Primal-Dual Approaches to the Steiner Problem

Tobias Polzin Siavash Vahdati Daneshmand

Theoretische Informatik,
Universität Mannheim, 68131 Mannheim, Germany
email: {polzin,vahdati}@informatik.uni-mannheim.de

Technical Report: Universität Mannheim, 14/2000

Abstract

We study several old and new algorithms for computing lower and upper bounds for the Steiner problem in networks using dual-ascent and primal-dual strategies. These strategies have been proven to be very useful for the algorithmic treatment of the Steiner problem. We show that none of the known algorithms can both generate tight lower bounds empirically and guarantee their quality theoretically; and we present a new algorithm which combines both features.
The new algorithm has running time $O(re \log n)$ and guarantees a ratio of at most two between the generated upper and lower bounds, whereas the fastest previous algorithm with comparably tight empirical bounds has running time $O(e^2)$ without a constant approximation ratio. We show that the approximation ratio two between the bounds can even be achieved in time $O(e + n \log n)$, improving the previous time bound of $O(n^2 \log n)$.
The presented insights can also be helpful for the development of further relaxation based approximation algorithms for the Steiner problem.

Keywords: Steiner problem; relaxation; lower bound; approximation algorithms; dual-ascent; primal-dual

1 Introduction

The Steiner problem in networks is the problem of connecting a set of required vertices in a weighted graph at minimum cost. This is a classical $\mathcal{NP}$-hard problem with many important applications in network design in general and VLSI design in particular (see for example [13]).

For combinatorial optimization problems like the Steiner problem which can naturally be formulated as integer programs, many approaches are based on linear programming. For an $\mathcal{NP}$-hard problem, the optimal value of the linear programming relaxation of such a (polynomially-sized) formulation can only be expected to represent a lower bound on the optimal solution value of the original problem, and the corresponding integrality gap (which we define as the ratio between the optimal values of the integer program and its relaxation) is a major criterion for the utility of a relaxation. For the Steiner problem, we have performed an extensive theoretical comparison of various relaxations in [18].

To use a relaxation algorithmically, many approaches are based on the LP-duality theory. Any feasible solution to the dual of such a relaxation provides a lower bound for the original problem. The classical dual-ascent algorithms construct a dual feasible solution step by step, in each step increasing some dual variables while preserving dual feasibility. A usual approach is ensuring primal complementary slackness conditions and relaxing dual conditions appropriately. As long as a feasible (integral) primal solution complementary slack to the current dual solution is not constructed, a violated primal constraint provides a direction of increase for the dual. This is also the main idea of many recent approximation algorithms based on the primal-dual method, where an approximate solution to the original problem and a feasible solution to the dual of an LP relaxation are constructed simultaneously. The performance guarantee is proved by comparing the values of both solutions. Typical features of these algorithms are simultaneous increasing of several dual variables corresponding to the violated primal constraints; and a clean-up phase to improve the quality of the generated primal feasible solution (for a detailed overview on this see [12]).

In this paper we study some old and new dual-ascent based algorithms for computing lower and upper bounds for the Steiner problem. Two approximation ratios will be of concern in this paper: the ratio between the upper bound and the optimum, and the ratio between the (integer) optimum and the lower bound. The main emphasis will be on lower bounds, with upper bounds mainly used in a primal-dual context to prove a performance guarantee for the lower bounds. Despite the fact that calculating tight lower bounds efficiently is highly desirable (for example in the context of exact algorithms or reduction tests [19, 6, 16]), this issue has found much less attention in the literature. For recent developments concerning upper bounds, see [19].

After some preliminaries, we will discuss in section 2 the classical primal-dual algorithm for the (generalized) Steiner problem based on an undirected relaxation. We give some new insights into this algorithm, which also explain the large empirically observed gaps between the upper and lower bounds produced by this algorithm. In section 3, we study a classical dual-ascent approach based on a directed relaxation, and show that it cannot guarantee a constant approximation ratio for the generated lower (or upper) bounds. In section 4, we introduce a new primal-dual algorithm based on the directed relaxation, analyse its running time, and show that it guarantees a ratio of at most 2 between the upper and lower bounds, while producing tight lower bounds empirically. In each of these sections, we give a short report on the empirical behaviour of the discussed algorithm; detailed computational results are given in an appendix. Section 5 contains some concluding remarks.

Preliminaries

For any undirected graph $G = (V, E)$, we define $n := |V|$, $e := |E|$, and assume that (v_i, v_j) and (v_j, v_i) denote the same (undirected) edge $\{v_i, v_j\}$. A network is here a weighted graph (V, E, c) with an edge weight function $c : E \to \mathbb{R}$. We sometimes refer to networks simply as graphs. For each edge (v_i, v_j), we use terms like cost, weight, length, etc. of (v_i, v_j) interchangeably to denote $c((v_i, v_j))$ (also denoted by $c(v_i, v_j)$ or c_{ij}). For a subgraph H of G, we abuse the notation $c(H)$ to denote the sum of the weights of the edges in H with respect to c. For any directed network $\vec{G} = (V, A, c)$, we use $[v_i, v_j]$ to denote the arc from v_i to v_j; and define $a := |A|$.

The **Steiner problem in networks** can be formulated as follows: Given a network $G = (V, E, c)$ and a non-empty set R, $R \subseteq V$, of **required vertices** (or **terminals**), find a subnetwork $T_G(R)$ of G containing all terminals such that in $T_G(R)$, there is a path between every pair of terminals, and $\sum_{(v_i,v_j)\in T_G(R)} c_{ij}$ is minimized. The directed version of this problem (also called the Steiner arborescence problem) is defined similarly (see [13]). Every instance of the undirected version can be transformed into an instance of the directed version in the corresponding bidirected network, fixing a terminal z_1 as the root. We define $R_1 := R \setminus \{z_1\}$ and $r := |R|$. We assume that $r > 1$. If the terminals are to be

distinguished, they are denoted by $z_1, \ldots, z_r$. The vertices in $V \setminus R$ are called **non-terminals**.

Without loss of generality, we assume that the edge weights are positive and that G (and $T_G(R)$) are connected. Now $T_G(R)$ is a tree. A **Steiner tree** is an acyclic, connected subnetwork of G, including R.

Given a network $G = (V, E, c)$ and a subset $W \subseteq V$ of vertices, we define the **distance network** of W, denoted with $D_G(W)$, as the complete network with the vertex set W and the cost function d_G, where $d_G(v_i, v_j)$, for any two vertices $v_i, v_j \in V$, is defined as the length of a shortest path between v_i and v_j in G. By computing a minimum spanning tree for $D_G(R)$ and replacing its edges with the corresponding paths in G, we get a feasible solution for the original instance; this is the core of a well-known heuristic for the Steiner problem which we call **DNH** (for Distance Network Heuristic; see for example [13]). This heuristic has a worst case performance ratio of $(2 - 2/r)$. Mehlhorn [17] showed how to compute such a tree efficiently by using a concept similar to that of Voronoi regions in algorithmic geometry. For each terminal z, we define a neighborhood $N(z)$ as the set of vertices which are not closer to any other terminal (ties broken arbitrarily). Consider a graph G' with the vertex set R in which two terminals z_i and z_j are adjacent if in G there is a path between z_i and z_j completely in $N(z_i) \cup N(z_j)$, with the cost of the corresponding edge being the length of a shortest such path, i.e. $c'(z_i, z_j) := \min\{d_G(z_i, v_i) + c(v_i, v_j) + d_G(z_j, v_j) \mid v_i \in N(z_i), v_j \in N(z_j)\}$. A minimum spanning tree T' for G' will be also a minimum spanning tree for $D_G(R)$. The neighborhoods $N(z)$ for all $z \in R$, the graph G' and the tree T' can be constructed in total time $O(e + n \log n)$ [17].

