on Euclidean Instances. To appear in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2009)*, Montréal Québec, Canada, 2009, ACM.
- Martin Gruber and Günther R. Raidl. Solving the Euclidean bounded diameter minimum spanning tree problem by clustering-based (meta-)heuristics. To

appear in *Proceedings of the Twelfth International Conference on Computer Aided Systems Theory (EUROCAST 2009)*, LNCS, Gran Canaria, Spain, 2009.

- Martin Gruber and Günther R. Raidl. (Meta-)heuristic separation of jump cuts for the bounded diameter minimum spanning tree problem. *Proceedings of Matheuristics 2008: Second International Workshop on Model Based Metaheuristics*, Bertinoro, Italy, 2008.
- Günther R. Raidl and Martin Gruber. A Lagrangian relax-and-cut approach for the bounded diameter minimum spanning tree problem. *Numerical Analysis and Applied Mathematics*, volume 1048 of *AIP Conference Proceedings*, pages 446–449. American Institute of Physics, 2008.
- Martin Gruber and Günther R. Raidl. Heuristic cut separation in a Branch&Cut approach for the bounded diameter minimum spanning tree problem. *Proceedings of the 2008 International Symposium on Applications and the Internet (SAINT 2008)*, pages 261–264, Turku, Finland, 2008, IEEE Computer Society.
- Martin Gruber, Jano van Hemert, and Günther R. Raidl. Neighborhood searches for the bounded diameter minimum spanning tree problem embedded in a VNS, EA, and ACO. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2006)*, volume 2, pages 1187–1194, Seattle, USA, 2006, ACM.
- Martin Gruber and Günther R. Raidl. Variable neighborhood search for the bounded diameter minimum spanning tree problem. *Proceedings of the 18th Mini Euro Conference on Variable Neighborhood Search*, Tenerife, Spain, 2005.
- Jakob Puchinger, Günther R. Raidl, and Martin Gruber. Cooperating memetic and Branch&Cut algorithms for solving the multidimensional knapsack problem. *Proceedings of MIC2005, the 6th Metaheuristics International Conference*, pages 775-780, Vienna, Austria, 2005.
- Martin Gruber and Günther R. Raidl. A new 0-1 ILP approach for the bounded diameter minimum spanning tree problem. *Proceedings of the 2nd International Network Optimization Conference*, volume 1, pages 178–185, Lisbon, Portugal, 2005.

Master Thesis

- Martin Gruber. Effiziente Gestaltung der Wortanalyse in SiSiSi. Master's thesis, Vienna University of Technology, Institute of Computer Graphics and Algorithms, November 2003. Supervised by Wilhelm Barth and Gabriele Koller.

Textbooks

- Martin Schönhacker and Martin Gruber. Programmieren 2. Textbook for the lecture with the same title at the Vienna University of Technology, Austria, 1998.

Organizational and Reviewing Activities

- Member of organizing committee of the *Graph Drawing Conference GD 2001* (Software Exhibition). Vienna, Austria, September 2001.
- *Evolutionary Computation Journal*, 2008, 2009.
- *Operations Research Letters*, 2009.