Diplomarbeit

Finding Provably Optimal Solutions for the (Prize Collecting) Steiner Tree Problem

Ausgeführt am
Institut für Computergraphik und Algorithmen
der Technischen Universität Wien

Unter der Anleitung von

Univ.Prof. Dr. Petra Mutzel

und

Dr. Ivana Ljubic

durch

Andreas Moser
Turracherstrasse 30
9560 Feldkirchen

Februar 2005 _____

Acknowledgments

First of all I want to thank my advisor Petra Mutzel and her group at the department of Algorithms and Data Structures at the Technical University of Vienna for the possibility to write my thesis about this interesting topic and for the outstanding support during the last years.

My special thanks also belong to Ivana Ljubic for the great support during the time of hard work and for taking the time for proofreading this thesis. I could not have imagined having a better advisor for writing this thesis.

I also want to thank Ulrich Pferschy, Gunnar Klau and René Weisskircher for the lively discussions we had about LP formulations for the Steiner Tree problem, which led to great results.

The major part of the practical work for this thesis was done during a six month stay at the University of La Laguna in Tenerife. I want to thank Prof. Dr. María Belén Melián Batista and her group at the institute for Estadística, Investigación Operativa y Computación for providing me with the necessary facilities, their support and for the possibility to spend a great time on this wonderful island.

I also want to give sincere thanks to Prof. Mutzel, Prof. Raidl and Prof. Mehlmann for holding the final exam.

Last but not least I want to thank all my friends for the wonderful time we spent together and my parents for their support during all those years.

Short Abstract

In this thesis we present a method to solve the Steiner Tree Problem as well as the prize collecting Steiner Tree Problem to provable optimality. Although those problems have been shown to be in the class of NP hard problems there exist important practical applications thus algorithms that produce good solutions are needed.

The approach presented in this thesis solves those problems using Integer Linear Programming. We propose the addition of $\{0, \frac{1}{2}\}$-Cuts and Local Branching to the solving process for the Integer Linear Program which drastically improve the quality of the obtained bounds compared to the standard branch and cut methods used to solve Integer Linear Programs.

We apply our new approach to the set of most difficult Steiner Tree instances and on a set of easily solvable instances for the prize collecting Steiner Tree problem available from literature and show that within short time our method is able to find very good bounds for most of those instances and better upper bounds than the currently best known ones for some of them.

Deutsche Zusammenfassung

In diese Diplomarbeit präsentieren wir eine Methode zur optimalen Lösung des "Steiner Tree Problems" und des "Prize Collecting Steiner Tree Problems". Es ist bekannt, dass diese zwei Probleme in der Klasse der NP schweren Probleme liegen. Trotzdem werden in der Praxis Algorithmen benötigt, die gute Lösungen liefern können, da wichtige praktische Anwendungsgebiete bestehen.

Die Ansätze die wir in dieser Arbeit präsentieren, lösen die genannten Probleme mittels Ganzzahliger Linearer Programmierung. Wir zeigen, wie man durch Hinzufügen von sogenannten $\{0, \frac{1}{2}\}$-Cuts und "Local Branching" den Lösungsprozess für Ganzzahlig Lineare Programme so verändern kann, dass die Qualität der erhaltenen Grenzen signifikant besser ist als die, die man bei Verwendung des "branch and cut" Standardlösungsansatzes für Ganzzahlige Programmierung erhält.

Wir verwenden unseren Lösungsansatz um eine Menge der schwierigsten Instanzen für das Steiner Tree Problem und eine Menge von einfacheren Instanzen für das prize collecting Steiner Tree Problem aus der Literatur zu lösen. Wir zeigen, dass unser Ansatz innerhalb kurzer Zeit sehr gute Lösungen für alle diese Instanzen liefert und für manche der schwierigen Instanzen sogar die besten bekannten Grenzen verbessern kann.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the last few years as more and more companies are trying to provide services that aim at a large number of customers various network planning problems have gained a lot of importance. Planning well designed distribution networks is a very challenging task but the quality of those networks affects directly the profit a company can make by providing services to their customers. One interesting subset of these problems arises in the field of energy companies namely the design and planning of district heating networks.

Since companies are forced to reduce their greenhouse emissions and also for plain economic reasons the use of biomass for heat generation is a very attractive possibility. This area of energy distribution is characterized by extremely high investment costs but also by an unusually loyal customer base and limited competition. Therefore constructing district heating networks is a business that pays well. However, the design phase is very crucial for building a high profit network.

In a typical planning scenario the input is a set of potential customers with known or estimated heat demands, which can be represented as future profits, and a potential network for laying the pipelines. This network is usually determined by the underlying street network of the district or town. Costs of the network are dominated by labor and right-of-way charges for laying the pipes and the costs for building the heating plant.

The planning process now consists of two main parts. On one hand a subset of profitable customers has to be selected from the set of all potential customers. This subset will be the customers who will be provided with district heating. On the other hand, a network has to be designed to connect all the selected customers to the heating plant. Clearly the cost of this network should be as low as possible.

This process can be described as an optimization problem in the following way. Given is an undirected graph $G = (V, E)$, where some vertices $S \subseteq V$ are considered customer nodes and the edges E with costs, $c : E \mapsto R^+$.

This graph represents the map of a district where a district heating network has to be built. The cost function c represents the costs that have to be paid for laying a pipe along the corresponding street segment. The goal of a profit orientated company is to maximize the total revenue which can be estimated by the sum of the gains for the connected customers minus the costs for building the network.

A formal definition of the problem can be given as follows:

Definition 1 The Steiner Tree Problem, ST *Given is a weighted undirected graph $G = (V, E, c)$ and a set $S \subseteq V$, the so called terminal nodes.* The Steiner Tree Problem *(ST) consists in finding a connected subgraph $T = (V_T, E_T)$ of G, $V_T \subseteq V, E_T \subseteq E$ that spans all the nodes in S, $S \subseteq V_T$, and minimizes the cost function*

$$costs(T) = \sum_{e \in E_T} c(e) \tag{1.1}$$

It is easy to see that every optimal solution T will be a tree. Otherwise T would contain a cycle and removing any edge from the cycle would reduce $costs(T)$ without violating the connectivity of T.

Of course when such district heating networks are designed in most cases a slightly more complex version of this problem has to be solved. As the goal of energy providing companies is mainly to improve their profit it often does not make sense for them to connect customers with very little demand or which are quite hard to reach to their network. In those cases the problem solving procedure consists of another part, namely in selecting a profitable subset of the potential customers which will be taken into consideration when actually constructing the network. Normally the demand for each customer is known to the company or can at least be estimated but as the steps of selecting customers and designing the actual network have to be done at the same time, the complexity of the problem clearly increases.

In the corresponding optimization problem an undirected graph $G = (V, E)$ is given, where the vertices V are associated with profits, $p : V \mapsto R^+$, and the edges E with costs, $c : E \mapsto R^+$. The profit function p now represents the estimated gains for connecting this customer to the network while, as before, the graph G represents the map of a district and function c represents the costs that have to be paid for laying a pipe. The total revenue, which has to be maximized, is now defined as the sum of the estimated gains for the connected customers minus the costs for building the network.

A formal definition which is similar to the definition of the Steiner Tree Problem is given as follows:

Definition 2 The prize collecting Steiner Tree Problem, PCST *Let $G = (V, E, c, p)$ be an undirected weighted graph. The* Linear prize collecting

Steiner Tree problem *(PCST) consists of finding a connected subgraph* $T = (V_T, E_T)$ *of* G, $V_T \subseteq V$, $E_T \subseteq E$ *that maximizes*

$$profit(T) = \sum_{v \in V_T} p(v) - \sum_{e \in E_T} c(e) \ . \tag{1.2}$$

Throughout this paper we will distinguish between *customer vertices*, defined as

$$R = \{v \in V \mid p(v) > 0\} \ ,$$

and *non-customer vertices*, corresponding to street intersections, and we assume that $R \neq \emptyset$. For the ST, we will use the same naming and refer to the set of terminal nodes $S \subseteq V$ as customer nodes or potential customer nodes. $p(v)$ and $c(e)$ will often be referred to as p_v and c_e, respectively.

Chapter 2

Known Variants of the Steiner Tree Problem

In practice often additional side constraints have to be considered when solving instances of those problems. The planning problem of the heating network for example clearly requires that the heating plant is connected to the network. This can be modeled as a ST or a PCST by introducing a special vertex for the plant which has to be present in every solution. This can be achieved by various means like for instance by giving the heating plant a very high profit.

In general, the *rooted prize-collecting Steiner tree problem* (RPCST), is defined as a variant of PCST with an additional source vertex $v_s \in V$ representing a depot or repository which must be a part of every feasible solution T. The *rooted Steiner Tree Problem* (RST) is defined analogous.

Another interesting variant of PCST arises if the energy company in our application chooses not to maximize the absolute gain of a project but rather the return on investment (RoI). In this case we want to maximize the ratio of profits to costs. This problem is called the *fractional prize-collecting Steiner tree problem* (FPCST). The objective function for this problem is:

$$\max \frac{\sum_{v \in V_T} p(v)}{c_0 + \sum_{e \in E_T} c(e)} \tag{2.1}$$

over all subtrees T of G, where $c_0 > 0$ represents the fixed cost of the project which could be the setup costs of the heating plant in our application. Note that without the inclusion of c_0 in the definition, the empty set, which is a trivial feasible solution, would produce an undefined objective function value.

As with the PCST, the rooted version of FPCST is a relevant special case to consider. However solving instances of the rooted FPCST is not as easy as solving the rooted PCST as a node with artificially high profit would distort the ratio in (2.1).

It is obvious that we cannot directly apply an integer linear programming procedure to solve this problem. Nevertheless, since both numerator and denominator are linear functions, the problem belongs to the broader group of so-called *linear fractional combinatorial optimization problems* (LFCOs). In the paper [23] the authors describe the solving of the FPCST using Newton's iterative method (cf. Radzik [29]) where several linear rooted PCST instances have to be solved to optimality for obtaining one solution for the FPCST.

In the special case where the given graph G is a tree, PCST can indeed be solved in $O(|V|)$ time which can be further exploited to construct an $O(|V| \log |V|)$ algorithm for FPCST in this special case.

Another interesting variation of the ST as well as of the PCST which arises in the field of distance heating networks is the so called *piecewise linear Steiner tree problem*. This variant tries to model the fact that normally the costs for laying district heating pipes do not only correspond to the length of the pipe and the area where the pipe has to be laid but also to the size of the pipe. Obviously burying a small pipe for connecting a few houses is a lot cheaper than burying the very large connection to the heating plant. Therefore now the penalty for constructing each part of the network consists of fixed and variable costs. For laying the pipe the company has to pay the fixed costs which are higher for larger pipes and depend on the surface and on the length. For transporting energy along these pipes, the variable costs have to be paid which depend on the amount of flow along the pipe. In exchange for the higher fixed investment, larger pipes have lower transport expenses.

This leads to an objective function that contains piecewise linear variables. Clearly every solution also has to state, which types of pipes have to be used to build the network.

Chapter 3

Previous Work

In 1972, Karp [22] showed the NP-completeness of the Steiner tree problem (ST) which also implies that the prize collecting Steiner tree problem, a more general case of the ST, is NP-complete.

In 1987, Segev [31] considered for the first time the so-called *Node Weighted Steiner Tree Problem* (NWST) – the Steiner tree problem with node weights. In addition to regular edge weights, nodes have weights and the goal is to minimize the sum of edge-costs and node-weights. The difference between NWST and PCST is that the former requires a set of terminal vertices to be included in the solution which is quite similar to the classical Steiner tree problem. Segev also noted that NWST can be turned into a directed Steiner tree problem, if the node weights are non-negative and the root of a solution tree is given.

He also contributed observations on a special case of NWST, called the *single point weighted Steiner tree* problem (SPWST). In this problem a special vertex is given that has to be included in the solution. The weights on the remaining vertices are non-positive profit values, while non-negative weights on the edges reflect the costs in obtaining or collecting these profits.

Negating the node weights to make them positive and subtracting them from the edge costs in the objective function, immediately yields the objective function of the rooted PCST. Therefore, as long as optimization is concerned, SPWST is equivalent to our definition of RPCST.

Many algorithms have been proposed for solving the ST. An exhaustive list of related publications can be found at [17]. In 1985 and 1987, Winter presented algorithms on the ST ([32] and [33]).

Another algorithm was presented by Beasley in 1989 ([4]). Definitions and algorithms on the ST can for example be found in [20].

An article on preprocessing techniques for the ST that may also be modified for the PCST was published by Duin in 1989 ([12]). Polzin and Vahdati applied a partitioning approach to the ST in [28].

The term “prize-collecting” was introduced in 1989 by Balas [3] in the

context of the traveling salesman problem. The node weights in his model are non-negative and can be seen as penalties for not including nodes in a solution.

3.1 Approximation Algorithms

The first approximation algorithm for both the PCST and the prize-collecting traveling salesman problem has been proposed by Bienstock et al. [5]. Those algorithms bear approximation guarantees of 3 and 5/2, respectively.

Goemans and Williamson presented in [19] a purely combinatorial general approximation technique for a large class of constrained forest problems. Their algorithm is based on a primal-dual schema, runs in $O(n^2 \log n)$ time ($n := |V|$), and yields solutions within a factor of $2 - \frac{1}{n-1}$ of optimality for most of the considered problems: the generalized Steiner tree problem, the T-join problem, the minimum-weight perfect matching problem, etc.

The authors also provided an extension of the basic algorithm and proposed, in particular, algorithms for the prize-collecting Steiner tree and prize-collecting TSP problems. To solve the unrooted PCST, the Goemans-Williamson algorithm is performed for each vertex as a possible root. Thus, the total running time of the algorithm is $O(n^3 \log n)$. Recently, Johnson et al. [21] improved the Goemans-Williamson algorithm by enhancing the second phase, the so-called *pruning phase*. The new algorithm is slightly faster and provides solutions that are provably at least as good and in practice significantly better. The authors also provided a modification to the growth phase to make the algorithm independent of the choice of the root vertex, thus giving a $(2 - \frac{1}{n-1})$–approximation algorithm for the general unrooted PCST which runs in $O(n^2 \log n)$ time. In exhaustive tests by Minkoff [26] the performance of the original Goemans-Williamson algorithm and the proposed improvement are compared on benchmark instances obtained from county street maps and randomly generated instances.

Feofiloff et al. [14] present a revised proof for the $(2-\frac{1}{n-1})$–approximation algorithm by Johnson et al. and give an example showing that this ratio is tight. The authors also proposed a modification of the Goemans-Williamson algorithm based on a slightly different linear programming formulation. The new algorithm achieves a ratio of $2 - \frac{2}{n}$ and runs in $O(n^2 \log n)$ time.

3.2 Lower Bounds and Polyhedral Studies

In [31], Segev presented single- and multi-commodity flow formulations for SPWST. Furthermore, the author developed two bounding procedures based on Lagrangian relaxations of the corresponding flow formulations which were embedded in a branch-and-bound procedure. In addition, heuristics to compute feasible solutions were also included. The proposed algorithm was

tested on a set of benchmark instances with up to 40 vertices.

Fischetti [15] studied the facial structure of a generalization of the problem, the so-called *Steiner arborescence* (or *directed Steiner tree*) problem and pointed out that the NWST can be transformed into it. The author considered several classes of valid inequalities and introduced a new inequality class with arbitrarily large coefficients, showing that all of them define distinct facets of the underlying polyhedron. More details on the transformation of the PCST into the Steiner arborescence problem are shown in Section 5.3.

Goemans provided in [18] a theoretical study on the polyhedral structure of the node-weighted Steiner tree problem.

Chopra presented formulations for the Steiner Tree Problem in 1994 ([9]). A comparison of Steiner Tree Relaxations is given by Polzin and Daneshmand in [27]. The authors also introduced a new formulation based on the standard multicomodity flow formulation and proved that the new approach provides the best lower bounds.

Engevall et al. [13] proposed another ILP formulation for the NWST, based on the *shortest spanning tree* problem formulation, introduced originally by Beasley [4] for the Steiner tree problem. In their formulation, besides the given root vertex r, an artificial root vertex 0 is introduced, and an edge between vertex 0 and r is set. They searched for a tree with additional constraints: each vertex v connected to vertex 0 must have degree one. The solution is interpreted so that the vertices adjacent to vertex 0 are not taken as a part of the final solution. For the description of the tree, the authors use a modification of the generalized subtour elimination constraints. For finding good lower bounds, the authors use a Lagrangian heuristic and subgradient procedure based on the shortest spanning tree formulation. Experimental results presented for instances with up to 100 vertices indicated that the new approach outperformed Segev's algorithm.

Lucena and Resende [25] presented a cutting plane algorithm for the PCST also based on Beasley's shortest spanning tree formulation and the generalized subtour elimination constraints. Their algorithm contains basic reduction steps similar to those already given by Duin and Volgenant [11], and was tested on two groups of benchmark instances: the first group contains instances adopted from Johnson et al. [21], ranging from 100 vertices and 284 edges to 400 vertices and 1507 edges. The second group is derived from the Steiner problem instances (series C and D) of the OR-library, with sizes ranging from 500 vertices and 625 edges to 1000 vertices and 25000 edges. The proposed algorithm solved many of the considered instances to optimality, but not all of them.

3.3 Heuristic Algorithms

Canuto et al. [6] developed a multi-start local-search-based algorithm with perturbations. Perturbations are done by changing the parameters of the input graph, either by setting profits of potential customers to zero, or by modifying the profits of non-zero vertices. Feasible solutions are obtained by the Goemans-Williamson algorithm, followed by a local search procedure. Within local search, the complete 1-flip neighborhood is examined, and the best solution found so far is selected and inserted in a pool of high-quality elite solutions. Between a randomly selected solution from the pool, and the solution found in the current iteration, *path relinking* is applied, exploring trajectories that connect these two solutions. A variable neighborhood search method is finally applied as a post-optimization step. The algorithm found optimal solutions on nearly all test instances for which the optimum is known.

Chapter 4

Preliminaries

In this thesis a method for finding provably optimal solutions for the PCST and the ST is presented that uses integer linear programming. Unlike the flow based formulations for the PCST (see Section 3.2), this formulation uses a different approach that is based on minimum s, t - cuts.

Section 4.1 gives a very short overview of (integer) linear programming. In Section 5.1 the idea behind cut based method is explained.

4.1 Integer Linear Programming

Linear Program A *Linear Program* (LP) is a mathematical optimization problem that consists of an objective function and of constraints formulated as inequalities. As the name states, every single constraint as well as the objective function is a linear combination of the available variables. The goal is to find a variable assignment that does not violate any of the given constraints and optimizes the value of the objective function.

The standard notation of a LP is

$$\max \quad c^T x \tag{4.1}$$

$$\text{subject to} \quad Ax \leq b \tag{4.2}$$

where the abbreviations have the following meaning:

x	$\in \mathbb{R}^n$	variables
A	$\in \mathbb{R}^{m \times n}$	coefficients of the variables in the constraints
b	$\in \mathbb{R}^m$	constant right hand side values
c	$\in \mathbb{R}^n$	coefficients of the variables in the objective function

(4.1) is the objective function that has to be maximized and (4.2) are the constraints that have to be considered. Although the standard form for LPs presented here seems very restricted, it can easily be enhanced by introducing other constraints like equalities or greater or equal - inequalities.

A constraint $a^T x \geq b$ can be rewritten as $-a^T x \leq -b$ and is therefore usable in the standard form of the LP. A equality $a^T x = b$ can be modeled by adding two constraints to the LP. The appropriate inequalities are

$$a^T x \leq b$$

and

$$-a^T x \leq -b.$$

By applying similar methods, the standard form may also be used to solve minimization problems. The objective function $\min c^T x$ can be rewritten in the form $\max -c^T x$ which satisfies the criteria of the standard formulation.

The widely known methods for solving LPs like the Ellipsoid method or Interior Point methods are able to solve this class of optimization problems in polynomial time which makes linear programs very useful for practical applications.

Integer Linear Program An *Integer Linear Program* is an extension of a Linear Program. If a nonempty real subset of the variables x is demanded to be integer in the solution, we call this a *Mixed Integer Program (MIP)*. If all variables have to be integer we have a so called *Integer Linear Program (ILP)*. Unfortunately, in the general case solving instances of Integer Linear Programs is NP hard and can only be solved using time consuming methods like Branch-and-Cut or Branch-and-Cut-and-Prize.

$\{0,1\}$-Integer Programming A special case of an ILP arises, if all integer variables are decision variables which means that for every integer variable x, $x \in \{0,1\}$. This special case is called $\{0,1\}$-*Integer Programming*.