A **cut** in $\vec{G} = (V, A, c)$ (or in $G = (V, E, c)$) is defined as a partition $C = (\bar{W}, W)$ of V ($\emptyset \subset W \subset V; V = W \dot{\cup} \bar{W}$). We use $\delta^-(W)$ to denote the set of arcs $[v_i, v_j] \in A$ with $v_i \in \bar{W}$ and $v_j \in W$. The sets $\delta^+(W)$ and, for the undirected version, $\delta(W)$ are defined similarly. For simplicity, we sometimes refer to these sets of arcs (or edges) as cuts. A cut $C = (\bar{W}, W)$ is called a **Steiner cut**, if $z_1 \in \bar{W}$ and $R_1 \cap W \neq \emptyset$ (for the undirected version: $R \cap W \neq \emptyset$ and $R \cap \bar{W} \neq \emptyset$). The (directed) **cut formulation** P_C [22] uses the concept of Steiner cuts to formulate the Steiner problem in a directed network $\vec{G} = (V, A, c)$ (which is, in this context, the bidirected network corresponding to an undirected network $G = (V, E, c)$) as an integer program. In this program, the (binary) vector x represents the incidence vector of the solution, i.e. $x_{ij} = 1$ if the arc $[v_i, v_j]$ is in the solution and 0 otherwise.

$$\boxed{P_C} \quad min \sum_{[v_i,v_j] \in A} c_{ij} x_{ij} \text{ s.t.: } \sum_{[v_i,v_j] \in \delta^-(W)} x_{ij} \geq 1 \quad ((\bar{W}, W) \text{ Steiner cut}); x_{ij} \in \{0,1\} \quad ([v_i, v_j] \in A).$$

The undirected cut formulation P_{UC} is defined similarly [1]. For an integer program like P_C, we denote with LP_C the corresponding linear programming relaxation and with DLP_C the program dual to LP_C. Introducing a dual variable y_W for each Steiner cut $(\bar{W}, W)$, we have:

$$\boxed{DLP_C} \quad max \sum_{(\bar{W},W) \text{ Steiner cut}} y_W \text{ s.t.: } \sum_{W,\, [v_i,v_j] \in \delta^-(W)} y_W \leq c_{ij} \quad ([v_i, v_j] \in A); y_W \geq 0 \quad ((\bar{W}, W) \text{ Steiner cut}).$$

The constraints $\sum_{W,\, [v_i,v_j] \in \delta^-(W)} y_W \leq c_{ij}$ are called the **(cut) packing constraints**.
For any (integer or linear) program Q, we denote with $v(Q)$ the value of an optimal solution for Q.

2 Undirected Cuts: A Primal-Dual Algorithm

Some of the best-known primal-dual approximation algorithms are designed for a class of constrained forest problems which includes the Steiner problem (see [11]). These algorithms are essentially dual-ascent algorithms based on undirected cut formulations, extended by a pruning phase to improve the primal feasible solution. For the Steiner problem, such an algorithm guarantees an upper bound of $2 - 2/r$ on the ratio between the values of the provided primal and dual solutions. This is the best possible guarantee when using the undirected cut relaxation LP_{UC}, since it is easy to construct instances (even with $r = n$) where the ratio $v(P_{UC})/v(LP_{UC})$ is exactly $2 - 2/r$ (see for example [9]). In the following, we briefly describe such an algorithm when restricted to the Steiner problem, show how to make it much faster for this special case, and give some new insights into it. We denote this algorithm with PD_{UC} (PD stands for Primal-Dual and UC stands for Undirected Cut). For a detailed description of the general algorithm, see [11].

The algorithm maintains a forest F, which is initially empty (i.e. the forest consists of isolated vertices of V). A connected component S of F is called an active component if $(\bar{S}, S)$ defines a Steiner cut. In each iteration, dual variables corresponding to active components are increased uniformly until a new packing constraint becomes tight, i.e. the reduced cost $c_e - \sum_{(\bar{S},S) \text{ Steiner cut}} y_S$ of some edge e becomes zero, which is then added to F (ties are broken arbitrarily). Note that this modifies the connected components

of F (only edges between distinct components may be added to F). The algorithm terminates when no active component is left; at this time, F defines a feasible Steiner tree and $\sum_{(\bar{S},S)\text{ Steiner cut}} y_S$ represents a lower bound on the weight of any Steiner tree for the observed instance. In a subsequent pruning phase, every edge of F which is not on a path (in F) between two terminals is removed. The resulting Steiner tree T after this phase is returned by the algorithm.
In [11], it is shown how to make this algorithm (for the generalized problem) run in $O(n^2 \log n)$ time; see also [7, 14] for some improvements. The performance guarantee of 2 can be proven by observing that in each iteration, the number of edges in all cuts corresponding to active components which are also in T is at most twice the number of active components, or more precisely: $\sum_{S \text{ active}} |T \cap \delta(S)| \leq (2 - \frac{2}{r})|\{\text{active components of } F\}|$. Using this and since the edges of T have zero reduced costs (satisfy the primal complementary slackness conditions), it follows that $c(T) \leq (2 - \frac{2}{r}) \sum_{(\bar{S},S)\text{ Steiner cut}} y_S$.

When restricted to the Steiner problem and as far as the constructed Steiner tree T is considered, the algorithm PD_{UC} is essentially the well-known DNH (Distance Network Heuristic) described in section 1, implemented by an interleaved computation of shortest paths trees out of terminals and a minimum spanning tree for the terminals with respect to their distances. In fact, every Steiner tree T provided by the Mehlhorn's $O(e+n \log n)$ time implementation of DNH can be considered as a possible result of PD_{UC}. We observed that even the lower bound calculation can be performed in the same time. Let T' be a minimum spanning tree for R provided by the Mehlhorn's implementation of DNH and let $e'_1, \ldots, e'_{r-1}$ be its edges in nondecreasing cost order. The algorithm PD_{UC} increases all dual variables corresponding to the initially r active components by $\frac{c'(e'_1)}{2}$, then the components corresponding to the vertices of e'_1 are merged. The dual variables of the remaining $r-1$ components are increased by $\frac{c'(e'_2)-c'(e'_1)}{2}$ (which is possibly zero) before the next two components are merged, and so on. Therefore, the lower bound provided by PD_{UC} is (defining $c'(e'_0) := 0$) simply $\sum_{i=1}^{r-1}(r-i+1)\frac{c'(e'_i)-c'(e'_{i-1})}{2} = \frac{1}{2}(c'(e'_{r-1}) + \sum_{i=1}^{r-1} c'(e'_i)) = \frac{1}{2}(c'(e'_{r-1}) + c'(T'))$, which can be easily computed in $O(r)$ time once T' is available. So we can compute both the upper and the lower bound provided by PD_{UC} in $O(e + n \log n)$ time.

Probably the most interesting insight we get from this new viewpoint at PD_{UC} is about the gap between the provided upper and lower bounds. Assuming that the cost of T' is not dominated by the cost of its longest edge and that the Steiner tree corresponding to T' is not much cheaper than T' itself (which is usually the case), the ratio between the upper and lower bound is nearly two; and this suggests that either the lower bound, or the upper bound, or both are not really tight.

For later comparison, let us summarize the worst case results we know about the algorithm PD_{UC}. For any instance of the Steiner problem, denote with *optimum* the value of an optimal solution and with *upper* and *lower* the upper and lower bounds calculated by PD_{UC}; and define $relaxed := v(LP_{UC})$. Certainly $lower \leq relaxed \leq optimum \leq upper$. We know that $upper/lower \leq 2 - 2/r$; and that each of the ratios $upper/optimum$, $optimum/relaxed$ and $relaxed/lower$ can be as large as $2 - 2/r$ (for the last ratio, consider r $(= n)$ terminals on a chain with equal edge lengths).

