Obtaining Optimal k-Cardinality Trees Fast

Markus Chimani*   Maria Kandyba*†   Ivana Ljubić‡§   Petra Mutzel*

Abstract

Given an undirected graph $G = (V, E)$ with edge weights and a positive integer number k, the k-Cardinality Tree problem consists of finding a subtree T of G with exactly k edges and the minimum possible weight. Many algorithms have been proposed to solve this NP-hard problem, resulting in mainly heuristic and metaheuristic approaches.

In this paper we present an exact ILP-based algorithm using directed cuts. We mathematically compare the strength of our formulation to the previously known ILP formulations of this problem, and give an extensive study on the algorithm's practical performance compared to the state-of-the-art metaheuristics.

In contrast to the widespread assumption that such a problem cannot be efficiently tackled by exact algorithms for medium and large graphs (between 200 and 5000 nodes), our results show that our algorithm not only has the advantage of proving the optimality of the computed solution, but also often outperforms the metaheuristic approaches in terms of running time.

1 Introduction

We consider the *k-Cardinality Tree problem* (KCT): given an undirected graph $G = (V, E)$, an edge weight function $w : E \to \mathbb{R}$, and a positive integer number k, find a subgraph T of G which is a minimum weight tree with exactly k edges. This problem has been extensively studied in literature as it has various applications, e.g., in oil-field leasing, facility layout, open pit mining, matrix decomposition, quorum-cast routing, telecommunications, etc [9]. A large amount of research was devoted to the development of heuristic [5, 14] and, in particular, metaheuristic methods [4, 8, 11, 7, 25]. An often used argument for heuristic approaches is that exact methods for this NP-hard problem would require too much computation time and could only be applied to very small graphs [9, 10].

The problem also received a lot of attention in the approximation algorithm community [1, 3, 17, 18]: a central idea thereby is the primal-dual scheme, based on integer linear programs (ILPs), which was proposed by Goemans and Williamson [19] for the prize-collecting Steiner tree problem. An exact approach was presented by Fischetti et al. [15], by formulating an ILP based on general subtour elimination constraints (GSEC). This formulation was implemented by Ehrgott and Freitag [13] using a Branch-and-Cut approach. The resulting algorithm was only able to solve graphs with up to 30 nodes, which may be mainly due to the comparably weak computers in 1996.

In this paper we show that the traditional argument for metaheuristics over exact algorithms is deceptive on this and related problems. We propose a novel exact ILP-based algorithm which can indeed be used to solve all known benchmark instances of KCTLIB [6]—containing graphs of up to 5000 nodes—to provable optimality. Furthermore, our algorithm often, in particular on mostly all graphs with up to 1000 nodes, is faster than the state-of-the-art metaheuristic approaches, which can neither guarantee nor assess the quality of their solution.

To achieve these results, we present Branch-and-Cut algorithms for KCT and NKCT—the node-weighted variant of KCT. Therefore, we transform both KCT and NKCT into a similar directed and rooted problem called *k-Cardinality Arborescence problem* (KCA), and formulate an ILP for the latter, see Section 2. In the section thereafter, we provide polyhedral and algorithmic comparison to the known GSEC formulation. In Section 4, we describe the resulting Branch-and-Cut algorithm in order to deal with the exponential ILP size. We conclude the paper with the extensive experimental study in Section 5, where we compare our algorithm with the state-of-the-art metaheuristics for the KCT.

2 Directed Cut Approach

2.1 Transformation into the k-Cardinality Arborescence Problem. Let $D = (V_D, A_D)$ be a directed graph with a distinguished root vertex $r \in V_D$ and arc costs c_a for all arcs $a \in A_D$. The k-Cardinality Arborescence problem (KCA) consists of finding a weight minimum rooted tree T_D with k arcs

which is directed from the root outwards. More formally, T_D has to satisfy the following properties:

(P1) T_D contains exactly k arcs,

(P2) for all $v \in V(T_D) \setminus \{r\}$, there exists a directed path $r \to v$ in T_D, and

(P3) for all $v \in V(T_D) \setminus \{r\}$, v has in-degree 1 in T_D.

We transform any given KCT instance ($G = (V, E), w, k$) into a corresponding KCA instance ($G_r, r, c, k+1$) as follows: we replace each edge $\{i, j\}$ of G by two arcs (i, j) and (j, i), introduce an artificial root vertex r and connect r to every node in V. Hence we obtain a digraph $G_r = (V \cup \{r\}, A \cup A_r)$ with $A = \{(i, j), (j, i) \mid \{i, j\} \in E\}$ and $A_r = \{(r, j) \mid j \in V\}$. For each arc $a = (i, j)$ we define the cost function $c(a) := 0$ if $i = r$, and $c(a) := w(\{i, j\})$ otherwise.

To be able to interpret each feasible solution T_{G_r} of this resulting KCA instance as a solution of the original KCT instance, we impose an additional constraint

(P4) T_{G_r} contains only a single arc of A_r.

If this property is satisfied, it is easy to see that a feasible KCT solution with the same objective value can be obtained by removing r from T_{G_r} and interpreting the directed arcs as undirected edges.