Branch-and-Cut A widely applied approach to solve ILPs is the Branch-and-Cut method. This is a simple divide and conquer method that starts by solving the linear program relaxation of an ILP. If the solution to this problem is integer, the problem is already solved. In most cases there are variables that have fractional values assigned in the LP solution. In this case, the problem is split up by rounding one of the fractional variables. In the case of $\{0,1\}$-Integer Programming two new problems are generated. For one of them the branching variable x_b is set to 0 and in the other to 1. Those problems are then solved recursively until no problems are left to process.

In this thesis we will present an ILP formulation for solving ST and PCST instances and we will also show how Local Branching, an improvement to the normal Branch-and-Cut method, can be applied to those problems (Section 8).

For solving the integer linear programs presented in this thesis we used the widely used commercial MIP solver ILOG Cplex 8.1.

4.2 Flow Networks

This section gives some important definitions of graphs and problems on graphs that are used throughout this thesis.

Definition 3 (Directed) Graph *A* Graph *is a tuple $G = (V, E)$ of a finite set of nodes V and a finite set of edges E. An edge e is a tuple (v, w) of two nodes $v, w \in V$. If all the edges of a graph are ordered, which means they are a* ordered *tuple of nodes, we call the graph* directed.

For every node v the number of edges incident to it is called the *degree* of v. If we consider only incoming arcs, we call it the *indegree*. Analogous we define the *outdegree* as the number of arcs going out from v.

For every edge (u, v), u is called the *source* of e and v the *target* of e. A *path* in a graph G is a sequence of vertices such that for every vertex in the sequence there exists an edge to its successor.

Definition 4 Flow Network *A Flow Network is a directed connected graph with some special properties. There exists a supply function $sup : V \mapsto \mathbb{Z}$ that determines for every node $v \in V$ how many units of flow it produces or consumes and a capacity function cap that maps edges to positive numbers, $cap : E \mapsto \mathbb{Z}^+$. Additionally every edge may get costs assigned by a cost function $cost : E \mapsto \mathbb{Z}$.*

There are two types of nodes that are determined by the supply function.

- If a node s has a value $sup(s) > 0$ we call this node a *source*.
- If a node t has a value $sup(t) < 0$ we call this node a *sink*.

Throughout this thesis we will deal with flow networks that have exactly one source s with $sup(s) = C$ and exactly one sink t with $sup(t) = -C$.

Definition 5 Maximum Flow Problem *The* Maximum Flow Problem *consists in finding the maximum flow between any two vertices s and t of a flow network considering the capacity function $cap : E \mapsto \mathbb{Z}^+$ and the supply function $sup : V \mapsto \mathbb{Z}$.*

Definition 6 Minimum weight s,t - Cut *The set of edges S with minimal $\sum_{e \in S} cap(e)$ in a* flow network *which separates the distinct vertices s and t when removed is called a* minimum weight s,t - Cut.

Theorem 1 Max-Flow-Min-Cut Theorem *The maximum flow through a single commodity capacitated network from a source node s to a sink node t equals the value of the minimum s,t - cut.*

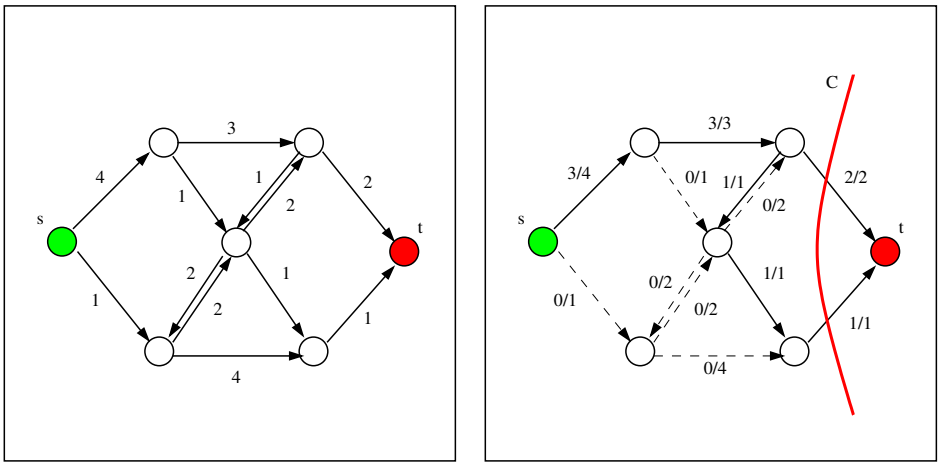


Figure 4.1: A flow network, a maximum flow and a minimum weight cut

The example in figure 4.1 shows a flow network with exactly one source s and sink t. On the left graph, the numbers denote the capacities of the corresponding edges. The flow on the right is a Maximum Flow through the network. Also a minimum cut C is given in the example with $\sum_{e \in C} cap(e) = 3$. Applying the Max-Flow-Min-Cut Theorem we know that the maximum flow from s to t equals 3.

Chapter 5

The Min - Cut - ILP - Formulation

5.1 Cut - Based - Methods

Nowadays many combinatorial problems have been formulated as (integer) linear programs which led to great results. However, the transformation of a given problem into inequalities is not always easily done and often various possibilities exist for achieving the same results which makes finding the best formulations often quite hard.

The idea behind solving combinatorial problems using LP solvers is to enforce the solver to produce only valid solutions using various constraints. The optimality of a given solution is then implied by the fact that the LP solver only produces optimal solutions under the given constraints.

For the PCST as well as for the ST, there exist various types of constraints that define valid solutions. Many different variants of ILP formulations for the PCST as well as a comparison among them can be found for example in [27].

Figure 5.1 shows an example of an invalid solution to a small ST instance. As one can see, nodes 5 and 6 are not connected to the tree which clearly invalidates the solution. One approach to formulating the ST as an ILP is to model the connectivity between nodes as a continuous flow going out of one specified root node r. In those flow based approaches every other customer node has to be connected to the root node. This induces a tree on the original graph that is always a valid solution for the ST. In those approaches the capacity of every edge e is defined as 0 if e is not part of the solution and $C > 0$ otherwise. If a flow $f > 0$ from r to s exists for every node s in that tree, a valid solution to the problem has been found.

For the PCST, the approach is somewhat similar. The only difference is, that the set of nodes in the solution is determined by the actual solution to the LP relaxation. For ever node s in the set $S' \subseteq S$ of potential customers

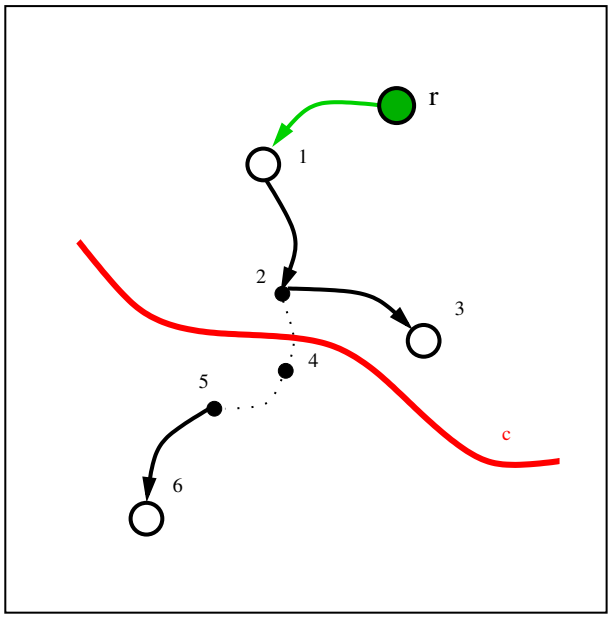


Figure 5.1: An example for an invalid solution for the ST. Bold nodes denote nonterminal vertices, circles represent customer nodes. A violated cut c is given.

that is part of the actual solution, a flow outgoing from r has to be found using only edges that are part of the solution too. If this condition is met, a valid solution tree has been found.

In linear programs representing valid solution using network flows normally there exist flow conservation constraints. Those constraints state, that for every node the outgoing flow has to be less than or equal to the incoming flow. In Figure 5.1, node 5 would violate this inequality because the ingoing flow is zero while there is a unit of flow going out to node 6.

There exist a lot of well known variants of flow based ILP formulations, the method presented in this thesis however uses a different approach. According to the Max-Flow-Min-Cut theorem (see Section 4.2), *the maximum flow through a single commodity capacitated network from a source node s to a sink node t equals the value of the minimum s,t - cut.* This means, that instead of ensuring that a maximal flow greater than some value exists from the root node r to every customer node, we can also enforce that the minimal s, t - cut between r and the customer node is not lower than that value. An example cut (c) is also given in figure 5.1. Note that the minimal cut is not uniquely defined, another valid minimal cut would cross the edge between nodes 4 and 5. The sum of flow along the edges crossed by this cut is zero (as expected because of the Max-Flow-Min-Cut theorem and the fact that there is no flow from r to node 6). The constraint that would have

to be inserted to invalidate the given solution should state, that the sum of the edges, crossed by the cut has to be greater or equal to one if node 6 is in the solution tree.

The problem using this approach is, that there exists an exponential number of possible s, t - cuts. Therefore a clever cut separation procedure has to be used to iteratively find possible solutions. In the presented method, the LP solver starts without any cut constraints which may of course lead to invalid solutions. Therefore every solution is checked for violated cuts and if such cuts are found, the appropriate constraints are generated, added to the problem and the process starts over from re-optimizing the problem. If no such cuts can be found, the solution is optimal and feasible for the LP relaxation and is therefore accepted.

Note that in this section it is assumed, that there exists a root node r that is always part of every solution. In Section 5.2 we show, how this can always be achieved by transforming the problem. In Section 5.3 we show how through further transformations of the problem we can resolve some problems that arise when solving LP relaxations of the problem.

In the following sections the exact formulation of the PCST and the ST as an ILP is described. Details on the cut separation procedure are given in Section 6 and the iterative algorithm is then presented in Section 6.4.

5.2 Transformation to the Rooted Steiner Tree Problem

As the presented formulation assumes that there exists a root node s which will always be part of any solution we have to ensure, that this node actually exists. For the Steiner Tree Problem this assumption always holds as every customer node has to be part of every solution. We can easily define any customer node as the root node r. For the prize collecting Steiner Tree Problem this assumption is not true. Therefore we apply the following transformation to introduce an artificial root node.

Suppose we are given a graph $G = (V, E)$ which does not contain a root node r. We define the auxiliary graph G^* by adding an artificial root node r as follows:

$$\begin{aligned} G^* &= (V^*, E^*) \\ V^* &= V \cup \{r\} \\ E^* &= E \cup \{(r, v) : \ v \in R\} \end{aligned}$$

We define the new cost function $c^* : E^* \to \mathbb{N}_0$ as

$$c^*(e) = \begin{cases} c(e) & \text{if } e \in E \\ 0 & \text{if } e = (r, v) \text{ and } v \in R \end{cases}$$

By adding one edge from the artificial root node to every customer node any customer node may now be the root node of the original graph G by connecting the artificial root node r to it. We have to ensure however, that for every solution at most one customer node is connected to r.

An alternative solution Clearly, the method presented above is not the only way of making sure that one vertex is definitely part of every solution tree. Actually some minor problems arise when the solution presented above is applied to solve instances of the PCST. Therefore we tried another algorithm, that exploits the fact, that all solutions which only differ by choosing another customer node as the root node are identical. The main idea is, that if we solve the problem setting one customer node i as root, we can be sure that, if we select every other customer that is present in the solution as root, we will get the same result. Therefore we can eliminate all the customer nodes present in the obtained solution from the set of possible roots and start over with solving the reduced problem until the set of possible roots is empty.

Note also that there can not be another globally optimal solution containing the actual root node i because we would have found the equivalent solution with root node i. Therefore we can remove node i from the set of nodes V. We now give an algorithm that describes this procedure.

1. $Roots = R$
2. Select i from the set of possible roots $Roots$
3. $Roots = Roots \setminus \{i\}$.
4. Solve the rooted PCSTP to optimality, where i is the root, obtaining solution S.
5. $Roots = Roots \setminus \{j\}$ for all $j \in S \cap R$.
6. Remove i from V.
7. Goto 2 until $Roots = \emptyset$.
8. Return the best found solution S.

Although for every instance several PCSTs have to be solved when applying this method, it works generally quite well. In our experiments, however, we mainly used the method of inserting an artificial root. The problems that arise and a solution to them are discussed in Section 5.8.

5.3 Transformation to the Steiner Arborescence Problem

While solving LP relaxations of the ST and PCST we encountered serious problems with so called *0.5 flow circles*. The problem is that when we tried to apply our formulation to undirected graphs, the formulation allows circles of 0.5 flow to be contained in valid solutions. This makes the CPLEX solver branch on those variables to enforce the integrality of the corresponding variables what slows down the running time of the algorithm significantly. An example of an 0.5 flow circle is given in Figure 5.2. It is easy to see that every possible s, t - cut, like the given example cut C, has a value of one. This would mean, that node t is connected to r in a valid way. As we can see, some edges in the example get a value of 0.5 assigned. This means that the given solution can never be a valid solution to the underlying integer linear program. When the ILP solver starts the branching process, this solution would be marked as illegal. This process however is very time consuming which is the reason why we introduced another transformation.

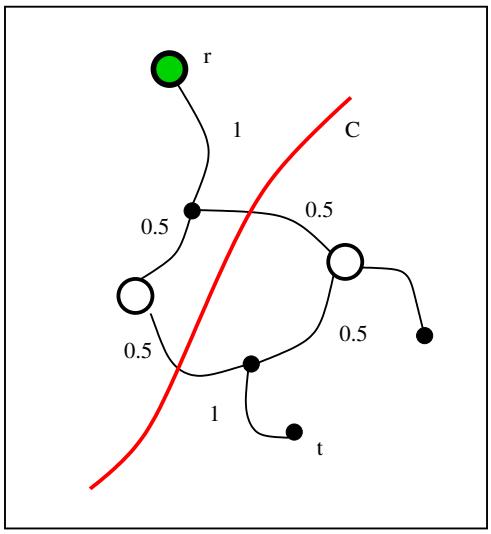
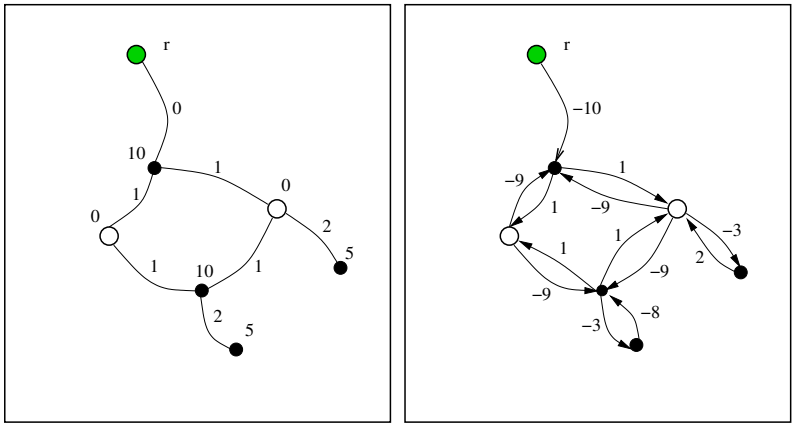


Figure 5.2: Example for an 0.5 flow circle

To overcome this problem we have chosen to transform our instances into the *(prize collecting) Steiner Arborescence problem (SA)*, which is the directed version of the ST. The difference to the ST is that in the Steiner Arborescence problem the prize from every node w is transferred to the edges (v, w) $\forall (v, w) \in E$. If we enforce that every possible solution is a tree, every node has an indegree of at most 1. This means that the prize for every node is collected at most once and therefore the problem is equivalent to the ST. Note that this transformation allows edge costs to be negative. If

no negative edge costs exist, the solution to an SA instance is only the root node r.



(a) An ST instance, nodes have their corresponding prizes, edges the costs assigned

(b) The transformed instance, the edges are now directed and the prizes have been subtracted from the costs of the incoming edges

Figure 5.3: An example for the transformation to a Steiner Arborescence

The transformation We transform the input graph $G = (V, E)$ into a bidirected graph $G = (V', A)$ by introducing (v, w) and (w, v) into A for every edge $(v, w) \in E$. Additionally, if we solve an instance of the PCST and have not done this before, we introduce an artificial root node r and connect it to every customer node $v \in R$.

$$V' = V \cup \{r\}$$

$$A = \{(r, v) \mid v \in R\} \cup \{(v, w), (w, v) \mid \forall (v, w) \in E\}.$$

The cost of each arc (v, w) is now given by $c'(v, w) = c(v, w) - p(w)$. If a new root node has been inserted, the costs for the newly inserted arcs going out from r are defined as $c'(r, v) = -p(v)$. For simplicity we will not refer to $c'(v, w)$ in the following sections but rather redefine $c(v, w)$ as $c(v, w) = c(v, w) - p(w)$.

For the ST there exists a set of target vertices T, that have to be part of every valid solution. In this case the Steiner Arborescence is defined as follows.

Definition 7 Steiner Arborescence for the ST *Given a digraph* $G =$

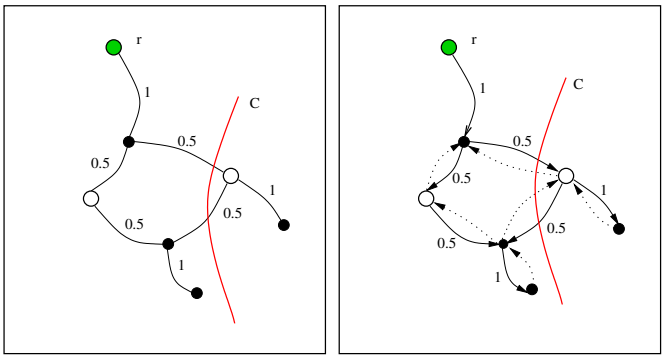
(V, A) A Steiner Arborescence (SA) is a partial digraph $T = (V_T, A_T)$ of G such that:

- $R \subseteq V_T$
- *all the vertices $v \in V_T$ have indegree equal to one, the root node r has indegree 0.*
- *for each node $v \in V_T \setminus \{r\}$, there exists a directed path from the root to v.*
- $\sum_{ij \in A_T} c'_{ij}$ *is minimal*

If we want to solve an instance of the PCST the set of customer nodes in the solution is not defined. In this case we have to search for the partial digraph $T = (V_T, A_T)$ of G such that:

Definition 8 Steiner Arborescence for the PCST *Given a digraph $G = (V, A)$ A Steiner Arborescence (SA) for the PCST is a partial digraph $T = (V_T, A_T)$ of G such that:*

- *all the vertices $v \in V_T$ have indegree equal to one, the root node r has indegree 0.*
- *for each node $v \in V_T \setminus \{r\}$, there exists a directed path from the root to v.*
- $\sum_{ij \in A_T} c'_{ij}$ *is minimal*



(a) (b)

Figure 5.4: An invalid solution before and after the transformation

An example for a transformation is given in Figure 5.3. Note that after this transformation the prize function p is not needed any more. The new objective function for a transformed PCST is

$$\text{minimize} \sum_{e \in E_T} c(e)$$

As we can see in Figure 5.4, the problems with the given 0.5 circles have disappeared. In Figure 5.4(a), there is no way to tell, that the solution is invalid. The given example cut has a value of 1 and is therefore valid for the instance. In Figure 5.4(b), our algorithm can detect, that one of the edges of cut C goes backwards and therefore the value of this edge has to be subtracted. The overall value of this cut is 0 and this determines that the solution is definitely infeasible.

In [27] the authors show, that the directed formulation dominates the undirected one. We can prove, that a large class of 0.5 circles can not be part of an optimal solution any more so by using this extension to the problem we were able to solve many of the benchmark instances of our test set without the need of branching.

5.4 The Formulation

After applying all necessary transformations, our goal is to construct an ILP formulation that corresponds to the instance we want to solve. For simplifying this task, we introduce for every node v a virtual self loop (v, v) with $c(v, v) = 0$. When we look at a solution to this newly generated problem, we can see that every single vertex v except the root node r has an indegree of exactly 1 which makes constructing inequalities a bit easier. Figure 5.5 shows a solution with added self loops. Note that with this extension every node has exactly one incoming edge in the solution graph.

For setting up an integer linear program we introduce for every edge (v, w) a corresponding integer variable $x_{vw} \in \{0, 1\}$. We interpret a solution to the integer linear program in the following way. If $x_{vw} = 1$, the edge (v, w) is taken into the solution tree. If the self loop of a node is in the solution, this node cannot be connected to the solution tree. Therefore a node v is only part of the solution if the corresponding x_{vv} has a value of 0 in the ILP solution. The root node r is part of every solution.

The Min Cut ILP formulation for the PCST is given in the following section.