Empirically, results on different types of instances show an average gap of about 45% (of *optimum*, or about 70% of *lower*) between the the upper and the lower bounds calculated by PD_{UC} (see the appendix). This is in accordance with the relation we established above between these two values. This gap is mainly due to the lower bounds, where the gap to optimum is typically over 30%. So although this heuristic can be implemented to be very fast empirically (small fractions of a second even for fairly large instances), it is not suitable for computing tight bounds, as needed (for example) in the context of exact algorithms.

3 Directed Cuts: An Old Dual-Ascent Algorithm

In the search for an approach for computing tighter lower and upper bounds, the directed cut relaxation is a promising alternative. Although no better upper bound than the $2 - 2/r$ one from the previous section is known on the integrality gap of this relaxation, the gap is conjectured to be much closer to 1, and the worst instance known has an integrality gap of approximately 8/7 [8]. There are many theoretical and empirical investigations which indicate that the directed relaxation is (at least usually) a much stronger relaxation than the undirected one (see for example [3, 4]). In [19], we could achieve impressive empirical results (including extremely tight lower and upper bounds) using this relaxation. In that work, extensions of a dual ascent algorithm of Wong [22] played a major role. Although many works on the Steiner problem use variants of this heuristic (see for example [6, 13, 21]), none of them includes a discussion of the theoretical quality of the generated lower (and, sometimes, upper) bounds. In this section, we show that none of these variants can guarantee a constant approximation ratio for the generated lower or upper bounds.

The dual-ascent algorithm in [22] is described for the directed Steiner problem using the equivalent multicommodity flow relaxation. Here we give a short alternative description of it as a dual-ascent algorithm for LP_C, which we denote with DA_C. The algorithm maintains a set H of arcs with zero reduced costs, which is initially empty. For each terminal $z_t \in R_1$, define the component of z_t as the set of all vertices for which there exists a directed path to z_t in H. A component is said to be active if it does not contain the root. In each iteration, an active component is chosen (this choice is discussed later) and the dual variable of the corresponding Steiner cut is increased until the packing constraint for an arc in this cut becomes tight. Then the reduced costs of the arcs in the cut are updated and the arcs with reduced cost zero are added to H. The algorithm terminates when no active component is left; at this time, H (regarded as a subgraph of G) is a feasible solution for the observed instance of the (directed) Steiner problem. To get a (directed) Steiner tree, in [22] the following method is suggested: Let Q be the set of vertices reachable from z_1 in H. Compute a minimum directed spanning tree for the subgraph of G induced by Q and prune this tree until all its leaves are terminals. In [13], this method is adapted to the undirected version, mainly by computing a minimum (undirected) spanning tree instead of a directed one. For the empirical results in this paper, we use this modified version.

The original work of Wong [22] contains no discussion of the (worst case) running time. In [6], an implementation of DA_C with running time $O(a \min\{a, rn\})$ is described. Actually, the algorithm is usually much faster than this bound would suggest. Nevertheless, we have constructed instances on which every dual-ascent algorithm following the same scheme must perform $\Theta(n^4)$ operations.

To show that the lower bound generated by DA_C can deviate arbitrarily from $v(LP_C)$, two difficulties must be considered. The first one is the choice of the root: although the value $v(LP_C)$ for an instance of (undirected) Steiner problem is independent of the choice of the root (see for example [10]), the lower bound generated by DA_C is not, so the argumentation must be independent of this choice. The second difficulty is the choice of an active component in each iteration. In the original work of Wong [22], the chosen component is merely required to be a so-called root component. A component S corresponding to a terminal z_t is called a root component if for any other terminal z_s in this component, there is a path from z_t to z_s in H. This is equivalent to S being a minimal (with respect to inclusion) active component. An empirically more successful variant uses a size criterion: at each iteration, an active component of minimum size is chosen (see [6, 19]). Note that such a component is always a root component. So, in this context it is sufficient to study the variant based on the size criterion.

Example 1 In the figure 1, there are $c^2 + c + 1$ terminals (filled circles); the top terminal is considered as the root. The edges incident with the left c terminals have costs c^2, all the other edges have costs c. According to the size criterion, each of the terminals (i.e. their components) at the left is chosen twice before any of the terminals at the bottom can be chosen a second time. But then, there is no active component anymore and the algorithm terminates. So, the lower bound generated by DA_C is in $\Theta(c^3)$. On the other hand, it is easy to see that for this instance: $v(LP_C) = v(P_C) \in \Theta(c^4)$.
Now imagine c copies of this graph sharing the top terminal (not necessarily the root anymore). For the resulting instance, we have $v(LP_C) = v(P_C) \in \Theta(c^5)$; but the lower bound generated by DA_C will be in $\Theta(c^4)$ independent of the choice of the root, because the observation above will remain valid in at least $c - 1$ copies.

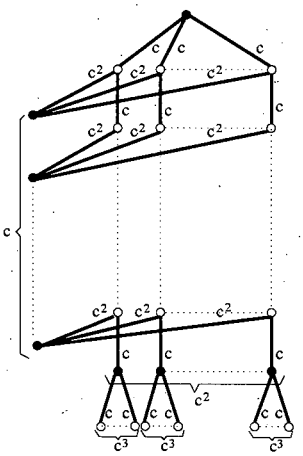

Figure 1: Arbitrarily bad case for DA_C

Turning to upper bounds, the observation above already shows that no constant performance ratio for DA_C can be proven using a primal-dual technique. Now we show that no such result could be achieved using an alternative proof technique. For any instance of the Steiner problem, let *optimum* be the value of an optimal solution and *upper* the upper bound calculated by the algorithm in this section. By changing the costs of the edges incident to the left terminals from c^2 to $c + \epsilon$ (for a small ϵ) in the figure 1, we get an instance for which the ratio *upper*/*optimum* can be arbitrarily large (this is also the case for all other approaches for computing upper bounds based on the graph H provided by DA_C in the literature, including those in [21, 22]). In fact, it is easy to see that *optimum* $\in \Theta(c^3)$ for such an instance, but there is no solution with cost $o(c^4)$ in the subgraph H generated by the algorithm DA_C.

Finally, let us turn to the empirical behaviour of the algorithm DA_C. Despite its bad performance in the worst case, the algorithm typically provides fairly tight lower bounds, with average gaps ranging from a small fraction of a percent to about 2%, depending on the type of instances (see the appendix). The upper bounds are not good, with average gaps from 8% to 30%, again depending on the type of

instances. The running times, although much larger than those of PD_{UC}, are still quite tolerable (about a second even for fairly large instances).

4 Directed Cuts: A New Primal-Dual Algorithm

The previously described heuristics had complementary advantages: The first, PD_{UC}, guarantees an upper bound of 2 on the ratio between the generated upper and lower bounds, but empirically, it does not perform much better than in the worst case. The second one, DA_C, cannot provide such a guarantee, but empirically it performs much better than the first one, especially for computing lower bounds. For the first time we were able to design a heuristic which combines both features.

The straightforward application of the primal-dual method of PD_{UC} (simultaneous increasing of all dual variables corresponding to active components and merging components which share a vertex) to the directed cut relaxation leads to an algorithm with performance ratio 2 and running time $O(e + n \log n)$, but the generated lower bounds are again not nearly as tight as those provided by DA_C.

The main idea for a successful new approach is not to merge the components, but to let them grow as long as they are (minimally) active. As a consequence, dual variables corresponding to several cuts which share the same arc may be increased simultaneously.