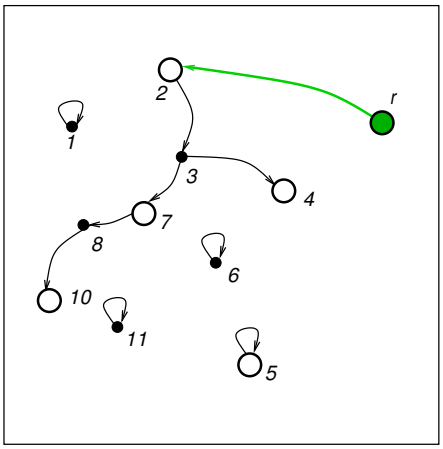


Figure 5.5: A solution to a small PCST instance with self loops added

5.5 Applying the Min Cut formulation to the prize collecting Steiner Tree Problem

As seen in Section 5.3 the objective function of the transformed problem is

$$\min \sum_{e \in A_T} c(e)$$

Because $e \in A_T$ corresponds to $x_e = 1$ in our ILP formulation, the objective function for the ILP is

$$\min \sum_{e \in A} c(e) x_e \, . \tag{5.1}$$

The following set of constraints describes the problem completely:

$$\sum_{e \in \delta(W)} x_e \geq 1 - x_{vv}, \; \forall \, W \subseteq V \setminus \{r\}, \; \forall v \in W \cap R \tag{5.2}$$

$$\sum_{(v,w) \in A} x_{vw} = 1 - x_{ww}, \forall w \in V, w \neq r \tag{5.3}$$

$$\sum_{(r,v) \in A} x_{rv} \leq 1 \tag{5.4}$$

$$\sum_{(v,r) \in A} x_{vr} = 0 \tag{5.5}$$

$$x_{vw} \in \{0,1\}, \forall (v,w) \in A \tag{5.6}$$

where $\delta(W) = \{e = (u, v) \in E \mid u \in W \wedge v \in V \setminus W\}$ represents an cut defined by the set W.

The so called *min-cut inequalities* (5.2) describe that for every subset of vertices that includes a vertex from R but not r there must be an edge that leads inside the set. Every node has to have an indegree of 1 in the solution. This is modeled by inequalities (5.3), which state that from all incoming edges to a node v and the virtual self loop (v, v) only one may be used for the solution.
Constraint (5.4) states that only one edge outgoing from the root node can be part of the solution. Note that this equality is only applied if an artificial root node has been introduced, otherwise we introduce constraint (5.5) which states, that in the solution no edges should be going into r.
The remaining constraints determine the domain of the integer variables x_e.

In our approach we used additional constraints that are not necessary for the definition of the problem but that tighten the formulation. The inequalities we used are:

$$\sum_{(v,w)\in A} x_{vw} \leq \sum_{(w,v)\in A} x_{wv}, \forall v \in V \setminus R, v \neq r \tag{5.7}$$

and

$$x_{vw} + x_{wv} \leq 1 - x_{vv}, \forall v \in V, \forall (v, w) \in A, v \neq r \tag{5.8}$$

Inequalities (5.7) are called *flow balance constraints*. Those constraints guarantee that the number of edges going into one non customer node v is less than or equal the number of edges going out of it.
Inequalities (5.8) state, that every edge (v, w) can only be part of a solution when the adjacent node v is part of the solution too.

5.6 Applying the Min Cut Formulation to the Steiner Tree Problem

The formulation given above can be used to solve the prize collecting Steiner Tree problem. If we want to apply this formulation to the Steiner Tree problem too, we have to change it slightly. One easy method would be to introduce the following set of inequalities that demand that every terminal node is part of the solution.

$$x_{vv} = 0, \forall v \in R$$

Applying this approach would result in a even bigger linear program than the one needed for solving the PCST because of the additional constraints.

Therefore the approach we took is to eliminate all x_{vv} variables from the formulation above for customer nodes. This leads to the following ILP:

$$\min \sum_{e \in A} c(e) x_e \ . \tag{5.9}$$

subject to

$$\sum_{e \in \delta(W)} x_e \geq 1, \ \forall \, W \subseteq V \setminus \{r\}, \ W \cap R \neq \emptyset \tag{5.10}$$

$$\sum_{(v,w) \in A} x_{vw} = 1 - x_{ww}, \forall w \in V \setminus R, w \neq r \tag{5.11}$$

$$\sum_{(v,w) \in A} x_{vw} = 1, \forall w \in R, w \neq r \tag{5.12}$$

$$\sum_{(v,r) \in A} x_{vr} = 0 \tag{5.13}$$

$$x_{vw} \in \{0,1\}, \forall (v,w) \in A \tag{5.14}$$

As before we insert additional constraints to strengthen the formulation. For solving the PCST we have to split up inequalities (5.8) because for nodes $v \in R$, there does not exists a corresponding x_{vv}. The resulting constraints are:

$$\sum_{(v,w) \in A} x_{vw} \leq \sum_{(w,v) \in A} x_{wv}, \forall v \in V \setminus R \tag{5.15}$$

$$x_{vw} + x_{wv} \leq 1 - x_{vv}, \forall v \in V \setminus R, \forall (v,w) \in A, v \neq r \tag{5.16}$$

$$x_{vw} + x_{wv} \leq 1, \forall v \in R, \forall (v,w) \in A, v \neq r \tag{5.17}$$

5.7 Analysis

The presented formulations have both a polynomial number of variables ($O(|E| + |V|)$ for the PCST, $O(|E| + |R|)$ for the ST). However the number of constraints, especially the number of *min-cut inequalities* (5.2) and (5.10), is exponential. Entering all those constraints at once would let the number of non-zeros in the ILP matrix grow exponentially which would make the problem practically unsolvable. With a clever cut separation strategy where constraints are added during the solving process, the number of constraints needed to actually solve the problem to optimality may be much smaller. The strategy we used is presented in Section 6. In the next section we will present another class of inequalities that help to overcome some problems introduced by adding an artificial root node.

5.8 Asymmetry Constraints for the PCST

As stated before, introducing an artificial root node also poses some new problems. An artificial root node makes it possible to construct identical solutions that only differ by the customer node that is connected to r. Those solutions have equal objective values but as they all have to be found by the LP solver, the existence of those solutions can slow down the solving procedure significantly. An example for this problem is given in Figure 5.6.

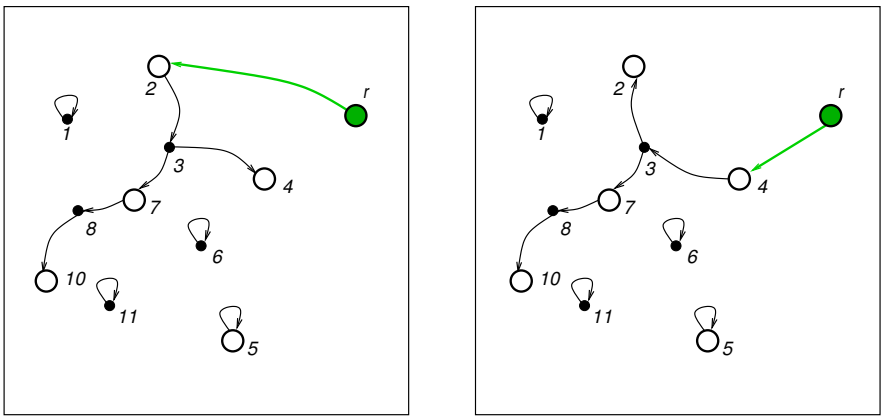


Figure 5.6: An example for equivalent solutions with different root nodes

To avoid this problem we introduced an arbitrary order on the customer nodes. We demand that in every solution tree the customer node with the highest index is connected to the artificial root node. The following inequalities guarantee this asymmetry and are therefore called *asymmetry constraints*.

$$x_{vv} \geq x_{rw}, \ \forall v, w \in R, v < w, v, w \neq r \tag{5.18}$$

In most cases inserting those constraints brings a big speedup in solving the problem. Note that this class of inequalities has to be used only for solving the PCST because when applying the formulation to an ST instance, a customer node can be declared as root and no identical solutions can be found.

Chapter 6

Cut Separation

As stated before the number of *min cut inequalities* is too high to deal with all of them right from the start of the optimization process. Therefore the method presented here starts with a reduced ILP formulation. When the linear problem is set up, all the presented inequalities get added except constraints (5.2) or (5.10) depending on the type of the problem that has to be solved. An iterative process is started that consists in solving this reduced linear program, separating violated min cut constraints, adding them to the problem and restarting with solving the enhanced problem. The separation of violated inequalities is done by calculating a maximum flow from the root node r to one node v, that is considered part of the solution, on a flow network which is generated from the solution of a LP relaxation. If an ST instance has to be solved, all nodes $v \in R$ are considered part of the solution. For the PCST, all nodes v where the value of the corresponding variable x_{vv} is lower than 1 are considered part of the solution. We now construct a flow network by using all nodes and arcs from G and giving each arc a capacity equal to the value of the corresponding variable in the LP solution. This flow network is called the *support graph*. From the obtained maximum flow a corresponding minimum r, v cut can be found quite easily using depth first search. Based on the calculated minimum cut various violated constraints may be derived.

Section 6.1 describes this process in more detail. The set of constraints that is actually added to the problem can be selected using various strategies. Some of them are presented in the following sections.

6.1 Calculating a Minimum s,t Cut

The first step for calculating a minimum s,t cut is to convert the solution obtained from the LP solver into a flow network. This can be done by defining a flow network $G^* = (V, E)$. The sets of vertices and arcs can be taken directly from the problem graph G. The supply function $sup : V \mapsto \mathbb{Z}$

can be defined as

$$sup(v) = \begin{cases} 1 & v = r \\ -1 & v = t \\ 0 & otherwise \end{cases}$$

and the capacity function $cap : E \mapsto \mathbb{R}$ as $cap(e) = \text{lpval}(x_e)$.

This flow network represents the actual solution to the LP relaxation of the problem. If the solution represents a valid solution tree for the LP relaxation of the ST or PCST, there exists a sufficient amount of flow from the root node r to every node v considered part of the solution. Note that this criteria does not imply that the solution is a valid solution to the original problem as some variables in the solution may still have fractional values.

For the ST, the amount of flow from r to every node $v \in R$ has to be equal to 1. In the case of the PCST, the sufficient amount of flow to the node v is defined as $1 - x_{vv}$. If the corresponding x_{vv} is 1, the node is not part of the solution so no flow has to be found.

If we can find a node v, that is part of the solution, for which no flow of sufficient quantity exists, we have found one ore more violated inequalities. So the first step in separating new constraints is calculating a maximum flow in this network.

Calculating a Maximum Flow The algorithm we chose for calculating the maximum flow is a push relabel implementation from Cherkassky and Goldberg [8]. The authors provide highly optimized c code which we slightly adapted to handle non integer capacity values. The push relabel algorithm with the combination of global relabeling and gap relabeling calculates a maximum flow which can then be converted in an s, t cut by applying a depth first search procedure.

Calculating a Minimum s,t Cut For transforming the maximum flow obtained into a minimum s, t cut we have to run a depth first search algorithm on the residual network of F. We define the residual capacity of each edge e in the residual network as $cap(e) - f(e)$ where $f(e)$ denotes the amount of flow that passes through edge e. We call an edge *saturated* if its residual capacity is 0. From the *max flow min cut theorem* we know, that a s, t cut containing only saturated edges in the residual network is always a minimum s, t cut.

Finding a cut in the residual network is quite easy. All we have to do is to start a depth first search procedure from the root node using only edges, that have a residual capacity > 0. While searching we mark all nodes that remain in the same component as the root node. Finally we scan through all edges and if the source node of this edge is marked but the target node is not, this edge is part of the resulting s, t cut.

Figure 6.1 gives an example for how this process works. Figure 6.1(a) sketches a flow network that is based on a solution for an LP relaxation. In Figure 6.1(b) a maximum flow of 0.5 has been calculated from the root node S to a vertex T. The following figure shows the residual network that we obtain by subtracting the flow over each edge from the capacity of the corresponding edge. Finally, the last figure shows which nodes have been marked by the depth first search procedure and also gives the obtained cut through saturated edges with marked source and unmarked target nodes. The value of this cut has to be 0.5 too.

Using this method exactly one cut will be found for every node. This allows us to add only at most one additional constraint for every node to the linear program. There exist strategies that are able to find many different violated inequalities without having to re-optimize the linear program which leads to many violated min cut constraints at once. Some of those strategies are presented in Section 6.3.

6.2 Deriving inequalities from minimum cuts

If we have constructed a minimum cut C from a maximum flow from the root node to node v we can quickly derive a violated constrained from it. As stated in Section 5.5 we have to insert inequalities of the following form if we want to solve the prize collecting Steiner Tree problem.

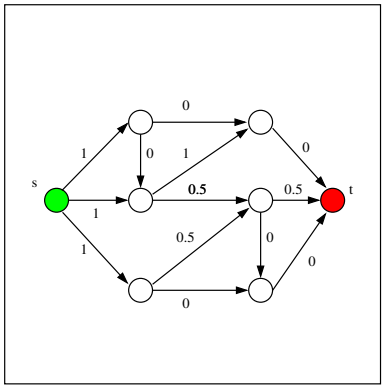
$$\sum_{e \in C} x_e \geq 1 - x_{vv}$$

This constraint states, that if node v is part of the solution, there must not be a minimum cut lower than 1 between the root node and v. For instances of the Steiner Tree problem, the x_{vv} values have been eliminated (See Section 5.6) and the constraint we have to add to the problem is

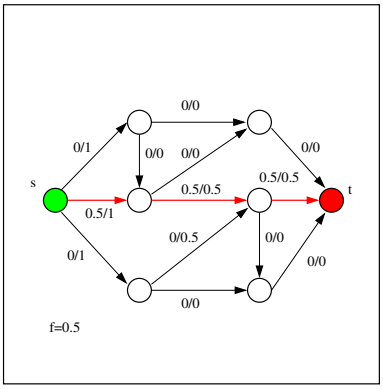
$$\sum_{e \in C} x_e \geq 1$$

6.3 Cut Generating Strategies

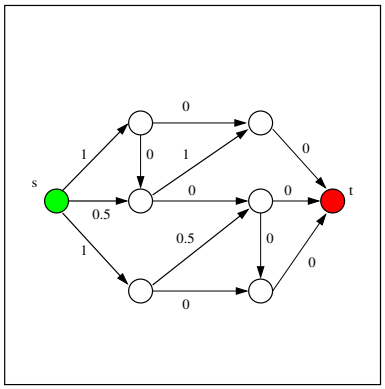
As outlined in the previous section, normally adding one violated inequality to a linear program is simply not sufficient. Although the quality of the obtained solution increases with every added cut the re-optimization time between the separation steps may just be to long to give appealing results. In the following sections three strategies are presented that can be used to generate more cuts in every iteration. However, the blind adding of cuts does not yield very good results either. When too many cuts are inserted in an linear program, the optimization time may increase very quickly. Finding a suitable number of cuts to be added turns out to be quite difficult.



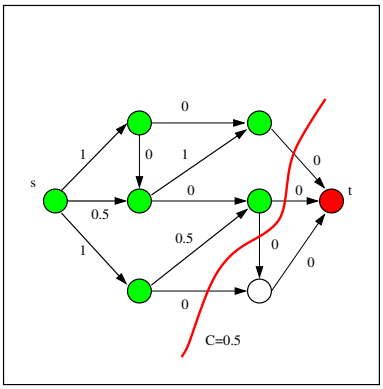
(a) A flow network generated from a solution to a LP relaxation. Note that some edges have a capacity of 0.5



(b) A maximum flow from s to t has been found. The value of the flow is 0.5



(c) The resulting residual network



(d) Nodes in the root component have been marked and a minimum cut C has been found

Figure 6.1: The process of finding a violated cut

6.3.1 Orthogonal Cuts

The idea of using *Orthogonal Cuts* is to increase the diversity of the added constraints. With the basic separation strategy presented above, in the graph in 6.2(a) the same cut $C1$ will be found for every node $n1$ to $n3$. As those cuts are very similar only the variable of one edge has to be set to one from the LP solver to satisfy all of them. To overcome this problem the

residual capacities of the edges that are already part of one cut are changed. If the algorithm sets the residual capacity of an edge to any value ≥ 1 there will never be a minimum cut going through this edge. This means that this edge will not be part of any other cut before the problem is re-optimized and the residual capacities are recalculated. As a consequence when the LP solver sets the corresponding variable of this edge to 1, it satisfies at most one cut. This leads to a faster improvement of the solution quality. There are various ways to chose edges from a cut that get their residual capacity adjusted. Our approach uses two alternatives:

- Set the residual capacities of all edges of the cut to 1
- Set the residual capacities of one random edge of the cut to 1

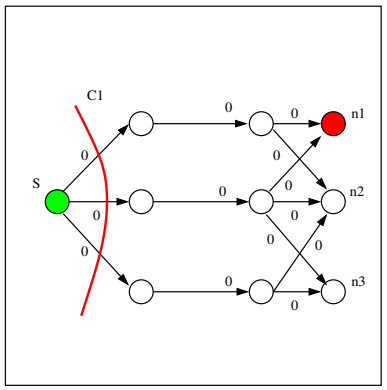
The results are shown in Figure 6.2 and Figure 6.3. If we applied the basic strategy, the generated cuts for all nodes would look like $C1$ from Figure 6.2(a). The improvement presented in this section gives cuts with more diversity. When using the first strategy the results are given in Figure 6.2. The three cuts found still share one edge so the amount of diversity is not optimal. The results for the second strategy are given in Figure 6.3.

Note that the results depend heavily on the order of the processed nodes. We tried to exploit this fact by presorting the nodes according to the value of their corresponding x_{vv} values or to the difference between the maximum flow going into a node and their x_{vv} values. It turned out that processing the nodes in a random order gives the best overall results. Note that processing the nodes every time in the same order slows down the solving procedure significantly and should therefore not be used.

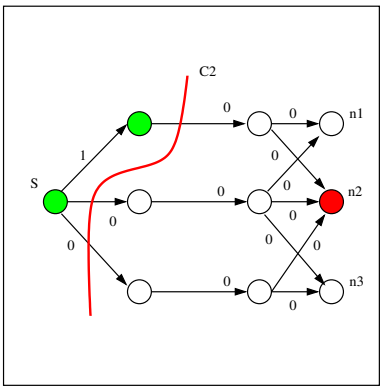
6.3.2 Nested Cuts

Is this section we will discuss another two extensions to the cut generating procedure that can be exploited when using *orthogonal cuts*. The idea behind this strategies is to generate more than one cut for every node or to use one cut for more than one customer node.

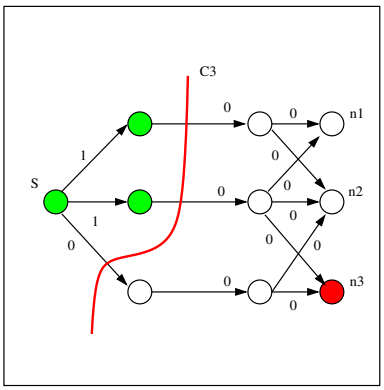
Generating more than one cut for every node can be done by applying a similar strategy to the one presented for generating *orthogonal cuts* in Section 6.3.1. This time however, the process is repeated for one node until there exists a maximum flow from the root node to this node of amount 1. The generation of more than one cut for every node may only be combined with the methods of Section 6.3.1. Every iteration of this process the residual capacity of at least one edge has to be adjusted or unlimited identical cuts will be generated and the process will not terminate. As before, various strategies for adjusting the residual capacities are possible. Note that we



(a) With the basic strategy all cuts would be similar



(b) Using *Orthogonal Cuts* a different cut is found for node $n2$

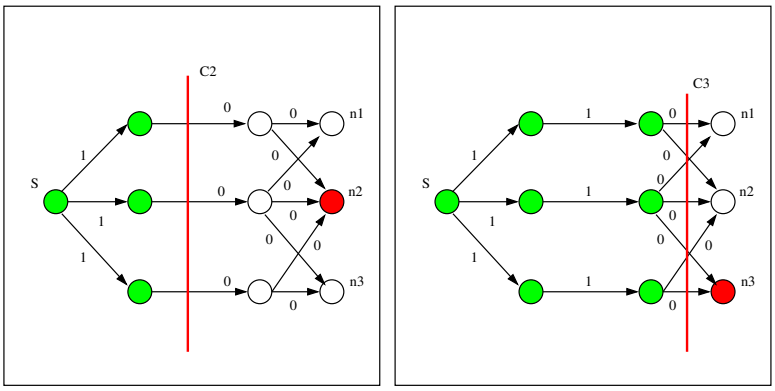


(c) The third cut generated for $n3$

Figure 6.2: *Orthogonal Cuts* found when using the first adjusting strategy

do not have to recalculate the maximum flow to the target node when we adjust the capacity of the edges. We may simply mark the target nodes of the adjusted edges as part of the root component and continue with finding the edges that form the next minimum cut.

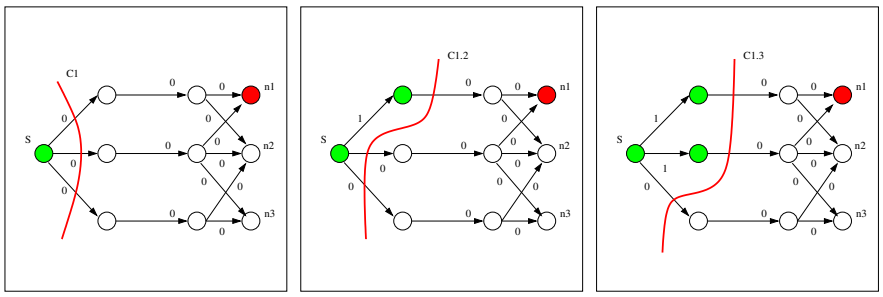
Figure 6.4 shows, how *nested cuts* are generated. The difference to the cuts generated in Figure 6.2 is that this time the discovered cuts are turned into constraints that state that the cuts have to be satisfied if node $n1$ is part of the solution. Opposed to that, plain *orthogonal cuts* generate one cut C_n for every node n_n.



(a) The resulting cut for node $n2$

(b) The resulting cut for node $n3$

Figure 6.3: *Orthogonal Cuts* found when using the second adjusting strategy



(a) (b) (c)

Figure 6.4: *Nested Cuts*

Generating one cut for more than one customer node Another approach to finding more cuts is to reuse a discovered cut for more than one terminal node. After a cut minimum cut C has been found by the algorithm, this cut divides the graph into two components. The max flow min cut theorem states that for all customer nodes that are in the opposite component of the root node r there cannot exist a flow from r to this node which is greater than the value of the minimum cut. Therefore we may look at all customer nodes that are unmarked after the depth first search procedure and verify if $1 - x_{vv} \leq f$. For every node where this condition is not met, we may introduce the inequality as described in Section 6.2.

If we want to find a solution to an instance of the ST, we simply have to scan through the set of customer nodes and add for all unmarked vertices the according constraint to the linear program.

6.3.3 Generating back-cuts by calculating an inverse flow

The last approach we used in our algorithm to generate more cuts works by calculating inverse flows in the residual network. The strategy presented in the previous sections splits up the set of nodes into two parts by applying a depth first search algorithm to the residual network after the maximum flow algorithm terminates. This approach works quite well as long as the minimum cut is uniquely defined. In this case we end up with the root component, which is the connected component containing the root node, the minimum cut and the target component, which is the connected component containing the target node. In many cases, there exist several additional saturated edges in the residual network to the ones building the cut. In this case, the target component contains also nodes that are not connected to the target node. Figure 6.5 shows an example for a remaining component that contains saturated edges as well.

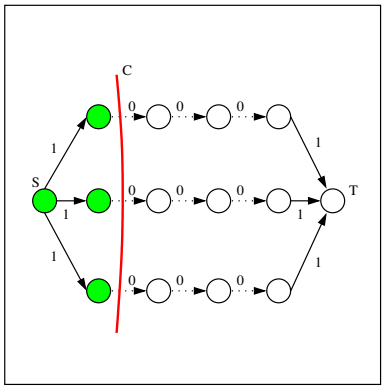


Figure 6.5: A not unique minimum cut

All the unmarked nodes are in the “remaining” component. Because the minimum cut is not uniquely defined the target component consists of too many nodes. Only the four rightmost nodes form a connected component but this is not detected by the algorithm. To effectively distinguish the target component from non reachable nodes, we use a second call to the depth first search procedure. This time however the residual edges are turned around before the invocation of the algorithm. This way the algorithm finds all nodes, where a flow to the target node would be possible. The result of this procedure can be seen in Figure 6.6.

The components are now correctly defined although there exist several possible minimum cuts. The additional back-cut $C2$ which is found along

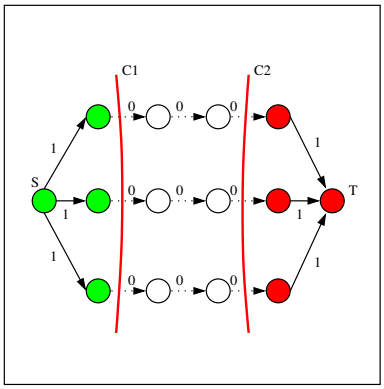


Figure 6.6: A correct partitioning

the border between the target component and the unmarked nodes usually gives constraints that improve the quality of a solution a lot. Of course this approach can be combined with the ideas of *orthogonal cuts* and *nested cuts*.

6.4 The algorithm

In this section we will give an overview of the algorithm that is used to generate violated constraints for the LP relaxation of the problem. Two variants of the algorithm are shown. Algorithm 1 shows the cut generation procedure when *nested cuts* are disabled. If both approaches presented in Section 6.3.2 are used, the procedure works as shown in Algorithm 2.

Algorithm 1 The algorithm without using *nested cuts*

```
repeat
  Solve the LP relaxation of the problem
  Construct a flow network using the solution obtained
  if ST then
    nodes ← list of nodes v ∈ R
  else
    nodes ← list of nodes v ∈ R with x_vv < 1
  end if
  Permute nodes
  for all v ∈ nodes do
    Calculate a maximum flow f from r to v
    if f < 1 − x_vv then
      Mark root component using DFS
      Generate a minimum cut and derive the appropriate constraint
      Add the inequality to the problem
      if BACKCUTS then
        Mark target component using inverse DFS
        Generate a minimum cut and derive the appropriate constraint
        Add the inequality to the problem
      end if
      if ORTHOGONALCUTS then
        Update capacities of the flow network
      end if
    end if
  end for
until No new cuts were found
```

Algorithm 2 The algorithm when *nested cuts* are used

```
repeat
  Solve the LP relaxation of the problem
  Construct a flow network using the solution obtained
  if ST then
    nodes ← list of nodes v ∈ R
  else
    nodes ← list of nodes v ∈ R with x_vv < 1
  end if
  Permute nodes
  for all v ∈ nodes do
    Calculate a maximum flow f from r to v
    while f < 1 − x_vv do
      Mark root component using DFS
      Generate a minimum cut
      for all w ∈ nodes do
        if w is not marked then
          Derive the appropriate constraint for node w
          Add the inequality to the problem
          Update capacities of the flow network
        end if
      end for
      if BACKCUTS then
        Mark target component using inverse DFS
        Generate a minimum cut and derive the appropriate constraint
        Add the inequality to the problem
        Update capacities of the flow network
      end if
      Recalculate f
    end while
  end for
until No new cuts were found
```

Chapter 7

$\{0, \frac{1}{2}\}$-Chvátal-Gomory Cuts

In this section we will show how we improved our solving procedure by introducing a second class of cuts, so called $\{0, \frac{1}{2}\}$-*Chvátal-Gomory Cuts* to the proposed linear programs, which were introduced by Caprara and Fischetti ([7]). As opposed to the cuts presented in the previous sections, this class of inequalities is not needed to model the ST or PCST instance as an integer linear program. The only purpose of those cuts is to reduce the solving time needed by the CPLEX optimizer by cutting the polyhedron which is already defined by the inequalities generated before.

The basic idea behind Chvátal-Gomory Cuts is to combine multiple constraints to generate a new one and applying a rounding procedure to the right hand side of this inequality. Under some circumstances the newly generated inequality is still a valid constraint for the integer linear program but cuts away some fractional solutions from the polyhedron of the LP relaxation of the problem. In our solving procedure we added $\{0, \frac{1}{2}\}$-Chvátal-Gomory Cuts, a special case of Chvátal-Gomory Cuts, to the LP relaxation by using a framework provided by Andreello, Caprara and Fischetti ([2]). The used LP solver, CPLEX 8.1, is capable of separating Chvátal-Gomory Cuts when solving integer linear programs. As the separation of those cuts is not trivial and the overhead by introducing too many cuts of this form would be enormous, the optimizer will never find and insert too many Chvátal-Gomory Cuts. This and the fact that the cuts generated by the framework generally seem to be of higher quality than the ones generated by CPLEX lead us to the strategy to combine both families of cuts to get better results.

The next section give a short overview on Chvátal-Gomory Cuts and the special case of $\{0, \frac{1}{2}\}$-Chvátal-Gomory Cuts. How those cuts can be separated is outlined in Section 7.2. Information on the framework and on parameter settings we used are given in Section 7.5.

7.1 Definition of $\{0, \frac{1}{2}\}$-Chvátal-Gomory Cuts

Given an integer linear program $Ax \leq b$ where $A = (a_{ij})$ is an $m \times n$ integer matrix and b an m-dimensional integer vector we let $P = \{x \in \mathbb{R}^n : Ax \leq b\}$ and $P_I = \text{conv}\{x \in \mathbb{Z}^n : Ax \leq b\}$. A Chvátal-Gomory Cut (CG cut) is a valid inequality for P_I of the form

$$\lambda^T Ax \leq \lfloor \lambda^T b \rfloor$$

where $\lambda \in \mathbb{R}^m_+$ is such that $\lambda^T A \in \mathbb{Z}^n$. We define a $\{0, \frac{1}{2}\}$-Chvátal-Gomory Cut as a CG cut where $\lambda \in \{0, \frac{1}{2}\}^m$.
The *rank-1 closure* of P is defined as

$$P_1 = \{x \in P : \lambda^T Ax \leq \lfloor \lambda^T b \rfloor, \text{ for } \lambda \in [0,1)^m \text{ such that } \lambda^T A \in \mathbb{Z}^n\}$$

The corresponding polyhedron obtained by intersecting P with the half-spaces induced by all possible $\{0, \frac{1}{2}\}$-cuts is defined as

$$P_{\frac{1}{2}} = \{x \in P : \lambda^T Ax \leq \lfloor \lambda^T b \rfloor, \text{ for } \lambda \in \{0, \frac{1}{2}\}^m \text{ such that } \lambda^T A \in \mathbb{Z}^n\}$$

It is easy to see that $P_I \subseteq P_1 \subseteq P_{1/2} \subseteq P$. This states that if all possible CG cuts were used by the optimizer, the class of $\{0, \frac{1}{2}\}$-*Chvátal-Gomory Cuts* would already be included. Practically the separation of those cuts is quite hard and therefore concentrating on $\{0, \frac{1}{2}\}$-cuts gives better results in some cases.

7.2 The separation of $\{0, \frac{1}{2}\}$-Chvátal-Gomory Cuts

In [7] the authors showed that CG cuts can be obtained in the following way. Let $\mu \in \mathbb{Z}^m_+$ and $q \in \mathbb{Z}_+$ be such that $\mu^T A \equiv 0 \mod q$ and $\mu^T b = kq + r$ with $k \in \mathbb{Z}$ and $r \in \{1, ..., q-1\}$. Then $\mu^T Ax \leq kq$ is a valid inequality for P_I.

This inequality can be written as $\mu^T(b - Ax) \geq r$, hence a given x^* violates $\mu^T Ax \leq kq$ if and only if $\mu^T(b - Ax^*) < r$. Further they state that it is sufficient to consider only multipliers $\mu_i \in \{0, ..., q-1\}$ as a larger μ_i leaves the $\mod q$ arithmetic unchanged but decreases the violation. This is also the reason why for a given slack vector $s^* = b - Ax^*$ the violation only depends on $(A, b) \mod q$.

The special case of $\{0, \frac{1}{2}\}$-cuts arises for $q = 2$ which implies $r = 1$. If we define the *binary support matrix* $\overline{Q}$ of an integer matrix $Q = (q_{ij})$ as

$$\overline{Q} = (\overline{q}_{ij}) = Q \mod 2$$

i.e., $\overline{q}_{ij} = 1$ if q_{ij} is odd and $\overline{q}_{ij} = 0$ otherwise, the problem of separating $\{0, \frac{1}{2}\}$-cuts ($\{0, \frac{1}{2}\}$-SEP) can be written as

Definition 9 *$\{0, \frac{1}{2}\}$-SEP: Given $x^* \in P$, solve* $\min\{{s^*}^T \mu : \mu \in F(\overline{A}, \overline{b})\}$, *where*

$$s^* = b - Ax^* \geq 0$$

and

$$F(\overline{A}, \overline{b}) = \{\mu \in \{0,1\}^m : \overline{b}^T \mu \equiv 1 \mod 2, \overline{A}^T \mu \equiv 0 \mod 2\}.$$

It is clear that there exists a $\{0, \frac{1}{2}\}$-cut violated by a given point x^* if and only if

$$\min\{{s^*}^T \mu : \mu \in F(\overline{A}, \overline{b})\} < 1.$$

Caprara and Fischetti show further that the problem $\{0, \frac{1}{2}\}$-SEP is equivalent to finding a *minimum weight binary clutter* by giving an algorithm to convert every instance of $\{0, \frac{1}{2}\}$-SEP into a corresponding instance of the minimum weight binary clutter problem (MW-BCP). This leads to the following corollary.

Corollary 1 *The recognition version of $\{0, \frac{1}{2}\}$-SEP is NP-complete*

There exist however important special cases where the $\{0, \frac{1}{2}\}$-SEP can be solved in polynomial time. The authors proved that if the matrix $\overline{A}$ has certain properties, the problem can be reduced to finding a minimum weight odd circle in an undirected multigraph. This problem is well known and can be solved in $O(n^3)$ time.

Among the various properties of the matrix A that allow the solving of $\{0, \frac{1}{2}\}$-SEP in polynomial time, the one that appears to be most important is that this conversion is possible if A has at most 2 odd entries in every row. If we define $M = \{1, ..., m\}$ and $N = \{1, ..., n\}$ the row and column indices of the matrix A and

$$O_i = \{j \in N : \overline{a}_{ij} = 1\} \text{ for all } i \in M$$

the following theorem can be derived.

Theorem 2 *$\{0, \frac{1}{2}\}$-SEP can be solved in polynomial time if $|O_i| \leq 2$ for all $i \in I$.*

Practically it will seldom be the case that a matrix has the properties demanded by Theorem 2. It is possible though to obtain a relaxation $P' = \{x \in \mathbb{R} : A'x \leq b'\} \supseteq P$ by weakening the system $Ax \leq b$ into $A'x \leq b'$ such that the $\{0, \frac{1}{2}\}$-SEP associated with $A'x \leq b'$ can be solved in polynomial time. The cuts obtained can then be used to optimize the original problem.

7.3 Weakening the system to obtain $\{0, \frac{1}{2}\}$-cuts

In [7] three methods of weakening constraints are proposed. For the application of those methods to the formulations presented in this thesis it is sufficient to consider the so called *L-weakening*. This is the simplest weakening that arises if the variables considered for the weakening have a lower bound of 0. In our formulations every single variable x has the same bounds

$$-x \leq 0$$

and

$$x \leq 1.$$

This means that the *L-weakening* can be applied to every variable and therefore this weakening is the only one needed.

L-weakening The relaxation of the system $Ax \leq b$ to $A'x \leq b$ is done to obtain a matrix A' which has the properties demanded by Theorem 2. Therefore the aim of the weakening is to replace the number of rows of the matrix A with more than 2 odd coefficients. This is done by replacing each constraint $\sum_j a_{ij}x_j \leq b_i$ with $|O_i| \geq 3$ by the constraints

$$a_{ih}x_h + a_{ik}x_k + \sum_{j \notin O_i} a_{ij}x_j + \sum_{j \in O_i \setminus \{h,k\}} (a_{ij} - 1)x_j \leq b_i$$

for all $\{h, k\} \in O_i$, $h < k$. This introduces $\binom{|O_i|}{2}$ inequalities for each row i which leads to an increase of the number of rows in the reduced system $A'x \leq b$ to $O(mn^2)$. To keep the overhead of having many rows in the system small, size reductions can be applied after the conversion.

Size reduction If a system is reduced by replacing rows with more than 3 odd coefficients by their corresponding *L-weakenings*, the resulting binary support matrix $\overline{A'}$ contains many identical rows. This leads to the fact that various vectors $\mu \in \{0, 1\}^m$ such that $\overline{b'}^T \mu \equiv 1 \mod 2$ and $\overline{A'}^T \mu \equiv 0 \mod 2$ can be found by selecting a different identical row. But as the problem of separating $\{0, \frac{1}{2}\}$-cuts tries to minimize $s^*\mu$ it is sufficient to keep only the row i with minimal slack $s^* = b' - A'x^*$ and removing all rows from A whose corresponding rows in the binary support matrix $\overline{A'}$ are identical.

7.4 Measuring the quality of $\{0, \frac{1}{2}\}$-cuts

Depending on the problem that has to be solved, the generating procedure presented above may generate a huge amount of $\{0, \frac{1}{2}\}$-cuts. This is a problem as inserting them all into the problem would lead to a big slowdown

in the solving procedure. It is essential to find a method that is capable to select from the vast amount of cuts a number of cuts that promise the best improvement to the actual LP solution when added to the problem. This measurement of the quality of cuts is however not that trivial.

Given a cut α i.e. $\alpha x \leq \alpha_0$ and an LP solution x^*, we can calculate the Euclidian Distance between x^* and the hyperplane induced by α ($\alpha x^* = \alpha_0$) as

$$\text{dist}(x^*, \alpha, \alpha_0) = \frac{|\alpha^T x^* - \alpha_0|}{\|\alpha\|}$$

This measurement is indeed a valid and quite reliable indicator for the quality of the cut α. The bigger the Euclidian Distance between the hyperplane and the point denoted by the solution is, the better we can assume the quality of the cut. Nevertheless this measurement has some drawbacks. If we have an $\alpha_i > 0$ and a corresponding $x_i^* = 0$, the numerator of $\text{dist}(x^*, \alpha, \alpha_0)$ is not affected in any way by α_i. Therefore we could set $\alpha_i = 0$ and produce a still valid and violated cut that has a greater Euclidian Distance. This newly generated cut is however strictly dominated by the original one and should therefore get a worse quality index.

Those drawbacks are not a big issue and therefore we define our measurement of efficacy of a cut as

$$\text{eff}(x^*, \alpha, \alpha_0) = \frac{\alpha^T x^* - \alpha_0}{\|\alpha\|}$$

Note that this value is negative if the cut is not violated and positive otherwise. We set up the rule that only cuts α are considered for adding whose $\text{eff}(x^*, \alpha, \alpha_0) \geq \textit{mineff}$. This parameter is adapted during the solving process and is described in Section7.5.

Similar Cuts Another problem that arises is that often very similar cuts are found. Adding more than one of those cuts does not improve the quality of the solution very much. We define that two cuts $\alpha x^* \leq \alpha_0$ and $\beta x^* \leq \beta_0$ are similar if they have almost the same efficacy and their hyperplanes are almost parallel. A measurement of the parallelity of two cuts can be given as

$$\text{par}(\alpha, \beta) = \frac{|\alpha^T \beta|}{\|\alpha\|\|\beta\|}$$

and so we can chose to only insert two cuts if $\text{par}(\alpha, \beta) \leq \textit{maxpar}$. Note that setting this parameter to 0 would introduce only *orthogonal cuts*. In our implementation we used a setting for this parameter that is close to one. This means that we insert very similar cuts too but we omit identical ones.

7.5 The algorithmical Framework

In this section we will describe the algorithmical framework we used for generating $\{0, \frac{1}{2}\}$-cuts. This framework was proposed by Andreello, Caprara and Fischetti in [2]. There are two reasons why we had use an additional framework to the ILP solver. One is that when cut generation methods are used, nearly always there is the problem of generating a lot of overhead by inserting too many cuts into the problem. In our case this problem arose too so we needed a clever way to handle cuts that were not yet inserted into the formulation. Secondly ILOG CPLEX does not provide any methods to efficiently handle constraints. For example it is not possible to remove constraints once they are added to a problem. All this makes the usage of a framework that extends the abilities of the solver a more than paying task. The various details as well as parameter settings we used are described in the following paragraphs.

The pool of cuts As we stated before, ILOG CPLEX as well as the *concert technology* do not provide methods to handle cuts efficiently. There exist data structures that may hold constraints in *concert technology* but when dealing with a large number of those cuts, this method is by far too slow. The used framework provides a buffer of cuts whose size is limited to 8000 cuts. For efficiency cuts are stored in this pool ordered by not increasing efficacy. If at any moment there are more than 8000 cuts queued in the pool, the buffer is truncated by removing the cuts with the least efficacy.

Number of cut generation attempts The number of attempts to generate $\{0, \frac{1}{2}\}$-cuts is also a very sensitive parameter. A too high number of tries would use a lot of calculating time without finding new cuts whereas a too low number would not find all the possible $\{0, \frac{1}{2}\}$-cuts and therefore increase the time spent in the LP solving process. The strategy proposed in this framework is to allow 5 attempts to generate cuts in the root node of the branch-and-cut tree. After that the $\{0, \frac{1}{2}\}$-separation procedure is only called at every 4th backtracking step. For our implementation we changed this parameter to try to generate cuts at the end of processing any node.