Using this idea directly in a primal-dual context is inhibited by a certain kind of imbalance: The reduced costs of arcs which are in the cuts of many active components are decreased much faster than the other ones. Because of that, we have been able to construct a problem instance where a straightforward primal-dual algorithm based on this approach fails to give a performance ratio of two. Therefore, we group all components that share a vertex together and postulate that in each iteration, the total increase Δ of dual variables corresponding to each *group* containing at least one active component must be the same. Note that the groups are vertex-disjoint. If we denote the number of active components in a group Γ with $activesInGroup(\Gamma)$, the dual variable corresponding to each of these components will be increased by $\Delta / activesInGroup(\Gamma)$. Similar to the case of DA_C, a component is called active if it does not contain the root or include another active component (ties are broken arbitrarily). A terminal is called active if its component is active; and a group is called active if it contains an active terminal (by this definition it is guaranteed that each active root component corresponds to one active terminal). If we denote with *activeGroups* the number of active groups, the lower bound *lower* will be increased in each iteration by $\Delta \cdot$ *activeGroups*.

To manage the reduced costs efficiently, a concept like that of distance estimates in the algorithm of Dijkstra is used (see for example [5]). For each arc x, the value $d(x)$ estimates the value of *dGroup* (amount of uniform increase of group duals, i.e. the sum of Δ-values) which would make x tight (set its reduced cost $\bar{c}(x)$ to zero). For an arc x with reduced cost $\bar{c}(x) > 0$, all active components S with $x \in \delta^-(S)$ will be in the same group Γ. If there are $activesOnArc(x)$ such components, then $d(x)$ should be $\bar{c}(x) \cdot activesInGroup(\Gamma) / activesOnArc(x) + dGroup$.

For updating the d-values we use two further variables for each arc x: $reducedCost(x)$ and $lastReducedCostUpdate(x)$; they are initially set to $c(x)$ and 0, respectively. If $activesOnArc(x)$ and/or $activesInGroup(\Gamma)$ change, the new value for $d(x)$ can be calculated by: $reducedCost(x) := reducedCost(x) - (dGroup - lastReducedCostUpdate(x)) \cdot activesOnArc_{old}(x) / activesInGroup_{old}(\Gamma)$;
$d(x) := reducedCost(x) \cdot activesInGroup_{new}(\Gamma) / activesOnArc_{new}(x) + dGroup$;
$lastReducedCostUpdate(x) := dGroup$.

Below we give a description of the algorithm PD_C in pseudocode. For readability, we have introduced some macros (a call to a macro is to be simply replaced by its body). The algorithm uses the following basic data structures (see [5]): A priority queue PQ manages the arcs using the d-values as keys. A disjoint-set data structure *Groups* is used to mange the groups. Two lists $\vec{H}$ and H are used to store tight arcs and the corresponding edges; new elements are appended at the end of the list. A Stack *Stack* is used to perform depth-first searches from vertices newly added to a component. Also some arrays are used: $visited[z, v]$ denotes whether the vertex v is in the component of the terminal z; $firstSeenFrom[v]$ gives the first terminal whose component has reached the vertex v; $active[z]$ denotes whether the terminal z is active; $d[x]$ gives the d-value of the arc x; $activesInGroup[\Gamma]$ gives the number of the active components in the group Γ; and $activesOnArc[x]$ gives the number of components which have x in their corresponding cuts. The variables Δ, *activeGroups*, *lower*, and *dGroup* are used as described above.

$PD_C(G, R, z_1)$
1 *initialize PQ, Groups, H, $\vec{H}$;*
2 **forall** $z \in R_1$: "initializing the components"
3 *Groups.MAKE-SET*(z); $activesInGroup[z] := 1$; $active[z] :=$ *TRUE*;

```
4     forall x ∈ δ⁻(z) :
5       activesOnArc[x] := 1; d[x] := c(x); PQ.INSERT(x, d[x]);
6     forall v ∈ V : visited[z, v] := FALSE;
7     visited[z, z] := TRUE; firstSeenFrom[z] := z;
8   forall v ∈ V \ R : firstSeenFrom[v] := 0;
9   activeGroups := r − 1; dGroup := 0; lower := 0;
10  while activeGroups > 0 :
11    x := [v_i, v_j] := PQ.EXTRACT-MIN();      "get the next arc becoming tight"
12    Δ := d[x]−dGroup; dGroup := d[x]; lower := lower + Δ· activeGroups;
13    mark [v_i, v_j] as tight;
14    if (v_i, v_j) is not in H :      "i.e. [v_j, v_i] is not tight"
15      H.APPEND((v_i, v_j)); H⃗.APPEND([v_i, v_j]);
16    z_i := firstSeenFrom[v_i]; z_j := firstSeenFrom[v_j];
17    if z_i = 0 : firstSeenFrom[v_i] := z_j;
18    else if Groups.FIND(z_i) ≠ Groups.FIND(z_j) : MERGE-GROUPS(z_i, z_j);
19    forall active z ∈ R_1 :
20      if visited[z, v_j] and not visited[z, v_i] : EXTEND-COMPONENT(z, v_i);
21  H' := H; H⃗' := H⃗;
22  PRUNE(H', H⃗');
23  return H', lower;      "upper: the cost of H' "

MERGE-GROUPS(z_i, z_j)
1   g_i := Groups.FIND(z_i); g_j := Groups.FIND(z_j);
2   if activesInGroup[g_i] > 0 and activesInGroup[g_j] > 0 :
3     update in PQ the keys of all arcs entering these groups;
4     activeGroups := activeGroups−1;
5   Groups.UNION(g_i, g_j);
6   activesInGroup[Groups.FIND(g_i)] :=activesInGroup[g_i] + activesInGroup[g_j];

EXTEND-COMPONENT(z, v_i)
1   Stack.INIT(); Stack.PUSH(v_i);      "modified depth-first search starting from v_i"
2   while not Stack.EMPTY() :
3     v :=Stack.POP();
4     if (v = z_1) or (v ∈ R \ {z} and active[v]) :
5       REMOVE-COMPONENT(z);
6       break;
7     if not visited[z, v] :
8       visited[z, v] := TRUE;
9       forall [v, w] ∈ δ⁺(v) :
10        if visited[z, w] :
11          activesOnArc[[v, w]] :=activesOnArc[[v, w]] − 1;
12          update the key of [v, w] in PQ;
13        else :
14          if [w, v] is already tight : Stack.PUSH(w);
15          else :
16            activesOnArc[[w, v]] :=activesOnArc[[w, v]] + 1;
17            update the key of [w, v] in PQ;

REMOVE-COMPONENT(z)
1   active[z] :=FALSE;
2   g :=Groups.FIND(z);
3   update in PQ the keys of all arcs entering g or the component of z;
4   activesInGroup[g] := activesInGroup[g] − 1;
5   if activesInGroup[g] = 0 : activeGroups := activeGroups −1;

PRUNE(H', H⃗')
1   forall [v_i, v_j] in H⃗', in reverse order :
2     if H' without (v_i, v_j) connects all terminals :
3       H'.DELETE((v_i, v_j)); H⃗'.DELETE([v_i, v_j]);
```

The running time of the algorithm PD_C can be estimated as follows. The initializations in lines 1-9 need obviously $O(rn + a \log n)$ time. The loop in the lines 10-20 is repeated at most a times, because each time an arc becomes tight and there will be no active terminal (group) when all arcs are tight. Assume for now that each basic arithmetic operation can be done in constant time; this issue is discussed later. In each iteration, line 11 needs $O(\log n)$ time and lines 12-20 excluding the macros MERGE-GROUPS and EXTEND-COMPONENT can be performed in $O(r)$ time, this gives a total time of $O(a(r + \log n))$ for all iterations excluding the macros. Each execution of MERGE-GROUPS need $O(a \log n)$ time and there can be at most $r - 1$ such executions; the same is true for REMOVE-COMPONENT. For each terminal, the adjacency list of each vertex is considered only once over all executions of EXTEND-COMPONENT, so each arc is considered (and its key is updated in PQ) at most twice for each terminal, leading to a total time of $O(ra \log n)$ for all executions of EXTEND-COMPONENT. So the lines 1-20 can be executed in $O(ra \log n)$ time.