Maximum number of cuts As the used LP solver does not provide methods to remove cuts from an LP once they are inserted, we have to chose a rather conservative cut insertion strategy. There is a workaround to this problem by declaring cuts as local for the node of the branch-and-cut tree which are eventually removed when the whole subtree has been processed. In our implementation this workaround has not been used so we imposed a limit on the number of cuts added at every step of the procedure. The limits are

- Do not add more than 1000 cuts per cut generation process
- Do not invoke the cut generation procedure more than once per node
- Do not add more than 0.3 times the number of rows cuts in every node of the B&C tree except in the root node

This effectively limits the number of overall cuts added.

Adapting the *mineff* parameter As mentioned before, the *mineff* parameter is adapted during the solving process. This is due to the fact, that at the beginning it is quite easy to find violated cuts with a high efficacy. Later in the process as the LP relaxation gets tighter, the finding of cuts with high efficacy gets more difficult. We start with a *mineff* of 0.7. This means that a cut is considered candidate for adding if its efficacy is not less than 70% of the efficacy of the best cut generated. During the separation process we count the number of unsuccessful generation attempts and every 20 failures we reduce the *mineff* criteria.

Recombination A last parameter of the process is the selection of rows that are candidate for the combination for the generation of $\{0, \frac{1}{2}\}$-cuts. There are two possibilities.

- Select candidate rows from the inital LP
- Select candidate rows from the actual LP including $\{0, \frac{1}{2}\}$-cuts

In our approach we selected the first method. Additionally we have the choice to include cuts generated by the separation procedure in Section 6 into the set of candidate rows. We obtained better results when we excluded those cuts so in our approach $\{0, \frac{1}{2}\}$-cuts are only derived from the initial LP.

7.6 Conclusion

For some of the tested instances the addition of $\{0, \frac{1}{2}\}$-cuts was of crucial importance when optimizing the root node of the branch-and-cut tree. The second improvement presented in this thesis, *Local Branching*, is a method that optimizes the B&C process. This means that this method can only be applied after the root node. Secondly for applying the *Local Branching* method, an integer solution has to be found early in the solving process. This can be done by using a primal heuristic (See Section 8.3) but also $\{0, \frac{1}{2}\}$-cuts improve the probability that fractional solutions in the root node are rejected because they violate a $\{0, \frac{1}{2}\}$-cut. This leads to an improvement of the lower bounds obtained in the root node and therefore to a better overall running

time. Note that the separation of $\{0, \frac{1}{2}\}$-cuts works as a black box approach. No knowledge about the problem has to be input into the $\{0, \frac{1}{2}\}$-separation procedure. The only thing to consider is the form of the inequalities that are selected as candidates for the cut separation procedure. As the constraints in our formulations are suitable for the *L-weakening* procedure and normally combinations can be found that produce $\{0, \frac{1}{2}\}$-cuts, this framework provides an easy way to strengthen our formulations.

Chapter 8

Local Branching

Additionally to the method of $\{0, \frac{1}{2}\}$-cuts we implemented another improvement to speed up the process of solving the given integer linear programs. The so called *local branching* procedure we used is an extension to the basic *branch and cut* method which was presented in [16].

The goal of state of the art MIP solvers is to efficiently solve the given problem to provable optimality. Only for the most easy instances of MIPs this can be achieved without the usage of *branch and cut* methods which means that the overall solving process for integer linear programs is NP hard. This makes many medium to large sized instances unsolvable in short periods of time. Nevertheless the goal of the solvers remains unchanged. Still they look for a provably optimal solution that often cannot be found before the given time limit is reached. The basic idea behind the presented method is to change this goal to find a good integer solution quickly in the solving process. Later on the search space that is partitioned and the neighbourhood of the actual incumbent solution is searched first. This is done because there may exist better incumbent solutions that do not differ that much from the actual best integer solution vector. The next section shows more in detail how this process is done. Because the presented method needs a starting solution we give in Section 8.3 a heuristic method that can be used as a primal heuristic and as well as a integer rounding heuristic to obtain integer solutions from fractional solution vectors during the solving process.

8.1 k-neighbourhoods

As stated before, this method applies to the branching process directly after the relaxation of the root node has been solved by the ILP solver. At this point it is of crucial importance that there is already a valid integer solution known to the optimizer. Using the method presented in the next section this is no problem as only in rare cases the given heuristic terminates without

giving a new incumbent solution. What we want to do now is to add new constraints to effectively restrict the search space P_I to the neighbourhood of the given solution vector $\overline{x}$. The first idea would be to fixate some of the given variables $\{x_1, ..., x_n\}$ to the corresponding values of the solution vector $\{\overline{x}_1, ..., \overline{x}_n\}$. This is called a *hard variable fixing* heuristic. The drawback is that it is very hard to select the right subset of variables that should remain unchanged. This fact makes hard variable fixing practically unusable.

There exists however another approach that is called *soft variable fixing* which avoids the problems given above. The idea is to define a maximum distance from the actual incumbent solution $\overline{x}$ where the search space is explored and not the actual subset of fixated variables. In the defined neighbourhood the ILP solver may decide itself which subset of variables should be fixated. This can easily be achieved by adding a inequality of the following type that restricts the distance from the actual incumbent vector $\overline{x}$. If we define $\overline{S} = \{x_i \in B : \overline{x}_i = 1\}$ where B denotes the set of variables, we can define a constraint that gives us a k-neighbourhood as follows.

$$\Delta(x, \overline{x}) = \sum_{j \in \overline{S}} (1 - x_j) + \sum_{j \in B \setminus \overline{S}} x_j \leq k \tag{8.1}$$

This constraint states that the number of variables that flip their value compared to the incumbent vector has to be less than k in any newly found solution. Note that this constraint applies only if the used variables x_i are boolean integer variables, i.e., $x \in \{0, 1\}^n$. For the formulations presented in this thesis all variables are of the type demanded therefore we only give the appropriate constraint.

If we only added constraint (8.1) to the problem, we would cut many valid solutions from the search space P_I. Therefore we have to use a branching process similar to *branch-and-cut* that starts with searching the direct neighbourhood of the solution but nevertheless continues with exploring the remaining space if no new solution could be found. How this is done is explained in the next section.

8.2 The branching process

Normally when the ILP solver cannot find an optimal integer solution for a given LP relaxation it starts a branching procedure that works as follows. It selects one of the variables x_i whose value in the current solution vector $\overline{x}$ is fractional, e.g., $0 < \overline{x}_i < 1$ and splits up the problem P into two subproblems P_0 and P_1. For those subproblems the constraints

$$x_i = 0$$

and

$$x_i = 1$$

are added respectively. Those two problems can then be solved independently by the ILP solver and the best overall solution found denotes the solution for the original problem P. Figure 8.1 shows a simple branch and bound tree where 3 subproblems have to be optimized for solving the original problem to provable optimality.

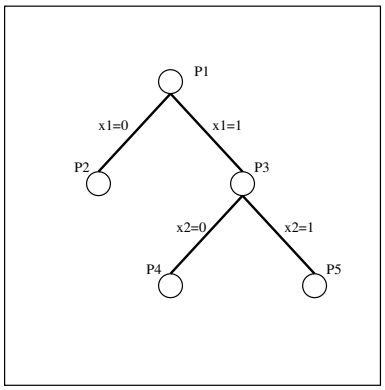


Figure 8.1: A simple branch-and-bound tree

Note that this strategy splits the solution space into two parts of equal size so solving the various subproblems may still be a very hard task to accomplish.

This is different in the method we present here. If we are given an incumbent solution $\overline{x}$ we can generate two subproblems as in the process outlined above. For one subproblem we add the constraint

$$\Delta(x, \overline{x}) \leq k$$

and we force the solver to explore this branch of the tree first. The second subproblem is generated by adding a constraint that defines the remaining search space. The constraint

$$\Delta(x, \overline{x}) \geq k + 1$$

accomplishes this task.

For a appropriate value of k the left branch defines only a small part of the search space and can therefore be explored quickly. Nevertheless the optimization of this subproblem often leads to new best integer solutions. If during the exploration of the left branch a new best integer solution $\overline{x}^2$ is found, the *local branching* procedure can be applied iteratively by searching the k-neighbourhood of $\overline{x}^2$ first. Figure 8.2 which is taken from [16] shows a local branching tree where 2 new best integer solutions could be found while exploring left branches of the tree. The local branching process is iteratively restarted at the nodes 3 and 5 with the discovered solutions. In node 7 the normal branch-and-cut process is continued.

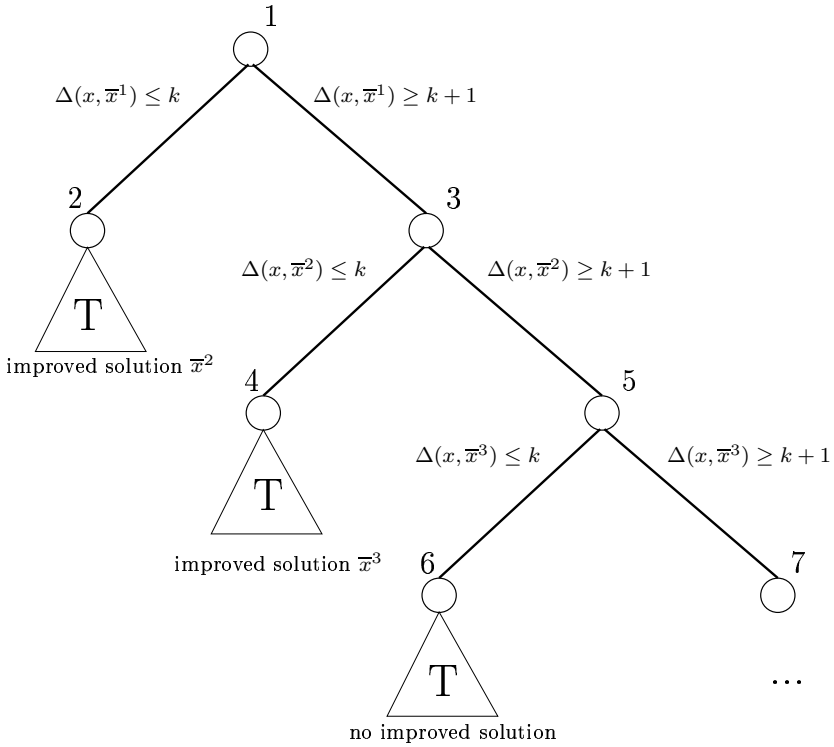


Figure 8.2: A local branching tree

For starting the *local branching* method a valid integer solution is necessary. How such a starting solution can be found is described in the following section.

8.3 Obtaining an incumbent solution

As we mentioned before, it is of crucial importance to obtain feasible integer solutions as early as possible in the solving process for two reasons. The first reason is that if no valid solution is known to the optimizer, the *local branching* process cannot be started. For obtaining good results, this process has to be initiated right after processing the root node of the branch-and-cut tree. Otherwise, if the *local branching* process starts in a node deeper in the tree, the search space cannot be parted correctly which makes the optimizer process it more than once. This clearly leads to an increase of the solving time which makes the availability of an incumbent solution right after the root node a necessity.

The second reason why we should spent some effort on finding incumbent solutions early is that many of the instances we tried to solve are too difficult to find optimal solutions in a reasonable amount of time. Often the imposed

time limit stops the optimizer after processing only a few nodes after the root node. In those cases the aim of the optimizing process is to find good upper bounds on the optimal solution which have to be found within the time limit.

For obtaining integer solutions we implemented a heuristical procedure that can be used as a *Primal Heuristic* to find a start solution before the optimization process starts and as a *MIP Heuristic* that tries to convert a fractional solution of an LP relaxation to a valid integer solution of the original problem during the solving process. The following section will show how this can be done for solving instances of the ST.

8.3.1 The ST Heuristic

For converting a fractional solution for an ST instance into an incumbent solution, we start by obtaining the values $LP(x_i)$ of the variables x_i that correspond to the edges e_i.

Now we construct a graph $G' = (V', E')$ that contains

- all customer nodes from V
- all edges e_i with $LP(x_i) \geq 0.5$
- all noncustomer nodes with at least one incident edge e_i with $LP(x_i) \geq 0.5$

Furthermore we redefine the length of each edge by multiplying the original length $c(e)$ with the corresponding fractional solution value $LP(x_i)$.

Note that in some rare cases, the graph G' might not be connected. In those cases, the LP solver would reject the obtained heuristical solution so we did not check for this property.

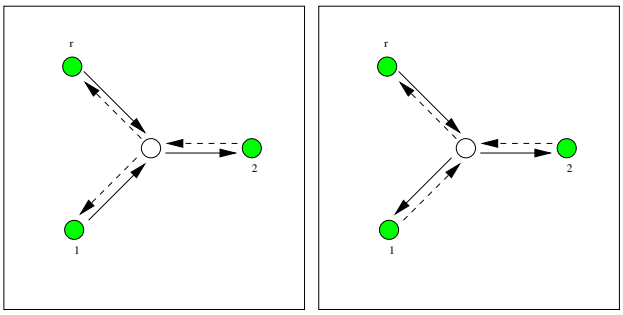
We stated before that we may also use this heuristic before the solving process. In this case we do not have a fractional solution and assume $LP(x_i) = 1$ for all x_i.

On this graph we calculate a distance network. This is done by selecting all the customer nodes of V' and calculate the shortest path $path(u, v)$ for all possible pairs $u, v \in V', u \neq v$. Then we construct a complete graph $G^* = (V^*, E^*)$ with $V^* = R$ and $E^* = \{(u^*, v^*) : u^*, v^* \in V^*\}$ and define a length function $l : E^* \mapsto \mathbb{R}$ on the edges as $l(e) = path(u, v)$ for $e = (u, v)$.

The next step in the heuristic is to calculate a minimum spanning tree on the constructed distance network. As the network is a complete graph this is always possible. Finally the obtained tree is converted back into a tree in G in the following way:

For every edge $e^* = (u^*, v^*)$ of the minimum spanning tree we calculate the shortest path from u to v in G. All edges that are contained in the corresponding shortest path are marked as part of the solution.

Finally we have to note that after the transformation to the *Steiner Arborescence Problem* we have to deal with a directed biconnected graph. Therefore the paths $path(u, v)$ and $path(v, u)$ contain different edges. This does not have an impact on the objective value of the solution but the given constraints demand that every solution is a tree. If we want the optimizer to accept a newly found solution, we have to ensure that it satisfies the given constraints. We implemented a simple depth first search procedure that turns around all edges that point to the root node. This problem is demonstrated in Figure 8.3. If the paths $p(R, 2)$ and $p(1, 2)$ are used to obtain the final solution, the resulting set of edges does not denote a tree (Figure 8.3(a)). After the repair procedure, one edge is turned around, giving a valid solution (Figure 8.3(b)).



(a) A solution obtained by adding the paths $(r, 2)$ and $(1, 2)$

(b) The same solution after the repairing process

Figure 8.3: An example for an invalid resulting solution

The last problem is that under certain circumstances the resulting solution may be invalid. If three or more paths cross each other unfavourably, a circle may emerge as shown in Figure 8.4. This problem arises seldom therefore we chose not to correct it. Instead if such a circle is detected, the heuristic is stopped. The next iteration of the heuristic that is invoked after optimizing the problem a bit further yields valid solutions in most cases so we decided not to address this issue.

An example for the heuristic is shown in Figure 8.5. Figure 8.5(a) shows an easy instance of an ST. For simplicity we only show an undirected graph. We assume that the variables corresponding to the dotted edges have a value in the fractional solution that is below the threshold and are therefore excluded in the first step of the heuristic. Figure 8.5(b) shows the distance network we obtain by calculating the shortest paths between all pairs of

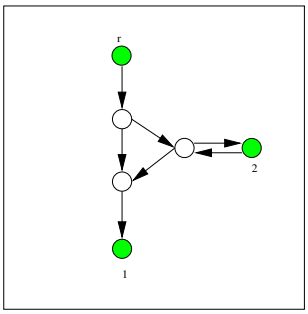


Figure 8.4: An invalid heuristic solution containing a circle

customer nodes and constructing a complete graph as described above. In Figure 8.5(c) a minimum spanning tree of the distance network is given. Figures 8.5(d) to 8.5(f) each show a path that corresponds to one of the edges in Figure 8.5(c). Finally we select all edges that were part of any path given in Figures 8.5(d) to 8.5(f) and obtain the valid incumbent solution in Figure 8.5(g).

Note that if the dotted edges would not have been excluded, the obtained incumbent solution would have been much worse.

8.3.2 The PCST Heuristic

The heuristic used for calculating PCST incumbent solution is very similar to the algorithm used for ST instances. The changes are presented in the following paragraph.

In the first step we have to define the graph G' a bit differently. For the PCST it is not determined that all customer nodes are taken into consideration for the further steps. Therefore we look at the corresponding variables x_{hh} and only take those customer nodes into the graph whose $LP(x_{hh}) < 0.5$. If no fractional solution is given, we simply take all customer nodes as candidates. The artificial root node as well as all its incident edges are excluded from G'.

The graph $G' = (V', E')$ is now constructed by selecting from G

- all customer nodes $v \in V$ with their corresponding $LP(x_{vv}) < 0.5$ and $v \neq r$
- all edges e_i not incident to r with $LP(x_i) > 0.5$
- all noncustomer nodes with at least one incident edge e_i with $LP(x_i) > 0.5$

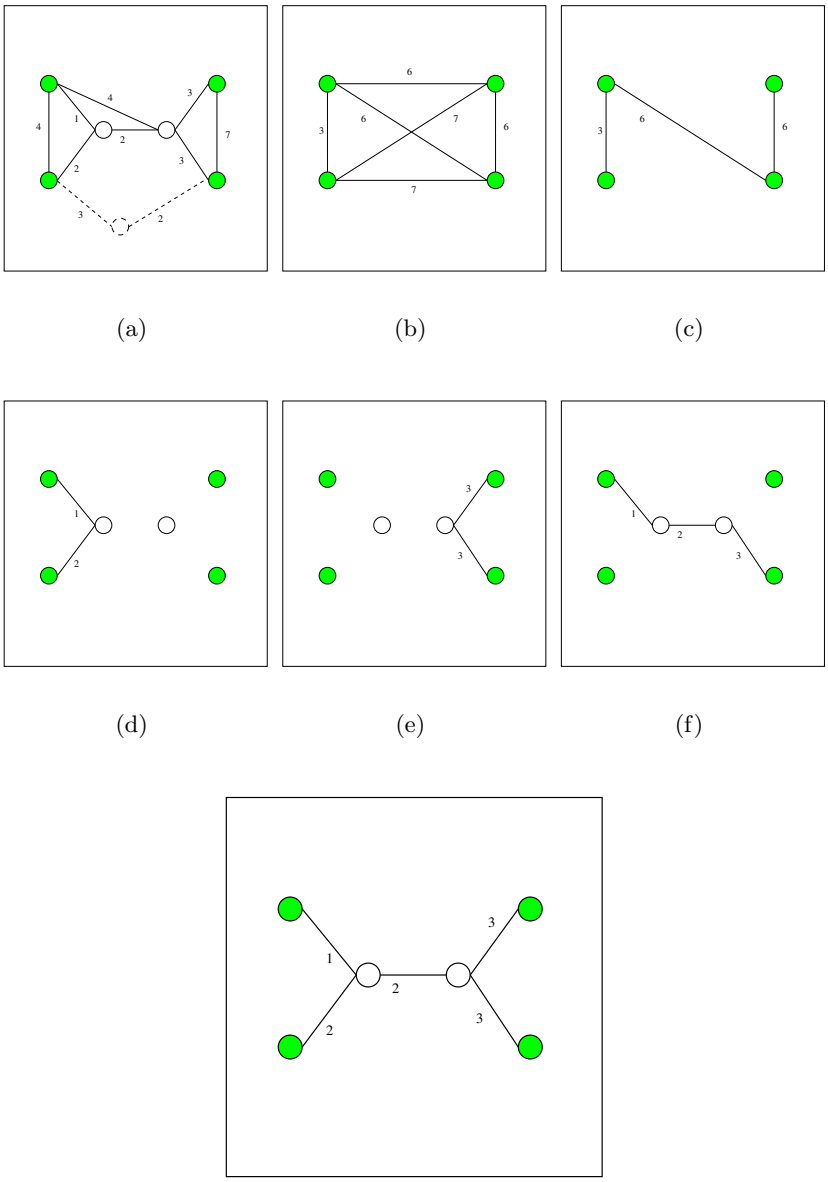


Figure 8.5: An easy example for the ST heuristic

The following steps are applied as in the ST heuristic until the edges have been marked for building the solution. The repair procedure can only be applied if the root node is part of the solution. Therefore we extend the solution by adding the root node r as well as the edge from r to the customer

node with the lowest index that is part of the solution. Note that we have to select the node with the lowest index because otherwise we would violate the asymmetry constraints. The rest of the heuristic for the PCST functions is as the heuristic presented before.

8.3.3 Analysis

For determining the asymptotic running time for the presented heuristics we have to look at the algorithms that are used in the various steps.

- The algorithm for calculating the *all pairs shortest path* matrix runs in $O(nm + n^2 \log n)$.
- The minimum spanning tree can be calculated in $O(m \log n)$.

The distance network contains $|R|$ nodes so the minimum spanning tree contains $|R| - 1$ edges. For every edge, one shortest path has to be traversed for marking the edges which can take a time of $O(m)$ for every single path, giving $|R|O(m)$.

The overall asymptotic running time for the heuristic can be given as

$$O(nm + n^2 \log n) + O(m \log |R|) + |R|O(m) = O(nm + n^2 \log n)$$

Summarized we can say that the running time of the heuristics is quite high compared to the time spent in the LP solving procedure. In Section 9 we will see however, that the big advantages in most cases outweigh the high running time.

Chapter 9

Computational Results

In this section we present results we obtained when we applied our method to various instances for the ST as well as for the PCST. First we will give some results on very hard instances for the ST in Section 9.1, results for the PCST instances are given in Section 9.2.

For solving the integer linear programs presented in this thesis we used the widely used commercial MIP solver ILOG Cplex 8.110. Further information on the solver can be found at [1].

The results given in this section have been calculated using the following machines:

- A Pentium IV with 2.8 GHz and 2 GB of RAM
  On this machine we calculated the results for the I640 test set as well as the results for the hc test set for the PCST.
- An Athlon XP 2100+ with 512 MB
  On this machine we only obtained the results give in table 9.3
- A Dual Pentium IV with 2x2.6 GHz and 1 GB RAM
  Our CPLEX license only allows us to use one CPU so the results should be comparable to using a single CPU Pentium IV, 2.6 GHz.
  On this machine we calculated the remaining results.

General Parameter Settings When we obtained our benchmark results, we applied our approach with the standard parameter settings as it is described in this thesis. We used the Nested Cuts approach and generated more than one cut for all the nodes in the target component (See Section 6.3.2) and we also generated backcuts at the same time (See Section 6.3.3). We included the flow balance constraints and added asymmetry constraints for the PCST instances (See Section 5.4). All the default settings for the generation of $\{0, \frac{1}{2}\}$-Cuts is described in Section 7.5. For our runs we left the default parameters of the CPLEX solver unchanged with one exception: We changed

the variable selection strategy (CPLEX parameter CPX_PARAM_VARSEL) to "branch based on pseudo reduced costs" (4) which gives slightly better results in most cases. For most runs we set a time limit of 5400 seconds except for testing the impact of the ILP heuristic, where we used a time limit of 1 hour.

Randomness As stated in Section 6.3.1 we process the candidate nodes with $x_{vv} < 1$ in a random order when we separate cuts. For our benchmark runs we used a fixed seed so the results are reproduceable. However we want to sketch out, that using a different seed may influence the quality of the obtained solutions heavily. We noticed that it is not uncommon that the running times differ for more than 10% if other seeds are used.

9.1 Results for the ST

For testing our approach on ST instances we have chosen two very hard benchmark sets from the library of Steiner Tree Problem instances available at http://elib.zib.de/steinlib ([24]). Details on the results are given in the next section.

9.1.1 The test sets

PUC The first set of instances we tried to solve is called the PUC test set. It was introduced by Rosetti et al. in 2001 ([30]). This test set contains a series of hypercube graphs as well as bipartite ones and is the hardest test set in the Steiner Tree Library. Some of the instances of this test set were so hard that our approach was not able to process the root node during the 5400 seconds we set as a time limit. Therefore we excluded those instances and present only results for the remaining ones whose overview is given in Table 9.1.

I640 The second set of instances is called the I640 test set. This set was introduced by Duin in 1993 ([10]). These instances are randomly generated sparse graphs whose edge weights are chosen in a way to defy preprocessing. Details on the instances are given in Table 9.2.

Name	|V|	|E|	|R|
bip42p	1200	3982	200
bip42u	1200	3982	200
bip52p	2200	7997	200
bip52u	2200	7997	200
bip62p	1200	10002	200
bip62u	1200	10002	200
bipa2p	3300	18073	300
bipa2u	3300	18073	300
bipe2p	550	5013	50
bipe2u	550	5013	50
cc3-4p	64	288	8
cc3-4u	64	288	8
cc3-5p	125	750	13
cc3-5u	125	750	13
cc5-3p	243	1215	27
cc5-3u	243	1215	27
cc6-2p	64	192	12
cc6-2u	64	192	12
hc6p	64	192	32
hc6u	64	192	32
hc7p	128	448	64
hc7u	128	448	64
hc8p	256	1024	128
hc8u	256	1024	128
hc9p	512	2304	256
hc9u	512	2304	256
hc10p	1024	5120	512
hc10u	1024	5120	512

Table 9.1: The PUC test set

Name	|V|	|E|	|R|	Name	|V|	|E|	|R|	Name	|V|	|E|	|R|	Name	|V|	|E|	|R|
i640-001	640	960	9	i640-101	640	960	25	i640-201	640	960	50	i640-301	640	960	160
i640-002	640	960	9	i640-102	640	960	25	i640-202	640	960	50	i640-302	640	960	160
i640-003	640	960	9	i640-103	640	960	25	i640-203	640	960	50	i640-303	640	960	160
i640-004	640	960	9	i640-104	640	960	25	i640-204	640	960	50	i640-304	640	960	160
i640-005	640	960	9	i640-105	640	960	25	i640-205	640	960	50	i640-305	640	960	160
i640-011	640	4135	9	i640-111	640	4135	25	i640-211	640	4135	50	i640-311	640	4135	160
i640-012	640	4135	9	i640-112	640	4135	25	i640-212	640	4135	50	i640-312	640	4135	160
i640-013	640	4135	9	i640-113	640	4135	25	i640-213	640	4135	50	i640-313	640	4135	160
i640-014	640	4135	9	i640-114	640	4135	25	i640-214	640	4135	50	i640-314	640	4135	160
i640-015	640	4135	9	i640-115	640	4135	25	i640-215	640	4135	50	i640-315	640	4135	160
i640-021	640	204480	9	i640-121	640	204480	25	i640-221	640	204480	50	i640-321	640	204480	160
i640-022	640	204479	9	i640-122	640	204480	25	i640-222	640	204480	50	i640-322	640	204480	160
i640-023	640	204480	9	i640-123	640	204480	25	i640-223	640	204480	50	i640-323	640	204480	160
i640-024	640	204480	9	i640-124	640	204480	25	i640-224	640	204480	50	i640-324	640	204480	160
i640-025	640	204480	9	i640-125	640	204480	25	i640-225	640	204480	50	i640-325	640	204480	160
i640-031	640	1280	9	i640-131	640	1280	25	i640-231	640	1280	50	i640-331	640	1280	160
i640-032	640	1280	9	i640-132	640	1280	25	i640-232	640	1280	50	i640-332	640	1280	160
i640-033	640	1280	9	i640-133	640	1280	25	i640-233	640	1280	50	i640-333	640	1280	160
i640-034	640	1280	9	i640-134	640	1280	25	i640-234	640	1280	50	i640-334	640	1280	160
i640-035	640	1280	9	i640-135	640	1280	25	i640-235	640	1280	50	i640-335	640	1280	160
i640-041	640	40896	9	i640-141	640	40896	25	i640-241	640	40896	50	i640-341	640	40896	160
i640-042	640	40896	9	i640-142	640	40896	25	i640-242	640	40896	50	i640-342	640	40896	160
i640-043	640	40896	9	i640-143	640	40896	25	i640-243	640	40896	50	i640-343	640	40896	160
i640-044	640	40896	9	i640-144	640	40896	25	i640-244	640	40896	50	i640-344	640	40896	160
i640-045	640	40896	9	i640-145	640	40896	25	i640-245	640	40896	50	i640-345	640	40896	160

Table 9.2: The I640 test set

9.1.2 Results for adding the ILP Heuristic

In this section we briefly want to show the impact of using the ILP Heuristic presented on the results (See Section 8.3). Table 9.3 shows the upper bounds we obtained on the PUC test set when we used the heuristic (column Heuristic) compared to the process without using the heuristic (column Basic Optimization). We set a time limit of 3600 seconds for obtaining those results which is very tight for those instances and we used the rather slow Athlon machine that in our experiments performed much worse than the Pentiums. It seems like the CPLEX solver is highly optimized for the Intel architecture so the bounds are not that good. We can see however, that only for one instance, the bound without using the heuristic is better. In most cases, using the heuristic yields better results, so for obtaining the results in the next sections we chose to use the Heuristic for all the runs.

9.1.3 Results for adding $\{0, \frac{1}{2}\}$- Cuts

This section will show how the adding of $\{0, \frac{1}{2}\}$-Cuts can improve the quality of the obtained bounds. In general we can say that the solution quality improves compared to the basic optimization but that the improvement depends heavily on the instance we apply the procedure to.

As we stated before, generating $\{0, \frac{1}{2}\}$-Cuts is a time consuming process. If such cuts can be found, often the quality of the bounds improves drastically so that the overall results are still better than the ones obtained by the basic optimization. The problem that arises is that not for all instances such cuts can be found. There exist some instances that are too easily solvable so that a lot of time is spent looking for good $\{0, \frac{1}{2}\}$-Cuts but no cuts are actually inserted into the LP. Clearly, this leads to greater running times compared to the original process.

Upper Bounds In Figure 9.1 we show how the optimization process works when $\{0, \frac{1}{2}\}$-Cuts are inserted. In those figures we compare the upper bounds of the basic process (See Section 6, labeled C0) to the ones of the same process augmented with the separation of $\{0, \frac{1}{2}\}$-Cuts (See Section 7, labeled C1). As we can see, for all instances the first integer solution is found later in the run with $\{0, \frac{1}{2}\}$-Cuts. This is expected behaviour as some time is needed especially at the root node for generating the cuts. Later on, the quality of the obtained bounds improves quickly for the run including $\{0, \frac{1}{2}\}$-Cuts, giving better results at the end.

Local Branching In Figure 9.2 we give an example of the solving process when Local Branching and $\{0, \frac{1}{2}\}$-Cuts are combined. Results for Local Branching are given in the next section but here we want to show, that adding $\{0, \frac{1}{2}\}$-Cuts can be used effectively to improve the quality of the

Instance	Basic Optimization	Heuristic
bip42p	24881	**24705**
bip42u	**237**	238
bip52p	no	**25182**
bip52u	237	237
bip62p	no	**23020**
bip62u	227	227
bipa2p	no	**36759**
bipa2u	no	**379**
bipe2p	5626	**5620**
bipe2u	54	54
cc3-4p	2338	2338
cc3-4u	23	23
cc3-5p	3699	3699
cc3-5u	36	36
cc6-2p	3271	3271
cc6-2u	32	32
hc6p	4003	4003
hc6u	39	39
hc7p	7925	**7921**
hc7u	77	77
hc8p	15416	**15375**
hc8u	151	**149**
hc9p	no	**30491**
hc9u	298	**297**
hc10p	no	**62445**

Table 9.3: Upper Bounds compared for using the ILP Heuristic on the PUC test set

obtained upper bounds even if Local Branching is used. The graphic shows the upper bounds found for two runs with a Local Branching neighbourhood of 25. In one run (C0L25) we have not applied $\{0, \frac{1}{2}\}$-Cuts, in the other (C1L25) we did. As we can see, the results are quite similar to the runs in Figure 9.1. The basic optimization process yields the first incumbent solution faster, but the quality of the bounds found by the process with $\{0, \frac{1}{2}\}$-Cuts improves quicker giving better overall results.

Lower Bounds Basically $\{0, \frac{1}{2}\}$-Cuts are a method to invalidate fractional solution. So the main goal is to improve the lower bound of a given LP relaxation. In most cases better upper bounds are found too but the main

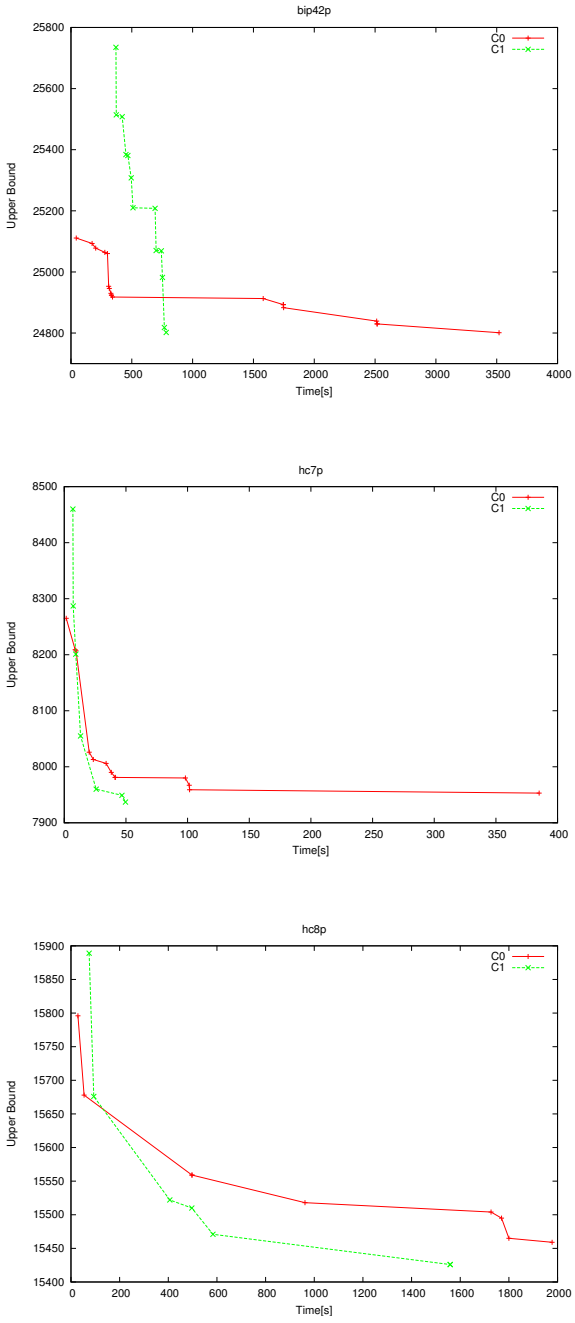


Figure 9.1: Comparing optimization with $\{0, \frac{1}{2}\}$-Cuts (C1) to the basic optimization (C0) for chosen instances

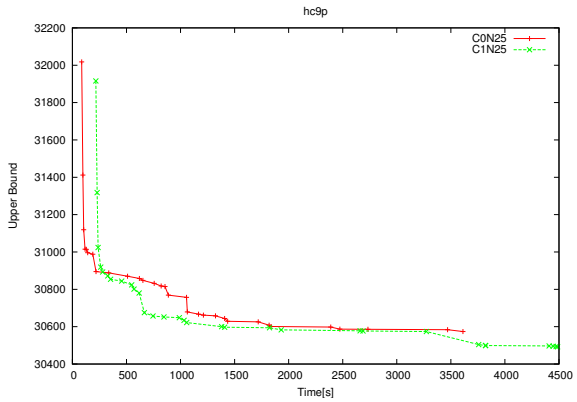


Figure 9.2: Comparing optimization with $\{0, \frac{1}{2}\}$-Cuts (C1) to the basic optimization (C0) with local branching

effect $\{0, \frac{1}{2}\}$-Cuts have on the solving process are better lower bounds. We show an example of this effect in Figure 9.3. The graph shows the lower bounds at the times when new best incumbent solutions are found. As we can see, in the run with $\{0, \frac{1}{2}\}$-Cuts (labeled C1) the first integer solution is found later but the lower bound is already a bit better than in the run without the $\{0, \frac{1}{2}\}$-Cuts (labeled C0). Later in the optimization process the difference between the bounds gets even bigger so the positive effect of the $\{0, \frac{1}{2}\}$-Cuts inserted is noticeable.

Overall Results Table 9.4 gives an overview on the improvement introduced by $\{0, \frac{1}{2}\}$-Cuts. The table compares upper and lower bounds for the basic optimization process (Columns C0) to the augmented one (Columns C1) and marks better values as bold. As we can see, $\{0, \frac{1}{2}\}$-Cuts improve the bounds in many cases but not for every single instance. The rightmost column gives the number of $\{0, \frac{1}{2}\}$-Cuts that could be found. Note that for some of the instances no cuts could be generated. For the especially difficult hypercube instances of this test set (hc*) the most $\{0, \frac{1}{2}\}$-Cuts could be found which is not surprising. As we will see in the next sections, it is harder to apply $\{0, \frac{1}{2}\}$-Cuts if the instances are easily solvable.

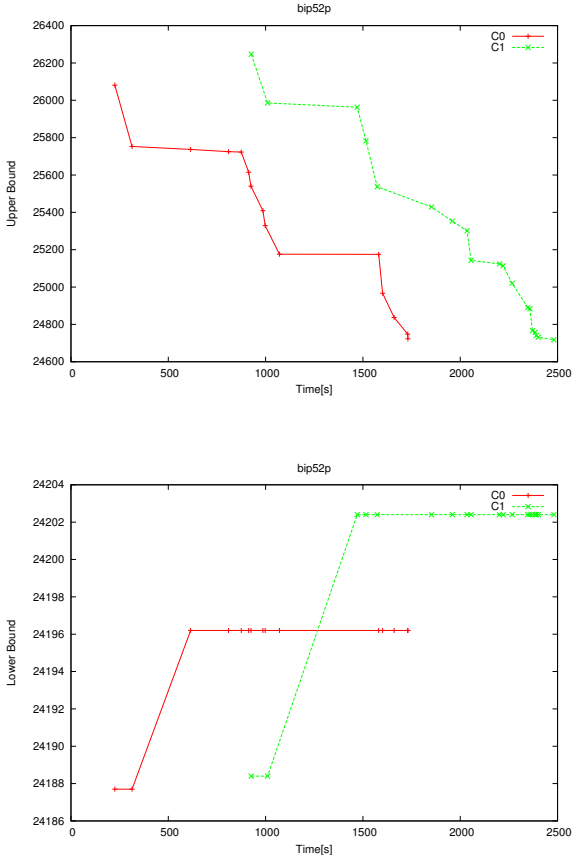


Figure 9.3: Comparing upper and lower bounds of runs with and without using $\{0, \frac{1}{2}\}$-Cuts

	C0		C1		
Instance	UB	LB	UB	LB	# cuts
bip42p	**24801**	24404.7	24802	**24408.6**	55
bip42u	**238**	**232.391**	241	232.197	48
bip52p	24723	24203.4	**24718**	**24207.8**	21
bip52u	236	229.332	236	**229.345**	55
bip62p	**22963**	22467.1	22986	22467.1	9
bip62u	243	213.557	243	213.557	0
bipa2p	37457	34697	**37284**	**34697.1**	9
bipa2u	**351**	329.511	377	329.511	1
bipe2p	5626	5545.21	5626	5545.21	0
bipe2u	54	opt	54	opt	0
cc3-4p	**2338**	**2247.17**	2339	2240.08	11
cc3-4u	23	22	23	22	7
cc3-5p	3779	3384.67	**3691**	**3384.89**	1
cc3-5u	36	**32.869**	36	32.8688	0
cc5-3p	8262	7161.08	8262	**7161.22**	0
cc5-3u	90	69.231	90	**69.2324**	0
hc6p	4003	opt	4003	opt	120
hc6u	39	37.7143	39	**37.8235**	84
hc7p	7953	7715.13	**7937**	**7715.31**	7
hc7u	77	73.5	77	**73.7**	67
hc8p	15459	**15131.7**	**15426**	15129.5	34
hc8u	149	145.143	149	145.143	6
hc9p	**30577**	29884.1	30630	29884.1	3
hc9u	**299**	286.875	345	286.875	9

Table 9.4: Overall results for applying $\{0, \frac{1}{2}\}$-Cuts to the PUC test set

9.1.4 Results when using Local Branching

In this section we want to show the effect of adding the Local Branching method to the optimization process. As we stated before, the idea behind this process is to find integer solutions quickly by dividing the search space and preferably searching the neighbourhood of the actual best incumbent solution. Note that this process aims at finding good solutions to very hard instances. Usually the running time does not improve if the whole search space is processed during the optimization. In most cases, the only improvement arises, if a time limit is set within which the instance cannot be solved to optimality.

Incumbent Solution Quality Figure 9.4 shows how the quality of the incumbent solutions improves when Local Branching is applied. We tested this approach using two parameter settings for the size of the neighbourhood. In the graphic, N10 and N25 denote the parameter settings of N=10 and N=25 for the maximal distance between a newly found solution and the actual best incumbent solution. L0 shows the solutions found by the basic optimization process. As we can see for all selected instances in the figure, after the first integer solution is found, the solution quality quickly improves for the Local Branching variants. For the three instances hc8p, hc10p and bipa2p, Local Branching yields better results than the basic optimization process because the time limit we used (5400 s) is quite tight for those hard instances. In the next paragraph we will show that this does not have to be the case all the time.