It is easy to observe that the reverse order deletion in PRUNE can be performed efficiently by the following procedure: Consider a graph $\tilde{H}$ with the edge set H in which the weight of each edge $\tilde{e}$ is the position $p(\tilde{a})$ of the corresponding arc $\tilde{a}$ in the list $\vec{H}$. Let T' be the (edge set of a) tree generated by computing a minimum spanning tree for $\tilde{H}$ and pruning it until it has only terminals as leaves. Then we have: $T' = H'$. Since the edges of $\tilde{H}$ are already available in a sorted list, the minimum spanning tree can be computed even in $O(e\ \alpha(e, n))$ time. This leads to a total time of $O(ra \log n)$ for PD_C.

Below we show that the ratio between the upper bound *upper* and the lower bound *lower* generated by PD_C is at most 2, thus guaranteeing an approximation ratio of 2 both for the upper bound and the lower bound calculation.

Let $\vec{T}$ be (the arcs of) the directed tree obtained by rooting H' at z_1. For each component S, we denote with *activesInGroupOf*(S) the total number of active components in the group of S. The variables *activesInGroup* and *activeGroups* are used as defined before.

Lemma 1 At the beginning of each iteration in the algorithm PD_C, it holds:

$$\sum_{S \text{ active}} \frac{|\vec{H}' \cap \delta^-(S)|}{activesInGroupOf(S)} \leq (2 - \frac{1}{r-1}) \cdot activeGroups.$$

Proof: First, we state several invariants which are valid at the beginning of each iteration in PD_C:

(1) All vertices in a group are connected by the edges currently in H.

(2) For each active group Γ, at most one arc of $\delta^-(\Gamma)$ will belong to $\vec{T}$, since all but one edge in $\delta(\Gamma) \cap H$ will be removed by PRUNE because of (1). So $\vec{T}$ will still be a tree if for each active group Γ, all arcs which begin and end in Γ are contracted.

(3) For each group Γ and each active component $S \subseteq \Gamma$, no arc $[v_i, v_j] \in \delta^-(S)$ with $v_i, v_j \in \Gamma$ will be in $\vec{H}'$, since it is not yet in $\vec{H}$ (otherwise it would not be in $\delta^-(S)$) and if it is added to $\vec{H}$ later, it will be removed by PRUNE because of (1).

(4) For each active group Γ and each arc $[v_i, v_j] \in \vec{T} \cap \delta^+(\Gamma)$, there is at least one active terminal in the subtree $\vec{T}_j$ of $\vec{T}$ with the root v_j. Otherwise (v_i, v_j) would be removed by PRUNE, because all terminals in $\vec{T}_j$ are already connected to the root by edges in H.

(5) Because of (2), (4) and since at least one arc in $\vec{T}$ leaves z_1, it holds:
$\sum_{\Gamma \text{ active group}} |\vec{T} \cap \delta^-(\Gamma)| \geq 1 + \sum_{\Gamma \text{ active group}} |\vec{T} \cap \delta^+(\Gamma)|$.

(6) Because of (3), for each active group Γ and each subset B of $\vec{H}'$ holds:
$\sum_{S \subseteq \Gamma,\ S \text{ active}} |B \cap \delta^-(S)| \leq activesInGroup(\Gamma) \cdot |B \cap \delta^-(\Gamma)|$.

We split $\vec{H}'$ into $\vec{H}' \cap \vec{T}$ and $\vec{H}' \backslash \vec{T}$. Observe that $\vec{H}'$ and $\vec{T}$ can only differ in the direction of some arcs. Therefore, $\vec{H}' \backslash \vec{T}$ is just $\vec{T} \backslash \vec{H}'$ with reversed arcs. Now we have:

$$\begin{aligned}
\sum_{S \text{ active}} \frac{|\vec{H}' \cap \delta^-(S)|}{activesInGroupOf(S)} &= \sum_{\Gamma \text{ active group}} \sum_{S \text{ active},\ S \subseteq \Gamma} \frac{|\vec{H}' \cap \delta^-(S)|}{activesInGroup(\Gamma)} \\
&\leq \sum_{\Gamma \text{ active group}} |\vec{H}' \cap \delta^-(\Gamma)| \qquad \text{(because of (6))} \\
&= \sum_{\Gamma \text{ active group}} |\vec{H}' \cap \vec{T} \cap \delta^-(\Gamma)| + |(\vec{T} \backslash \vec{H}') \cap \delta^+(\Gamma)|
\end{aligned}$$

$$\leq \left(\sum_{\Gamma \text{ active group}} |\vec{H}' \cap \vec{T} \cap \delta^-(\Gamma)| + |\vec{T} \cap \delta^-(\Gamma)| \right) - 1 \qquad \text{(because of (5))}$$

$$\leq 2 \cdot activeGroups - 1. \qquad \text{(because of (2))}$$

Because $activeGroups \leq r - 1$ this proves the lemma. □

Theorem 2 Let *upper* and *lower* be the upper and the lower bound generated by PD_C. It holds that:

$$\frac{upper}{lower} \leq (2 - \frac{1}{r-1}).$$

Proof: Let Δ_i be the value of Δ in the iteration i. For each directed Steiner cut $(\bar{S}, S)$, let y_S be the value of the corresponding dual variable as (implicitly) calculated by PD_C (as described before, in iteration i each dual variable y_S corresponding to an active component S is increased by $\Delta_i / activesInGroupOf(S)$). Since all arcs of $\vec{H}'$ have zero reduced costs, we have: $upper = \sum_{x \in \vec{H}'} c(x) = \sum_{x \in \vec{H}'} \sum_{S,\ x \in \delta^-(S)} y_S = \sum_S |\vec{H}' \cap \delta^-(S)| \cdot y_S$. This value is zero at the beginning and is increased by $\sum_S$ active $|\vec{H}' \cap \delta^-(S)| \cdot \Delta_i / activesInGroupOf(S)$ in the iteration i. By lemma 3, this increase is at most $(2 - \frac{1}{r-1}) \cdot activeGroups \cdot \Delta_i$. Since *lower* is zero at the beginning and is increased exactly by $activeGroups \cdot \Delta_i$ in the iteration i, we have $upper \leq (2 - \frac{1}{r-1}) \cdot lower$ after the last iteration. □

We found examples which show that the approximation ratio is tight for the upper bound as well as for the lower bound.

The discussion above assumes exact real arithmetic. Even if we adopt the (usual) assumption that all numbers in the input are integers, using exact arithmetic could deteriorate the worst case running time due to the growing denominators and using floating-point numbers is not appropriate due to the unpredictable roundoff errors. But if we allow a deterioration of ϵ (for a small constant ϵ) in the approximation ratio, we can solve this problem by an appropriate fixed-point representation of all numbers.

Empirically, this algorithm behaves similarly to DA_C. The lower bounds are again fairly tight, with average gaps from a fraction of a percent to about 2%, depending on the type of instances (see the appendix). The upper bounds, although more stable than those of DA_C, are not good; the average gaps are about 8%. The running times are, depending on the type of instances, sometimes better and sometimes worse than those of DA_C, altogether they are still tolerable (several seconds for large and dense graphs).

5 Concluding Remarks

In this article, we have studied some old and new LP-duality based algorithms for computing lower and upper bounds for the Steiner problem in networks. Among other things, we have shown that none of the known algorithms both generates tight lower bound empirically and guarantees their quality theoretically; and we have presented a new algorithm which combines both features.

One major point remains to be improved: The approximation ratio of 2. Assuming that the integrality gap of the directed cut relaxation is well below 2, an obvious desire is to develop algorithms based on it with a better worst case ratio between the upper and lower bounds (thus proving the assumption). There are two major approaches for devising approximation algorithms based on linear programming relaxations: LP-rounding and primal-dual schema. A discussion in [20] indicates that no better guarantee can be obtained using a standard LP-rounding approach based on this relaxation. The discussion in this paper indicates the same for a standard primal-dual approach. Thus, to get a better ratio, extensions of the primal-dual schema will be needed. Two such extensions are used in [20], where a ratio of 3/2 is proven for the special class of quasi-bipartite graphs.

References

[1] Y. P. Aneja. An integer linear programming approach to the Steiner problem in graphs. *Networks*, 10:167–178, 1980.

[2] J. E. Beasley. OR-Library. `http://graph.ms.ic.ac.uk/info.html`, 1990.

[3] S. Chopra, E. R. Gorres, and M. R. Rao. Solving the Steiner tree problem on a graph using branch and cut. *ORSA Journal on Computing*, 4:320–335, 1992.

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

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

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

[7] H. N. Gabow, M. X. Goemans, and D. P. Williamson. An efficient approximation algorithm for the survivable network design problem. In *Proceedings 3rd Symposium on Integer Programming and Combinatorial Optimization*, pages 57–74, 1993.

[8] M. X. Goemans. Personal communication, 1998.

[9] M. X. Goemans and D. J. Bertsimas. Survivable networks, linear programming relaxations and the parsimonious property. *Mathematical Programming*, 60:145–166, 1993.

[10] M. X. Goemans and Y. Myung. A catalog of Steiner tree formulations. *Networks*, 23:19–28, 1993.

[11] M. X. Goemans and D. P. Williamson. A general approximation technique for constrained forest problems. *SIAM Journal on Computing*, 24(2):296–317, 1995.

[12] M. X. Goemans and D. P. Williamson. The primal-dual method for approximation algorithms and its application to network design problem. In D. S. Hochbaum, editor, *Approximation Algorithms for NP-hard Problems*. PWS Publishing Company, 1996.

[13] F. K. Hwang, D. S. Richards, and P. Winter. *The Steiner Tree Problem*, volume 53 of *Annals of Discrete Mathematics*. North-Holland, Amsterdam, 1992.

[14] P. N. Klein. A data structure for bicategories, with application to speeding up an approximation algorithm. *Information Processing Letters*, 52(6):303–307, 1994.

[15] T. Koch and A. Martin. SteinLib. `ftp://ftp.zib.de/pub/Packages/mp-testdata/steinlib/index.html`, 1997.

[16] T. Koch and A. Martin. Solving Steiner tree problems in graphs to optimality. *Networks*, 32:207–232, 1998.

[17] K. Mehlhorn. A faster approximation algorithm for the Steiner problem in graphs. *Information Processing Letters*, 27:125–128, 1988.

[18] T. Polzin and S. Vahdati Daneshmand. A comparison of Steiner tree relaxations. Technical Report 5/1998, Universität Mannheim, 1998. (to appear in Discrete Applied Mathematics).

[19] T. Polzin and S. Vahdati Daneshmand. Improved Algorithms for the Steiner Problem in Networks. Technical Report 06/1998, Universität Mannheim, 1998. (to appear in Discrete Applied Mathematics).

[20] S. Rajagopalan and V. V. Vazirani. On the bidirected cut relaxation for the metric Steiner tree problem. In *Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms*, page ?, 1999.

[21] S. Voß. Steiner's problem in graphs: Heuristic methods. *Discrete Applied Mathematics*, 40:45–72, 1992.

[22] R. T. Wong. A dual ascent approach for Steiner tree problems on a directed graph. *Mathematical Programming*, 28:271–287, 1984.

Appendix 1: Empirical Results on Benchmark Instances

We give empirical results on the quality of the lower and the upper bounds and the running times for the different algorithms (PD_{UC}, DA_C, and PD_C) discussed in this paper using benchmark instances. There are two major benchmark libraries for the Steiner problem in networks: the collection in the OR-Library [2] and SteinLib [15]. We have chosen the group with the largest instances (2500 vertices) from the OR-Library and a group of VLSI-instances (up to 6836 vertices) from SteinLib for the comparisons in this paper. Our experiments on other groups of instances have shown that the (relative) behaviour of the discussed algorithms is fairly well represented by this choice.

For each algorithm (PD_{UC}, DA_C, and PD_C) and for each group of instances, a table is presented where for each instance, the value of an optimal solution (*optimum*), the lower and the upper bound calculated by the respective algorithm (*lower, upper*) and the time spent are given. In addition, at the bottom of each table the average gaps (defined as $(upper - optimum)/optimum$ and $(optimum - lower)/optimum$) and the average running times are given. The tests were performed using a Pentium II 450 MHz processor.

instance	time (in sec.)	*lower*	*optimum*	*upper*
steinE01	0.01	92	111	125
steinE02	0.01	164	214	255
steinE03	0.03	2405	4013	4251
steinE04	0.03	2920	5101	5360
steinE05	0.02	4442	8128	8367
steinE06	0.02	59	73	86
steinE07	0.01	112	145	169
steinE08	0.03	1522	2640	2867
steinE09	0.03	2039	3604	3881
steinE10	0.03	3015	5600	5872
steinE11	0.05	27	34	39
steinE12	0.03	46	67	71
steinE13	0.06	738	1280	1461
steinE14	0.07	969	1732	1917
steinE15	0.06	1492	2784	3025
steinE16	0.22	12	15	19
steinE17	0.23	17	25	29
steinE18	0.32	345	564	725
steinE19	0.34	418	758	862
steinE20	0.33	684	1342	1373
average:	0.10	35.6%		12.1%

Table 1: PD_{UC} on E-instances

instance	time (in sec.)	*lower*	*optimum*	*upper*
steinE01	0.03	111	111	111
steinE02	0.02	214	214	272
steinE03	0.63	4009	4013	4035
steinE04	1.53	5096	5101	5132
steinE05	2.96	8128	8128	8129
steinE06	0.03	73	73	73
steinE07	0.05	145	145	147
steinE08	0.78	2634	2640	2671
steinE09	1.31	3600	3604	3626
steinE10	2.76	5599	5600	5612
steinE11	0.03	34	34	34
steinE12	0.06	66	67	81
steinE13	0.92	1274	1280	1347
steinE14	1.41	1730	1732	1779
steinE15	2.96	2784	2784	2802
steinE16	0.14	15	15	25
steinE17	0.09	25	25	28
steinE18	0.79	551	564	672
steinE19	1.44	754	758	863
steinE20	1.64	1342	1342	1396
average:	0.98	0.27%		8.84%

Table 2: DA_C on E-instances

instance	time (in sec.)	*lower*	*optimum*	*upper*
steinE01	0.01	111	111	127
steinE02	0.04	213	214	235
steinE03	0.73	3999	4013	4266
steinE04	0.92	5089	5101	5349
steinE05	1.73	8101	8128	8334
steinE06	0.05	73	73	78
steinE07	0.06	145	145	152
steinE08	1.06	2623	2640	2875
steinE09	1.55	3595	3604	3820
steinE10	2.28	5587	5600	5806
steinE11	0.10	34	34	40
steinE12	0.11	66	67	82
steinE13	2.87	1270	1280	1421
steinE14	3.84	1723	1732	1843
steinE15	4.77	2772	2784	2879
steinE16	0.74	15	15	16
steinE17	0.90	25	25	28
steinE18	17.7	554	564	647
steinE19	15.7	741	758	814
steinE20	12.8	1340	1342	1358
average:	3.41	0.49%		8.55%