Note that in the graphic for the bipa2p instance, the Local Branching with the smaller neighbourhood setting actually gives better results. Although the optimizer with N=25 finds a quite good solution quickly, it cannot improve that solution for quite a long time. Local Branching, like other local search methods, directly depends on the neighbourhood structure of the underlying ST instance. Using Local Branching, the solver can easily get stuck in a local optimum which explains why different settings for the searched neighbourhood size influence the quality of the incumbent solutions heavily. In the given example, the optimizer got stuck in a local optimum. Luckily, another solution could be found within the given neighbourhood so the Local Branching process was able to continue. The time lost gives however a slightly worse overall result than the one from the Local Branching process with N=10.

Drawbacks of Local Branching As we mentioned before, if the optimizer gets enough time to search most of the search space for one given instance, the application of Local Branching yields improvements to a lesser extend. In Figure 9.5 this effect can be seen. At first, the runs with Local Branching enabled find good solutions quickly. After a while however, the

basic optimization process discovers solutions that are significantly better than the ones found from the Local Branching process before. This is not unexpected behaviour. The application of Local Branching does not violate the optimality of the algorithm therefore it would find the optimal solution after some time. In this case however we have the same problem of getting stuck in a local optimum. Especially for the bip62p instance, where many small improvements of the solution take place, we can assume, that the optimizer searches the solution space around a local optimum. If we set the time limit for example to 1000 or 2000 seconds for the instances bip62p and bipa2u respectively, we would get significantly better results for the optimization process with Local Branching enabled. But as the time limit we used is sufficient to find good solutions without the help of Local Branching, the basic optimizer outperforms the Local Branching procedure in those cases.

Results for Local Branching In Table 9.5 we show a summary on the results we obtained by applying Local Branching to the subset of the PUC test suite we used. The columns L0, N10 and N25 represent the three parameter settings (No Local Branching, a neighbourhood of 10, a neighbourhood of 25) respectively, bold values denote the best upper bound for a instance. Note that for the easier instances (bip*) often the basic optimizer yields the best bounds. For the hard ones however (hc8-11) the best bound is obtained using Local Branching with N=25 for 7 out of 8 instances.

9.1.5 Overall Performance

In this section we show the overall results we obtained using our methods to solve the PUC and the I640 test set. The following tables give the obtained bounds, the running time and the best known bounds from the Steinlib ([24]).

In Table 9.6, known optima are denoted bold in the column Steinlib. For three instances of the PUC test set (bip42p, bip52p and bipe2p) we found better upper bounds using a time limit of only 1.5 hours. Those values are denoted italic. For one instance (bipe2u) we were able to find the formerly unknown optimum solution.

For many of the instance in the I640 test set, the optima are known. In Tables 9.7 and 9.8, for instances marked with a $*$ no optimum is known and the column Steinlib gives the best known upper bound. For all other instances, the values given in the column Steinlib denote the optimum solution.

The values we give for our approach were achieved using neither Local Branching nor $\{0, \frac{1}{2}\}$-Cuts. We chose those settings because although the instances from the I640 test set are quite hard we expected to solve many of them to optimality. The fact that we were not interested in finding good

	no LB			N=10			N=25		
Instance	UB	LB	Time [s]	UB	LB	Time [s]	UB	LB	Time [s]
bip42p	24801	24404.7	5576.5	24882	24395.9	5582.5	**24781**	24376.4	5545.3
bip42u	**238**	232.391	5517.9	239	232.262	5482.3	240	232.221	5493.3
bip52p	**24723**	24203.4	5522.2	24917	24187.7	5466.9	24954	24187.7	5479.9
bip52u	**236**	229.332	5470.8	237	229.401	5544.4	237	229.19	5520.0
bip62p	**22963**	22467.1	5535.8	23084	22445.2	5464.9	23174	22445.2	5442.3
bip62u	243	213.557	5434.6	235	213.557	5429.6	**230**	213.39	5446.5
bipa2p	37457	34697	5483.4	**36140**	34687.6	5455.8	36199	34687.6	5452.3
bipa2u	**351**	329.511	5437.3	365	329.325	5453.4	355	329.325	5431.6
bipe2p	5626	5545.21	5623.0	5677	5534.5	5626.0	**5620**	5535.94	5578.3
bipe2u	54	opt	543.4	54	opt	179.4	54	opt	344.3
cc3-4p	**2338**	2247.17	5435.8	2339	2209	5439.9	2343	2149	5427.5
cc3-4u	23	22	5440.6	23	22	5436.9	23	21	5436.5
cc3-5p	3779	3384.67	5468.6	**3688**	3384.67	5442.9	3690	3384.67	5432.6
cc3-5u	36	32.869	5414.9	36	32.869	5414.3	36	32.8688	5414.2
cc5-3p	8262	7161.08	5414.9	8262	7161.08	5414.8	8262	7160.98	5415.6
cc5-3u	90	69.231	5412.2	90	69.2324	5412.0	90	69.2324	5412.9
cc6-2p	3271	opt	1645.0	3271	opt	2805.7	3271	3184.34	5508.4
cc6-2u	32	opt	199.3	32	opt	214.2	32	opt	389.5
hc6p	4003	opt	681.1	4003	opt	691.6	4003	opt	659.5
hc6u	39	37.7143	5451.1	39	37.75	5438.6	39	37.5131	5445.0
hc7p	7953	7715.13	5544.4	7909	7717.48	5538.1	7909	7714.17	5544.5
hc7u	77	73.5	5494.1	77	73.5	5470.6	77	73.5	5493.8
hc8p	15459	15131.7	5506.8	15412	15115.8	5479.2	**15336**	15115.8	5454.3
hc8u	149	145.143	5467.1	149	145.143	5448.5	149	145.143	5490.0
hc9p	30577	29884.1	5435.6	30775	29878.1	5483.8	**30574**	29878.1	5470.6
hc9u	299	286.875	5417.6	299	286.875	5436.8	**297**	286.875	5426.9
hc10p	62015	59219.3	5418.4	61436	59214.3	5421.6	**60963**	59214.3	5425.7
hc10u	730	567.778	5408.3	649	567.778	5408.0	**601**	567.778	5408.4
hc11p	126418	117395	5417.8	126446	117395	5417.6	**125748**	117395	5440.1
hc11u	1499	1125.3	5414.9	1492	1125.3	5416.1	**1477**	1125.3	5416.1

Table 9.5: Upper Bounds for L0, N=10, N=25

upper bounds made the disabling of the improvements a good choice as we know that the results only get significantly better, if the instances are very hard.

Name	Steinlib	UB	LB	Gap	Time [s]
bip42p	24818	*24705*	24383.7	1.31 %	3653.4
bip42u	237	238	232.391	2.59 %	5517.9
bip52p	24936	*24718*	24207.8	2.11 %	5767.2
bip52u	235	236	229.345	3.06 %	5650.9
bip62p	22944	22963	22467.1	2.21 %	5535.8
bip62u	221	227	213.388	6.37 %	3614.2
bipa2p	35774	36140	34687.6	4.19 %	5455.8
bipa2u	342	351	329.511	6.69 %	5437.3
bipe2p	5660	*5620*	5534.5	1.55 %	5629.3
bipe2u	54	**54**	**opt**	-	730.6
cc3-4p	**2338**	2338	2189.15	6.81 %	5433.8
cc3-4u	**23**	23	21	9.52 %	5437.2
cc3-5p	**3661**	3688	3384.67	8.98 %	5442.9
cc3-5u	**36**	36	32.8695	12.50 %	5414.9
cc5-3p	7299	8262	7161.11	15.37 %	5415.1
cc5-3u	71	90	69.231	30.43 %	5411.9
cc6-2p	**3271**	3271	opt	-	1645.0
cc6-2u	**32**	32	opt	-	386.1
hc6p	**4003**	4003	opt	-	713.8
hc6u	**39**	39	37.7143	5.41 %	5438.4
hc7p	**7905**	7906	7715.13	2.48 %	5552.1
hc7u	**77**	77	73.5	5.48 %	5514.0
hc8p	15322	15336	15115.8	1.46 %	5454.3
hc8u	148	148	145.143	2.07 %	5423.0
hc9p	30258	30492	29878.1	2.06 %	5469.8
hc9u	292	297	286.875	3.85 %	5426.9
hc10p	60494	60963	59214.3	2.95 %	5425.7
hc10u	582	601	567.778	6.00 %	5408.4
hc11p	120096	125748	117395	7.12 %	5440.1
hc11u	1162	1477	1125.3	31.29 %	5416.1

Table 9.6: Results for the PUC test set

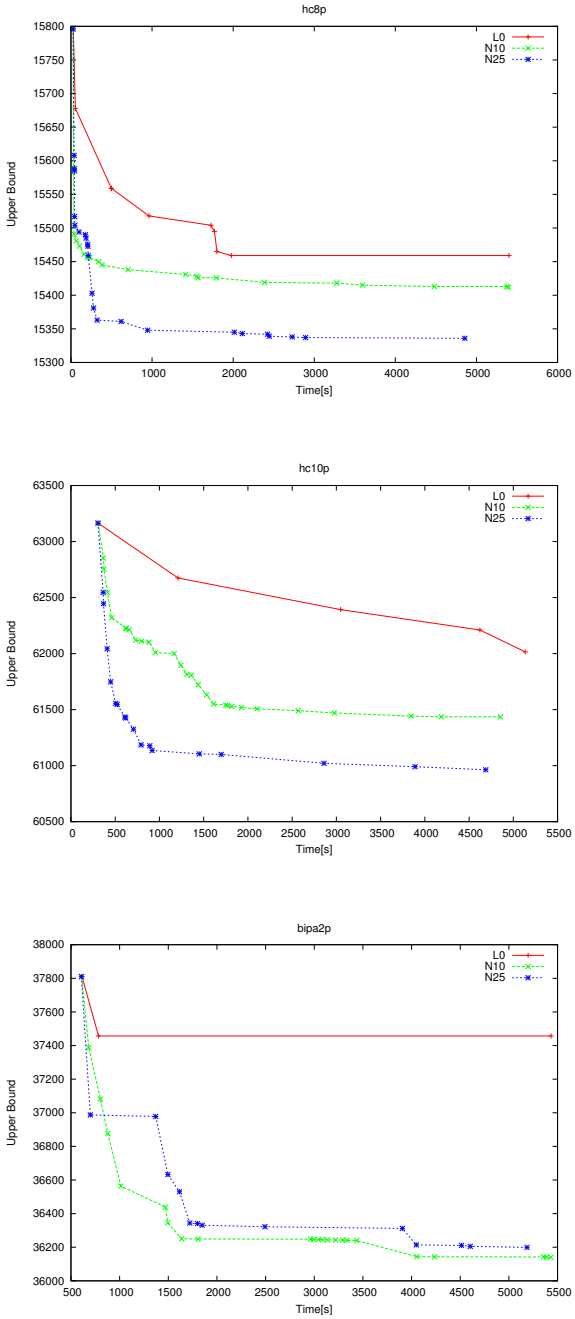


Figure 9.4: Incumbent solutions compared for Local Branching with parameters N=10 (N10), N=25 (N25) and the standard Branch and Cut algorithm (L0)

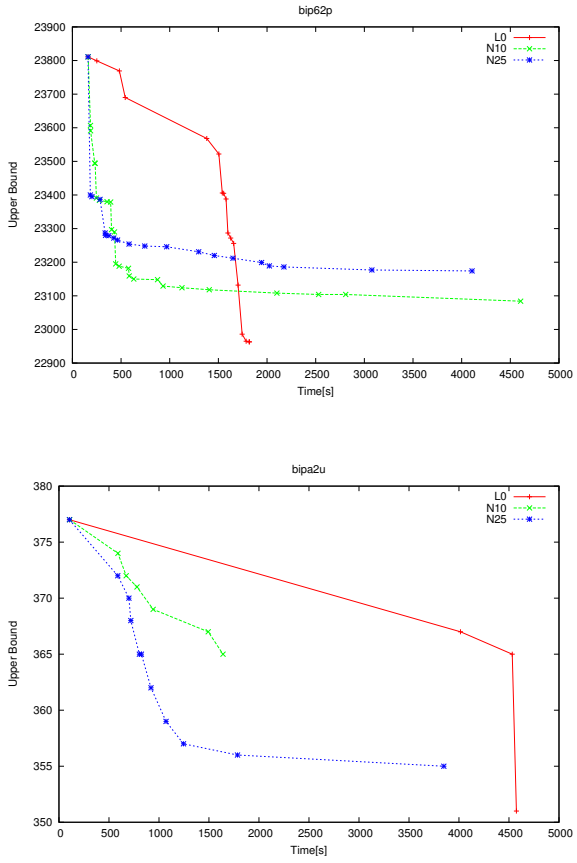


Figure 9.5: Incumbent solutions compared for Local Branching with parameters N=10 (N10), N=25 (N25) and the standard Branch and Cut algorithm (L0)

Name	Steinlib	UB	LB	Gap	Time [s]
i640-001	4033	4033	opt	-	0.3
i640-002	3588	3588	opt	-	0.2
i640-003	3438	3438	opt	-	0.2
i640-004	4000	4000	opt	-	0.4
i640-005	4006	4006	opt	-	0.2
i640-011	2392	2392	opt	-	11.6
i640-012	2465	2465	opt	-	203.4
i640-013	2399	2399	opt	-	54.3
i640-014	2171	2171	opt	-	3.9
i640-015	2347	2347	opt	-	29.5
i640-021	1749	1749	opt	-	918.4
i640-022	1756	1756	opt	-	607.6
i640-023	1754	1754	opt	-	554.0
i640-024	1751	1751	opt	-	596.5
i640-025	1745	1745	opt	-	428.5
i640-031	3278	3278	opt	-	1.1
i640-032	3187	3187	opt	-	0.6
i640-033	3260	3260	opt	-	1.7
i640-034	2953	2953	opt	-	0.6
i640-035	3292	3292	opt	-	2.1
i640-041	1897	1897	opt	-	239.8
i640-042	1934	1951	1924	1.40 %	5457.3
i640-043	1931	1936	1906.5	1.57 %	5462.6
i640-044	1938	1947	1916.5	1.62 %	5478.2
i640-045	1866	1866	opt	-	111.7
i640-101	8764	8764	opt	-	1.2
i640-102	9109	9109	opt	-	0.3
i640-103	8819	8819	opt	-	0.3
i640-104	9040	9040	opt	-	0.5
i640-105	9623	9623	opt	-	14.2
i640-111	6167	6191	6090.1	1.66 %	5418.5
i640-112	6304	6304	6254.43	0.80 %	5420.3
i640-113	6249	7630	6170.49	23.66 %	5424.3
i640-114	6308	6308	6248.92	0.96 %	5425.1
i640-121	4906	4906	opt	-	4426.8
i640-122	4911	4911	opt	-	3996.7
i640-123	4913	5326	4910.31	8.47 %	5507.0
i640-124	4906	4906	opt	-	4978.4

Table 9.7: Results for the I640 test set, part 1

Name	Steinlib	UB	LB	Gap	Time [s]
i640-131	8097	8097	opt	-	5.5
i640-132	8154	8154	opt	-	32.6
i640-133	8021	8021	opt	-	5.2
i640-134	7754	7754	opt	-	15.1
i640-135	7696	7696	opt	-	6.4
i640-141	5199	5550	5152.28	7.73 %	5460.3
i640-142	5193	5572	5157.42	8.05 %	5445.7
i640-143	5194	5357	5162.28	3.78 %	5449.9
i640-144	5205	5416	5167.99	4.82 %	5447.9
i640-145	5218	5978	5174.6	15.54 %	5453.6
i640-201	16079	16079	opt	-	0.3
i640-202	16324	16324	opt	-	0.2
i640-203	16124	16124	opt	-	0.5
i640-204	16239	16239	opt	-	0.6
i640-205	16616	16616	opt	-	1.3
i640-212*	11795	14763	11673.6	26.47 %	5419.2
i640-213*	11879	13743	11740.6	17.06 %	5418.2
i640-214*	11898	13895	11730.7	18.46 %	5430.4
i640-222	9798	10205	9772.63	4.43 %	5505.9
i640-225	9807	10660	9776.88	9.04 %	5504.3
i640-231	15014	15014	opt	-	96.9
i640-232	14630	14630	opt	-	20.7
i640-233	14797	14797	opt	-	124.6
i640-234	15203	15203	opt	-	1.6
i640-235	14803	14803	opt	-	929.0
i640-242	10195	10916	10121.7	7.85 %	5441.6
i640-243	10215	10971	10150.2	8.09 %	5447.0
i640-244	10246	10827	10150.7	6.67 %	5451.8
i640-301	45005	45005	opt	-	0.7
i640-302	45736	45736	opt	-	2.1
i640-303	44922	44922	opt	-	0.5
i640-304	46233	46233	opt	-	0.9
i640-305	45902	45902	opt	-	2.0
i640-311*	35766	36285	35258	2.91 %	5416.6
i640-312*	35829	40508	35245.9	14.93 %	5419.0
i640-313*	35535	40186	35143	14.35 %	5435.2
i640-314*	35538	39433	35051.6	12.50 %	5436.5
i640-315*	35741	39197	35268.1	11.14 %	5476.5
i640-331	42796	42796	opt	-	117.8
i640-332	42548	42548	opt	-	152.9
i640-333	42345	42345	opt	-	658.9
i640-334	42768	42778	42750.5	0.07 %	5432.6
i640-335	43035	43035	opt	-	2252.6

Table 9.8: Results for the I640 test set, part 2

9.2 Results for the PCST

In this section we will give some computational results we got by applying our approach to PCST instances. The test sets we used are a lot easier solvable than the ones available for the ST except for some instances we derived directly from hard ST instances. Additionally we applied our methods to instance files that have been preprocessed using methods similar to the ones presented in [12].

9.2.1 The test sets

The K and P test sets The first test set of quite small instances was introduced by Johnson, Minkoff and Phillips ([21]). This set consists of generated instances ranging from 100 vertices and 284 edges to 400 vertices and 1507 edges.

The Stein{c,d,e} test sets The second test suite we used are instances that are derived from ST instances from the Steinlib. Without preprocessing, the c series instance have 500 nodes and 625 to 12500 edges. The slightly bigger d series instances have 1000 nodes and 1250 to 25000 edges. The biggest set of instances, the e series, consist of 2500 nodes and 3125 to 62500 edges. The size of the preprocessed instances we actually solved is given in Section 9.2.4.

The hc test set Additionally we applied our approach to 8 instances, which we derived from the instances hc6-hc9 from the PUC test set. Those instances are significantly harder to solve than the ones mentioned before. Preprocessing could not be applied to those instances because they were designed to defy preprocessing. For the size of the instances see Table 9.1.

9.2.2 Results for adding $\{0, \frac{1}{2}\}$- Cuts

As we mentioned before, most of the instances we solved for the PCST are quite easy. Some of them are big, so we need some time for solving them to optimality but for example all but 2 instances may be solved to optimality without the use of branching and even without the use of $\{0, \frac{1}{2}\}$-Cuts. As we have seen in the section before, the impact of $\{0, \frac{1}{2}\}$-Cuts gets bigger, as the instances get harder. This explains why for the quite easy test sets we tried to solve, the addition of $\{0, \frac{1}{2}\}$-Cuts does not reduce the computing time. The times for solving the e series instances, which turned out to be the hardest of our test set, are shown in Figure 9.6. As we can see, the running time for C1, which denotes the process with using $\{0, \frac{1}{2}\}$-Cuts, is always higher.

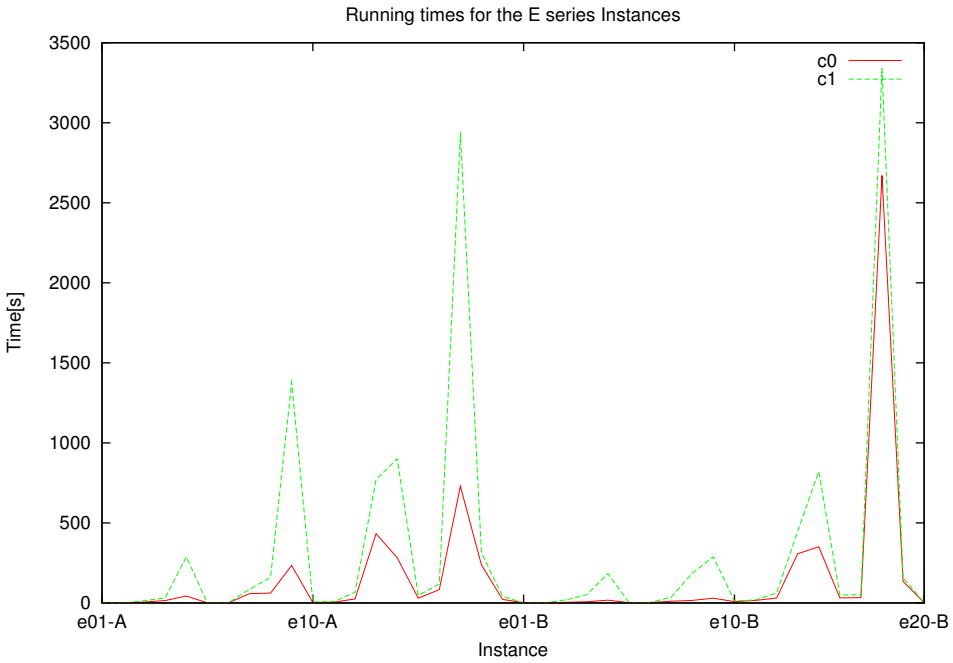


Figure 9.6: Running times compared for C0 and C1 for the E series instances

It turned out, that only for 4 instances of the easier test sets, a small amount of $\{0, \frac{1}{2}\}$-Cuts could be found. However, not even those instances benefit from the additional cuts because they are too easily solved. Results for those instances are given in Table 9.9. Columns C0 and C1 give the running time in seconds for the different parameter settings. As we can see, only a very small amount of cuts was separated and therefore the running time is much worse than without using $\{0, \frac{1}{2}\}$-Cuts.