Table 3: PD_C on E-instances

instance	time (in sec.)	*lower*	*optimum*	*upper*
taq0014	0.06	2960	5326	5671
taq0023	0.01	392	621	644
taq0365	0.03	1182	1914	1991
taq0377	0.07	3664	6393	6849
taq0431	0.01	570	897	950
taq0631	0.01	387	581	667
taq0739	0.01	515	848	918
taq0741	0.01	514	847	908
taq0751	0.01	601	939	1032
taq0891	0.01	240	319	332
taq0903	0.04	2902	5099	5503
taq0910	0.01	232	370	391
taq0920	0.01	127	210	215
taq0978	0.01	385	566	574
average:	0.02	37.38%		6.35%

Table 4: PD_{UC} on TAQ-instances

instance	time (in sec.)	*lower*	*optimum*	*upper*
taq0014	1.64	5238	5326	6975
taq0023	0.02	610	621	840
taq0365	0.52	1870	1914	2352
taq0377	1.82	6197	6393	8465
taq0431	0.06	881	897	1490
taq0631	0.03	557	581	775
taq0739	0.05	815	848	1160
taq0741	0.02	835	847	1237
taq0751	0.05	897	939	1154
taq0891	0.01	316	319	489
taq0903	1.51	4950	5099	6546
taq0910	0.01	370	370	407
taq0920	0.01	210	210	232
taq0978	0.03	561	566	596
average:	0.41	2.09%		31.01%

Table 5: DA_C on TAQ-instances

instance	time (in sec.)	*lower*	*optimum*	*upper*
taq0014	0.65	5245	5326	5782
taq0023	0.02	612	621	641
taq0365	0.13	1870	1914	2107
taq0377	0.82	6183	6393	6956
taq0431	0.02	874	897	947
taq0631	0.01	560	581	632
taq0739	0.03	815	848	930
taq0741	0.01	834	847	926
taq0751	0.03	906	939	1034
taq0891	0.01	309	319	337
taq0903	0.59	4949	5099	5504
taq0910	0.01	369	370	383
taq0920	0.01	210	210	217
taq0978	0.01	561	566	571
average:	0.17	2.21%		6.82%

Table 6: PD_C on TAQ-instances

Appendix 2: Additional Explanations

In this appendix we present some technical proofs, examples and comments which round off the main results presented in the body of the paper.

Alternative derivation of the bounds provided by PD_{UC} (section 2)

Using our line of argumentation in section 2, the results concerning the bounds provided by PD_{UC} can be proven without the notion of primal-dual algorithms altogether:

Lemma 3 Let T_{opt} be an optimal Steiner tree for an instance (G, R) of the Steiner problem and T' a minimum spanning tree in the distance network $D_G(R)$ with edges $e'_1, \ldots, e'_{r-1}$ as described before. Define $L := \frac{1}{2}(c'(e'_{r-1}) + c'(T'))$. It holds: $L \leq c(T_{opt}) \leq c'(T') \leq (2 - \frac{2}{r})L$.

Proof: Consider a preorder walk around the tree T_{opt} beginning with an arbitrary terminal as the root. Such a walk will traverse every edge of T_{opt} exactly twice. Introduce a new edge e''_j with the same cost

$c''(e''_j)$ as the corresponding path in the walk every time a new terminal is encountered and when the walk terminates at the root. Let $e''_1, \ldots, e''_r$ be the so constructed edges in nondecreasing cost order. The edges $e''_1, \ldots, e''_{r-1}$ build a tree T'' with the cost $c''(T'') = c''(e''_1) + \cdots + c''(e''_{r-1})$ which spans all terminals. Obviously, T'' is no cheaper than T'; and its longest edge e''_{r-1} is not cheaper than e'_{r-1} (otherwise, replacing e'_{r-1} by a suitable edge of T'' would create a tree spanning all terminals and cheaper than T'). So we have: $2c(T_{opt}) = c''(T'') + c''(e''_r) \geq c''(T'') + c'(e'_{r-1}) - c''(e''_{r-1}) + c''(e''_r) \geq c''(T'') + c'(e'_{r-1}) - c''(e''_r) + c''(e''_r) = c''(T'') + c'(e'_{r-1}) \geq c'(T') + c'(e'_{r-1}) = 2L$.
On the other hand, we have: $2L = c'(T') + c'(e'_{r-1}) \geq c'(T') + \frac{1}{r-1}c'(T') = (1 + \frac{1}{r-1})c'(T')$, so $c(T_{opt}) \leq c'(T') \leq (2 - \frac{2}{r})L$. □

DUAL-ASCENT cannot always reach $v(LP_C)$ (section 3)

The discussion in section 3 also shows the importance of the strategy for choosing active components. A question arises naturally: Is there another (efficient) strategy which always leads to better lower bounds or even $v(LP_C)$ itself? Note that the latter would not be surprising from a complexity theory point of view, because $v(LP_C)$ can be calculated in polynomial time. This problem is particularly relevant if an instance is to be solved to optimality: although the lower bounds generated by DA_C usually come close to $v(LP_C)$, the remaining gap is sometimes larger than desired in the context of exact algorithms. In [19], we used methods like Lagrangean relaxation and row generation to further improve the bounds provided by dual-ascent. But these methods tend to be slow for very large instances. Therefore, if such instances are to be solved quickly, a modification of the described dual-ascent algorithm would be the only known alternative. A principal difficulty here is that LP_C might have only fractional optimal solutions and so $v(LP_C)$ might be fractional even for integer edge costs; something which is never the case for the solutions provided by DA_C. But this could be remedied by allowing dual variables to be increased by a suitable fraction of (reduced) edge costs (see also section 4). So the question here is: Is there a variant of DA_C, probably involving a more sophisticated strategy for choosing active components and allowing arbitrary increment of dual variables, which always reaches the value $v(LP_C)$? We answer this question in the negative by presenting an instance for which no order of choosing the components can lead to the "correct" cuts.

Example 2 In the figure 2, the top terminal is considered as the root; and all edges have costs 1. It is easy to see that for this instance, $v(P_C) = v(LP_C) = 6$ (the relevant cuts are sketched using dashed lines). Now consider the behaviour of DA_C: If the dual variables corresponding to the terminals in the middle are not raised over 1, the dual variables of the components corresponding to the terminal at the bottom will sum to 3, and the generated lower bound will be 5. Now assume that the dual variables corresponding to a terminal in the middle sum to $1+\alpha$, $0 < \alpha \leq 1$. In the network with the resulting reduced costs, it is easy to find a Steiner tree with the cost $5-2\alpha$, so the generated lower bound would be at most $(1+\alpha)+(5-2\alpha) = 6-\alpha$. Again, the argumentation can be made independent of the choice of the root by letting several copies of this graph share the top terminal.

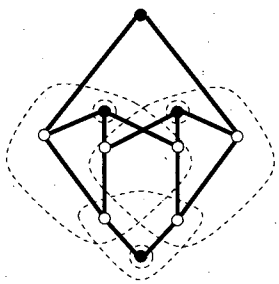

Figure 2: No choice in DA_C leads to correct cuts

Direct modification of PD_{UC} for directed relaxation (section 4)

An idea which is promising at the first sight is the direct application of the primal-dual method of PD_{UC} (simultaneous increasing of all dual variables corresponding to active components and merging components which share a vertex) to the directed cut relaxation. The obtained lower bound (using an additional trick to make it independent of the choice of the root) is the weight of a minimum spanning tree in a graph $\hat{G}$ similar to the graph G' defined in section 1, but with the cost of each edge (z_i, z_j) defined as $\hat{c}(z_i, z_j) := \min\{\min\{d_G(z_i, v_i), d_G(z_j, v_j)\} + c(v_i, v_j) \mid v_i \in N(z_i), v_j \in N(z_j)\}$. This lower bound $\hat{l}$ was already used by us in [19] with a completely different derivation. It holds that $\hat{l} \geq \frac{1}{2}c'(T')$, where T' is a minimum spanning tree for G': Let $z_i - \hat{v}_i - \hat{v}_j$ be the path in G corresponding to the edge (z_i, z_j) in $\hat{G}$. Since $\hat{v}_j \in N(z_j)$, we have $d_G(z_j, \hat{v}_j) \leq d_G(z_i, \hat{v}_i) + c(\hat{v}_i, \hat{v}_j)$, so $\hat{c}(z_i, z_j) \geq \frac{1}{2}(d_G(z_i, \hat{v}_i) + c(\hat{v}_i, \hat{v}_j) + d_G(z_j, \hat{v}_j)) \geq \frac{1}{2}c'(z_i, z_j)$, and the claimed relation between the weights of the corresponding minimum spanning trees follows straightforwardly. So, we have again a ratio of at most 2 between the upper and the lower bound, and we can calculate both bounds in $O(e + n \log n)$ time, as it was the case for PD_{UC}; although the two methods for computing lower bounds are incomparable (neither method always generate tighter bounds

than the other). But the main drawback remains: Empirically, the generated lower bounds are again not nearly as tight as those provided by DA_C.

Efficient implementation of PRUNE (section 4)

The following lemma enables us to perform the reverse order deletion in PRUNE efficiently.

Lemma 4 Consider a graph $\tilde{H}$ with the edge set H in which the weight of each edge $\tilde{e}$ is the position $p(\tilde{a})$ of the corresponding arc $\tilde{a}$ in the list $\vec{H}$. Let T' be the (edge set of a) tree generated by computing a minimum spanning tree for $\tilde{H}$ and pruning it until it has only terminals as leaves. Then we have: $T' = H'$.

Proof: First notice that there is a unique minimum spanning tree in $\tilde{H}$, since no two edge weights are equal. Now consider the computation of the minimum spanning tree $\tilde{T}$ for $\tilde{H}$ with the algorithm of Kruskal. Let e^* be an edge in H'. By the construction of H', we know that the endpoints of e^* are not connected by edges in $E_1 := \{\tilde{e} \in H' \mid p(\tilde{e}) > p(e^*)\} \cup \{\tilde{e} \in H \mid p(\tilde{e}) < p(e^*)\}$, because adding e^* to E_1 changes the connectivity relation for at least one pair of terminals. Consequently, the endpoints are not connected by edges in $\{\tilde{e} \in \tilde{T} \mid p(\tilde{e}) < p(e^*)\}$, which is a subset of E_1. Thus, e^* is included in $\tilde{T}$ by the algorithm of Kruskal.
Now we have $H' \subseteq \tilde{T}$. No edge $\tilde{e}$ in $\tilde{T} \setminus H'$ can be on a path between two terminals in $\tilde{T}$, because then there would be another path between these two terminals in H' (and hence in $\tilde{T}$) which does not use $\tilde{e}$ and $\tilde{T}$ would contain a cycle. So, no edge in $\tilde{T} \setminus H'$ will be present in T', meaning that $T' \subseteq H'$.
Now observe that T' is a tree which contains all terminals. No edges can be added to such a tree without creating cycles or non-terminals of degree 1, both features which are not present in H' by its construction. So we have $H' = T'$. □

Using lemma 4, PRUNE can be implemented by (mainly) computing a minimum spanning tree in $\tilde{H}$. Since the edges of $\tilde{H}$ are already available in a sorted list, this minimum spanning tree can be computed even in $O(e\ \alpha(e, n))$ time.

Arithmetic errors in PD_C (section 4)

The discussion of the algorithm PD_C in section 4 assumes exact real arithmetic. Of course, actual computers cannot handle infinite precision arithmetic; and simply replacing real numbers with floating-point numbers is not appropriate due to the unpredictable perturbations caused by the roundoff errors. But even if we adopt the (usual) assumption that all numbers in the input are integers, using exact arithmetic could deteriorate the worst case running time due to the growing denominators. But if we allow a deterioration of ϵ (for a small constant ϵ) in the approximation ratio, we can overcome this difficulty as follows.
We rescale the *reducedCost*-values using $1/step_r$ units and the d-values using $1/step_d$ units, where $step_r$ and $step_d$ are integers described below. In each recalculation of a d-value (see page 5), we round up the result of the division in the first assignment and round down in the second assignment; this can be done by appropriately using the integer DIV operation. The only inaccuracy introduced this way is that the reduced costs of some arcs in $\vec{H}$ can be slightly larger than zero. For each arc x in $\vec{H}$, this error can be bounded by splitting it up into the error made by rounding down in the final d-value calculation (at most $r \cdot step_r/step_d^2$) and the sum of the errors made by rounding up in all updates of *reducedCost*(x). The effect of the latter errors can be kept small by choosing $step_r \gg step_d$, because then the error made in each updating of *reducedCost*(x) is relatively small (at most $r/step_r$) when compared to the change in *reducedCost*(x) (at least $1/(r \cdot step_d)$), meaning that the total relative error made this way is small (at most $(r/step_r)/(1/(r \cdot step_d)) = r^2 \cdot step_d/step_r$ over all updates of *reducedCost*(x)). Finally, since there are at most n arcs in $\vec{H}'$, a maximum error of ϵ in the approximation ratio can be guaranteed with polynomially large factors $step_r$ and $step_d$ (e.g. $(4n/\epsilon)^3 r^5$ and $(4n/\epsilon)^2 r^3$, respectively), meaning that all numbers involved in the computation are (up to a constant factor) of the same size as the input length.

Approximation ratios for PD_C are tight (section 4)

The following two examples show that the proven approximation ratios for upper and lower bounds are both tight.

Example 3 In the figure 3, the top-left terminal is considered as the root; and all edges have costs 1. Note that this graph is even bipartite. It is easy to see that for this instance, $v(P_C) = v(LP_C) = 2l + k$. But PD_C will deliver the lower bound $l + k + \frac{l+k}{k+1}$. Choosing $l = k^2 >> 1$ we will get a gap of approximately 2.
Again, the argumentation can be made independent of the choice of the root by letting l copies of this graph share the top-left terminal.

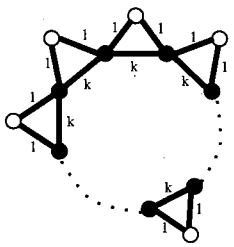

Figure 4:
PD_C : $upper \approx 2v(P_C) = 2v(LP_C)$

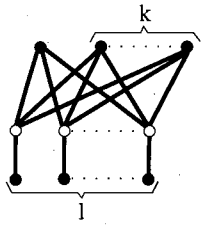

Figure 3:
PD_C : $v(P_C) = v(LP_C) \approx 2 \cdot lower$

Example 4 In the figure 4, setting $k = 1 + \epsilon$ for a small ϵ will ensure that for all terminals not adjacent to the (arbitrary) root, the incident arcs with costs 1 will be inserted into $\vec{H}$ before those with cost k, meaning that PRUNE will remove the latter. So we will have $upper = 2(r-2) + (1+\epsilon)$, whereas $v(P_C) = v(LP_C) = (r-1)(1+\epsilon)$. By choosing a large r we will get a gap of approximately 2.