Name	OPT	Time for C0	Time for C1	# cuts
d13-A	413	7.2	13.0	1
c18-B	77	3.4	11.7	8
e18-A	431	729.4	2938.7	4
e18-B	409	2670.8	3340.8	20

Table 9.9: Running times for the instances where $\{0, \frac{1}{2}\}$-Cuts could be found

The hc test set For the small set of harder solvable hc instances, the addition of $\{0, \frac{1}{2}\}$-Cuts brings an impact on the solution quality. As for the ST problem not all instances get the same benefit from adding $\{0, \frac{1}{2}\}$-Cuts. For some instances the results are significantly better which can be seen in Table 9.10.

	C0			C1		
Instance	UB	LB	Time[s]	UB	LB	Time[s]
hc6p	3908	opt	42.1	3908	opt	48.0
hc6u	36	opt	0.9	36	opt	2.2
hc7p	7734	7649.02	5541.2	**7726**	**7653.77**	5541.4
hc7u	**72**	**opt**	4914.3	73	71	5506.4
hc8p	**15239**	**15017.9**	5516.7	15294	15012.5	5516.4
hc8u	145	141	5409.2	no	141	5409.6
hc9p	30774	29661.3	5469.0	**30133**	**29661.5**	5449.7
hc9u	no	279.358	5411.0	no	279.358	5410.4

Table 9.10: Results for adding $\{0, \frac{1}{2}\}$-Cuts to the converted PUC instances

9.2.3 Local Branching

For the Local Branching process applies the same problem we mentioned before. As the Local Branching process starts after processing the root node and all but 2 instances of the easier test sets can be solved to optimality

at the root node, we can say that those instances are too easy to apply Local Branching. The results for the two instances, where Local Branching can be applied is given in table 9.11. As those instances can be solved to optimality too, as expected, the running times are higher, when Local Branching is applied.

Instance	OPT	no LB	N=10	N=25
e18-A	431	729.4	845.6	855.6
e18-B	409	2670.8	2976.9	2960.2

Table 9.11: Comparing Local Branching parameters for the e18 instances

The hc test set From the hc test set, no instance could be solved to optimality at the root node using our method. Therefore we applied the Local Branching approach to those instances. We chose the neighbourhood settings of N=10 and N=25 because they gave good results on the ST instances we derived the test set from. As we can see in Table 9.12, the upper bounds of 3 of the instances have improved and for one instance (hc7u) the running time to solve the instance to optimality has been reduced using the setting N=10.

	no Local Branching			Local Branching, N=10			Local Branching, N=25		
Instance	UB	LB	Time[s]	UB	LB	Time[s]	UB	LB	Time[s]
hc6p	3908	opt	42.1	3908	opt	57.6	3908	opt	98.8
hc6u	36	opt	0.9	36	opt	0.9	36	opt	0.9
hc7p	7734	7649.02	5541.2	**7721**	7654.19	5558.9	7729	7647.56	5548.3
hc7u	72	opt	4914.3	72	opt	3568.5	72	71	5454.6
hc8p	15239	15017.9	5516.7	15368	14985.4	5470.9	**15232**	14985.4	5498.9
hc8u	145	141	5409.2	145	141	5410.6	145	141	5416.9
hc9p	30774	29661.3	5469.0	31170	29643.9	5424.0	**30329**	29643.9	5429.0
hc9u	no	279.358	5411.0	no	279.358	5417.2	no	279.358	5410.2

Table 9.12: Results for adding Local Branching to the converted PUC instances

9.2.4 Overall Performance

In this section we give the overall results for all the easier test sets. As before, the tables give the size of the preprocessed instances, the optima we found, and the running time we used including the time spent on preprocessing the instances.

Instance	|V|	|E|	OPT	T_{prep}	T_{solve}	T
K100	45	191	113132	0.08	0.1	0.18
K100.1	42	185	113744	0.12	0.1	0.22
K100.2	24	83	138871	0.14	0.1	0.24
K100.3	26	123	95138	0.10	0.1	0.2
K100.4	29	113	69653	0.11	0.0	0.11
K100.5	31	120	106799	0.10	0.1	0.2
K100.6	22	64	112140	0.13	0.0	0.13
K100.7	25	93	137713	0.13	0.1	0.23
K100.8	43	144	173394	0.12	0.1	0.22
K100.9	22	70	108306	0.11	0.1	0.21
K100.10	27	78	113771	0.13	0.0	0.13
K200	81	271	296935	0.51	1.3	1.81
K400	231	914	322470	2.98	64.3	67.28
K400.1	217	854	435701	2.72	150.7	153.42
K400.2	228	948	452422	3.00	169.0	172
K400.3	210	806	385629	3.79	35.0	38.79
K400.4	197	784	376275	2.93	45.7	48.63
K400.5	220	799	472936	2.84	156.9	159.74
K400.6	241	1035	351842	3.97	60.3	64.27
K400.7	225	867	452176	2.24	160.0	162.24
K400.8	235	987	359045	3.09	57.5	60.59
K400.9	211	862	343984	2.99	78.7	81.69
K400.10	221	923	350987	3.69	150.7	154.39
P100	66	163	564753	0.09	0.0	0.09
P100.1	84	196	833544	0.06	0.1	0.16
P100.2	75	187	346602	0.06	0.0	0.06
P100.3	91	237	575606	0.04	0.0	0.04
P100.4	69	186	705356	0.06	0.0	0.06
P200	166	438	1257107	0.24	0.1	0.34
P400	345	1002	2255191	1.96	0.4	2.36
P400.1	323	983	2504926	2.52	0.5	3.02
P400.2	341	997	2240337	2.14	0.3	2.44
P400.3	334	969	2611102	2.31	0.5	2.81
P400.4	344	949	2668758	1.71	0.5	2.21

Table 9.13: Results for the K and P test sets

Instance	|V|	|E|	OPT	T_{prep}	T_{solve}	T
c1-B	125	226	85	1.19	0.1	1.29
c1-A	116	214	10	1.24	0.1	1.34
c2-B	111	209	133	1.10	0.1	1.2
c2-A	109	207	42	1.08	0.0	1.08
c3-B	185	304	628	1.26	0.2	1.46
c3-A	160	277	276	1.10	0.1	1.2
c4-B	218	341	916	1.34	0.2	1.54
c4-A	178	300	373	1.17	0.2	1.37
c5-B	199	314	859	1.74	0.3	2.04
c5-A	163	274	553	1.17	0.2	1.37
c6-B	356	823	55	2.15	0.3	2.45
c6-A	355	822	13	2.11	0.1	2.21
c7-B	365	842	94	2.48	0.2	2.68
c7-A	365	842	47	2.56	0.1	2.66
c8-B	369	850	441	3.03	0.3	3.33
c8-A	367	849	306	2.74	0.2	2.94
c9-B	389	879	607	2.84	1.0	3.84
c9-A	387	877	462	2.38	1.3	3.68
c10-B	323	798	675	3.42	0.7	4.12
c10-A	359	841	647	3.26	0.9	4.16
c11-B	489	2143	32	9.46	4.4	13.86
c11-A	489	2143	18	9.45	0.3	9.75
c12-B	484	2186	45	6.85	0.7	7.55
c12-A	484	2186	37	6.78	0.4	7.18
c13-B	471	2112	235	9.78	3.1	12.88
c13-A	472	2113	215	9.85	0.7	10.55
c14-B	459	2048	259	7.54	0.6	8.14
c14-A	466	2081	250	7.46	0.6	8.06
c15-B	370	1753	337	6.00	0.7	6.7
c15-A	406	1871	366	6.52	1.2	7.72
c16-B	500	4740	11	2.35	1.6	3.95
c16-A	500	4740	11	2.39	1.6	3.99
c17-B	498	4694	16	2.31	1.9	4.21
c17-A	498	4694	16	2.37	2.6	4.97
c18-B	465	4538	77	2.94	3.4	6.34
c18-A	469	4569	79	2.59	1.7	4.29
c19-B	416	3867	61	2.84	0.7	3.54
c19-A	430	3982	75	2.85	0.8	3.65
c20-B	133	563	31	4.99	0.1	5.09
c20-A	241	1222	69	6.09	0.2	6.29

Table 9.14: Results for the C test set

Instance	|V|	|E|	OPT	T_{prep}	T_{solve}	T
d1-B	233	443	106	4.88	0.2	5.08
d1-A	231	440	13	4.87	0.1	4.97
d2-B	264	488	218	4.87	0.1	4.97
d2-A	257	481	30	4.86	0.1	4.96
d3-B	372	606	1330	6.34	0.5	6.84
d3-A	301	529	480	5.49	0.2	5.69
d4-B	387	621	1483	7.22	0.8	8.02
d4-A	311	541	725	5.63	0.5	6.13
d5-B	411	649	1778	11.48	1.6	13.08
d5-A	348	588	1025	7.56	1.2	8.76
d6-B	741	1708	67	14.73	1.3	16.03
d6-A	740	1707	15	14.37	0.2	14.57
d7-B	736	1707	103	11.39	0.3	11.69
d7-A	734	1705	42	11.28	0.2	11.48
d8-B	778	1757	954	12.34	1.0	13.34
d8-A	764	1738	649	11.74	3.7	15.44
d9-B	761	1724	1293	20.92	4.5	25.42
d9-A	752	1716	908	17.95	14.8	32.75
d10-B	629	1586	1346	18.52	2.6	21.12
d10-A	694	1661	1251	14.58	4.4	18.98
d11-B	986	4658	29	23.56	4.8	28.36
d11-A	986	4658	18	27.73	1.5	29.23
d12-B	991	4639	40	22.29	1.1	23.39
d12-A	991	4639	40	23.15	1.6	24.75
d13-B	961	4566	438	27.96	3.5	31.46
d13-A	966	4572	413	27.69	7.2	34.89
d14-B	931	4469	570	37.20	5.0	42.2
d14-A	946	4500	542	35.45	4.0	39.45
d15-B	747	3896	678	49.19	5.2	54.39
d15-A	832	4175	776	47.14	17.7	64.84
d16-B	1000	10595	13	10.83	4.8	15.63
d16-A	1000	10595	13	10.82	4.0	14.82
d17-B	999	10534	22	10.72	5.0	15.72
d17-A	999	10534	22	10.84	4.9	15.74
d18-B	929	9816	151	12.01	6.1	18.11
d18-A	944	9949	161	11.72	50.1	61.82
d19-B	862	9131	170	13.08	11.9	24.98
d19-A	897	9532	202	12.36	12.0	24.36
d20-B	307	1383	87	32.92	0.4	33.32
d20-A	488	2511	155	37.27	0.7	37.97

Table 9.15: Results for the D test set

Instance	|V|	|E|	OPT	T_{prep}	T_{solve}	T
e01-A	651	1246	3	21.51	0.1	21.61
e01-B	655	1250	109	21.84	1.0	22.84
e02-A	694	1304	19	20.70	0.2	20.9
e02-B	697	1307	156	20.67	0.4	21.07
e03-A	813	1414	1422	29.42	5.6	35.02
e03-B	962	1572	3340	30.88	4.4	35.28
e04-A	829	1425	1938	24.91	14.9	39.81
e04-B	980	1588	3846	26.18	7.7	33.88
e05-A	893	1502	2930	36.93	43.4	80.33
e05-B	1029	1644	4816	44.96	17.5	62.46
e06-A	1821	4283	19	37.87	0.5	38.37
e06-B	1821	4283	70	37.60	1.6	39.2
e07-A	1863	4339	35	39.30	0.6	39.9
e07-B	1865	4341	136	39.39	2.7	42.09
e08-A	1902	4379	1656	40.06	58.2	98.26
e08-B	1911	4387	2314	50.38	11.0	61.38
e09-A	1909	4388	2385	50.50	61.8	112.3
e09-B	1918	4397	3046	54.74	15.6	70.34
e10-A	1716	4181	3272	60.62	233.8	294.42
e10-B	1594	4045	3533	84.56	29.7	114.26
e11-A	2491	12063	21	145.10	2.7	147.8
e11-B	2491	12063	34	146.18	8.4	154.58
e12-A	2490	12090	49	82.49	4.6	87.09
e12-B	2490	12090	67	85.53	15.6	101.13
e13-A	2430	11949	1073	148.17	25.4	173.57
e13-B	2407	11915	1120	146.73	30.5	177.23
e14-A	2366	11872	1362	144.21	432.1	576.31
e14-B	2311	11737	1373	145.68	307.7	453.38
e15-A	2044	10845	1906	207.85	283.2	491.05
e15-B	1864	10264	1691	234.27	350.5	584.77
e16-A	2500	29332	15	82.21	29.7	111.91
e16-B	2500	29332	15	81.91	31.9	113.81
e17-A	2500	29090	25	81.61	83.1	164.71
e17-B	2500	29090	25	81.78	33.0	114.78
e18-A	2378	28454	431	85.66	729.4	815.06
e18-B	2347	28269	409	86.86	2670.8	2757.66
e19-A	2156	25011	401	92.15	237.5	329.65
e19-B	2085	23641	340	94.02	134.7	228.72
e20-A	1525	12770	352	107.28	22.1	129.38
e20-B	861	3881	202	231.50	1.3	232.8

Table 9.16: Results for the E test set

Chapter 10

Conclusion and Future Work

We propose the application of $\{0, \frac{1}{2}\}$-Cuts and Local Branching to the ST and PCST problems. We used the new approaches on the set of most difficult ST instances and on a set of easily solvable instances for the PCST available from literature. Our computational results show:

- For several of the harder instances $\{0, \frac{1}{2}\}$-Cuts improve the performance significantly and affect the quality of both the upper and the lower bounds. For some instances we can see however, that the expense of separating $\{0, \frac{1}{2}\}$-Cuts using our algorithm outweighs the advantages.
- Local Branching improves the quality of the upper bounds drastically compared to the basic branch and cut algorithm in limited time. If the available time is sufficient for solving the instance to optimality, the optimizer often gets stuck in local optima and is outperformed slightly by the basic optimization process.
- For three instances of the most difficult test set of ST instances we could improve the best known upper bounds using our method. For one instance of this test set we found a new optimum solution.

The next improvement we want to add to our approach is the so called diversity which is an extension to the Local Branching procedure proposed in [16]. The idea behind this improvement is to avoid the problem of getting stuck in local optima. This is done by setting a time limit on the subproblems defined by the Local Branching process. If the k-neighbourhood of an incumbent solution is searched, and no better solution is found within a certain time, we assume that the optimizer is stuck in a local optimum. In this case we extend the neighbourhood to a maximal distance of 2k from the incumbent solution and thus we increase the probability of finding a better integer solution and leaving the local optimum.

Bibliography

[1] Ilog cplex. http://www.cplex.com.

[2] Giuseppe Andreello, Alberto Caprara, and Matteo Fischetti. Embedding cuts in a branch&cut framework: a computational study with $\{0, \frac{1}{2}\}$-cuts. 2003.

[3] E. Balas. The prize-collecting traveling salesman problem. *Networks*, 19:621–636, 1989.

[4] J.E. Beasley. An SST-based algorithm for the Steiner problem in graphs. *Networks*, 19:1–16, 1989.

[5] D. Bienstock, M. X. Goemans, D. Simchi-Levi, and D. Williamson. A note on the prize collecting traveling salesman problem. *Mathematical Programming*, 59:413–420, 1993.

[6] S. A. Canuto, M. G. C. Resende, and C. C. Ribeiro. Local search with perturbations for the prize-collecting Steiner tree problem in graphs. *Networks*, 38:50–58, 2001.

[7] Alberto Caprara and Matteo Fischetti. {0, 1/2}-Chvátal-Gomory cuts. *Math. Program.*, 74:221–235, 1996.

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

[9] S. Chopra and M. R. Rao. The Steiner tree problem I: Formulations, compositions and extension of facets. *Mathematical Programming*, 64:209–229, 1994.

[10] C. Duin. *Steiner's Problem in Graphs*. PhD thesis, University of Amsterdam, 1993.

[11] C. W. Duin and A. Volgenant. Some generalizations of the Steiner problem in graphs. *Networks*, 17(2):353–364, 1987.

[12] C. W. Duin and A. Volgenant. Reduction tests for the Steiner problem in graphs. *Networks*, 19:549–567, 1989.

[13] S. Engevall, M. Göthe-Lundgren, and P. Värbrand. A strong lower bound for the node weighted Steiner tree problem. *Networks*, 31(1):11–17, 1998.

[14] P. Feofiloff, C.G. Fernandes, C.E. Ferreira, and J.C. Pina. Primal-dual approximation algorithms for the prize-collecting Steiner tree problem. 2003. submitted.

[15] M. Fischetti. Facets of two Steiner arborescence polyhedra. *Mathematical Programming*, 51:401–419, 1991.

[16] M. Fischetti and A. Lodi. Local branching. In *Proceedings of the Integer Programming Conference in honor of Egon Balas*, 2002.

[17] Joe Ganley. Steiner tree bibliography. http://ganley.org/steiner/steinerbib.html.

[18] M. X. Goemans. The Steiner tree polytope and related polyhedra. *Mathematical Programming*, 63:157–182, 1994.

[19] M. X. Goemans and D. P. Williamson. The primal-dual method for approximation algorithms and its application to network design problems. In D. S. Hochbaum, editor, *Approximation algorithms for NP-hard problems*, pages 144–191. P. W. S. Publishing Co., 1996.

[20] F. K. Hwang, D. S. Richards, and P. Winter. *The Steiner Tree Problem*. North-Holland, 1992.

[21] D. S. Johnson, M. Minkoff, and S. Phillips. The prize-collecting Steiner tree problem: Theory and practice. In *Proceedings of 11th ACM-SIAM Symposium on Discrete Algorithms*, pages 760–769, San Francisco, CA, 2000.

[22] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[23] G.W. Klau, I. Ljubić, P. Mutzel, U. Pferschy, and R. Weiskircher. The fractional prize-collecting Steiner tree problem on trees. In G. Di Battista and U. Zwick, editors, *ESA 2003*, volume 2832 of *LNCS*, pages 691–702. Springer-Verlag, 2003.

[24] T. Koch, A. Martin, and S. Voß. SteinLib: An updated library on steiner tree problems in graphs. http://elib.zib.de/steinlib.

[25] A. Lucena and M. G. C. Resende. Strong lower bounds for the prize-collecting Steiner problem in graphs. *Discrete Applied Mathematics*, 2003. to appear.

[26] M. Minkoff. The prize-collecting Steiner tree problem. Master's thesis, MIT, May, 2000.

[27] T. Polzin and S. V. Daneshmand. A comparison of Steiner tree relaxations. *Discrete Applied Mathematics*, 112:241–261, 2001.

[28] T. Polzin and S. Vahdati. Partitioning techniques for the Steiner problem. Technical Report MPI-I-2001-1-1006, Max-Planck-Institut für Informatik, 2001.

[29] T. Radzik. Newton's method for fractional combinatorial optimization. In *Proceedings of 33rd Annual Symposium on Foundations of Computer Science*, pages 659–669, 1992.

[30] I. Rosseti, M. Poggi de Aragão, C. C. Ribeiro, E. Uchoa, and R. F. Werneck. New benchmark instances for the Steiner problem in graphs. In *Extended Abstracts of the 4th Metaheuristics International Conference*, pages 557–561, Porto, Portugal, 2001.

[31] A. Segev. The node-weighted Steiner tree problem. *Networks*, 17:1–17, 1987.

[32] P. Winter. An algorithm for the Steiner problem in the Euclidean plane. *Networks*, 15:323–345, 1985.

[33] P. Winter. Steiner problem in networks: A survey. *Networks*, 17:129–167, 1987.