2.2 The Node-weighted k-Cardinality Tree Problem. The Node-weighted k-Cardinality Tree problem (NKCT) is defined analogously to KCT but its weight function $w' : V \to \mathbb{R}$ uses the nodes as its basic set, instead of the edges (see, e.g., [10] for the list of references). We can also consider the general All-weighted k-Cardinality Tree problem (AKCT), where a weight-function w for the edges, and a weight-function w' for the nodes are given.

We can transform any NKCT and AKCT instance into a corresponding KCA instance using the ideas of [24]: the solution of KCA is a rooted, directed tree where each vertex (except for the unweighted root) has in-degree 1. Thereby, a one-to-one relationship between each selected arc and its target node allows us to precompute the node-weights into the arc-weights of KCA: for all $(i, j) \in A \cup A_r$ we have $c((i, j)) := w'(j)$ for NKCT, and $c((i, j)) := w(\{i, j\}) + w'(j)$ for AKCT.

2.3 ILP for the KCA. In the following let the graphs be defined as described in Section 2.1. To model KCA as an ILP, we introduce two sets of binary variables:

$$x_a, y_v \in \{0, 1\} \quad \forall a \in A \cup A_r, \forall v \in V$$

Thereby, the variables are 1, if the corresponding vertex or arc is in the solution and 0 otherwise.

Let $S \subseteq V$. The sets $E(S)$ and $A(S)$ are the edges and arcs of the subgraphs of G and G_r, respectively, induced by S. Furthermore, we denote by $\delta^+(S) = \{(i, j) \in A \cup A_r \mid i \in S, j \in V \setminus S\}$ and $\delta^-(S) = \{(i, j) \in A \cup A_r \mid i \in V \setminus S, j \in S\}$ the outgoing and ingoing edges of a set S, respectively. We can give the following ILP formulation, using $x(B) := \sum_{b \in B} x_b$, with $B \subseteq A$, as a shorthand:

$$(2.1) \qquad \text{DCUT}: \qquad \min \sum_{a \in A} c(a) \cdot x_a$$

$$(2.2) \qquad x(\delta^-(S)) \geq y_v \qquad \forall S \subseteq V \setminus \{r\}, \forall v \in S$$

$$(2.3) \qquad x(\delta^-(v)) = y_v \qquad \forall v \in V$$

$$(2.4) \qquad x(A) = k$$

$$(2.5) \qquad x(\delta^+(r)) = 1$$

$$(2.6) \qquad x_a, y_v \in \{0, 1\} \qquad \forall a \in A \cup A_r, \forall v \in V$$

The *dcut-constraints* (2.2) ensure property (P2) via directed cuts, while property (P3) is ensured by the in-degree constraints (2.3). Constraint (2.4) ensures the k-cardinality requirement (P1) and property (P4) is modeled by (2.5).

LEMMA 2.1. *By replacing all in-degree constraints (2.3) by a single node-cardinality constraint*

$$(2.7) \qquad y(V) = k + 1,$$

we obtain an equivalent ILP and an equivalent LP-relaxation.

Proof. The node-cardinality constraint can be generated directly from (2.3) and (2.4), (2.5). Vice versa, we can generate (2.3) from (2.7), using the dcut-constraints (2.2). □

Although the formulation using (2.7) requires less constraints, the ILP using in-degree constraints has certain advantages in practice, see Section 4.

3 Polyhedral Comparison

In [15], Fischetti et al. give an ILP formulation for the undirected KCT problem based on general subtour elimination constraints (GSEC). We reformulate this approach and show that both GSEC and DCUT are equivalent from the polyhedral point of view.

In order to distinguish between undirected edges and directed arcs we introduce the binary variables $z_e \in \{0, 1\}$ for every edge $e \in E$, which are 1 if $e \in T$ and 0 otherwise. For representing the selection of the

nodes we use the y-variables as in the previous section. The constraints (3.9) are called the *gsec-constraints*.

$$(3.8) \qquad \text{GSEC}: \qquad \min \sum_{e \in E} c(e) \cdot z_e$$

$$(3.9) \quad z(E(S)) \leq y(S \setminus \{t\}) \quad \forall S \subseteq V, |S| \geq 2, \forall t \in S$$
$$(3.10) \qquad z(E) = k$$
$$(3.11) \qquad y(V) = k + 1$$
$$(3.12) \qquad z_e, y_v \in \{0, 1\} \qquad \forall e \in E, \forall v \in V$$

Let $\mathcal{P}_D$ and $\mathcal{P}_G$ be the polyhedra corresponding to the DCUT and GSEC LP-relaxations, respectively. I.e.,

$$\mathcal{P}_D := \{ \quad (x, y) \in \mathbb{R}^{|A \cup A_r| + |V|} \mid 0 \leq x_e, y_v \leq 1$$
$$\text{and } (x, y) \text{ satisfies } (2.2)–(2.5) \quad \}$$

$$\mathcal{P}_G := \{ \quad (z, y) \in \mathbb{R}^{|E| + |V|} \mid 0 \leq z_e, y_v \leq 1$$
$$\text{and } (z, y) \text{ satisfies } (3.9)–(3.11) \quad \}$$

THEOREM 3.1. *The* GSEC *and the* DCUT *formulations have equally strong LP-relaxations, i.e.,*

$$\mathcal{P}_G = \text{proj}_z(\mathcal{P}_D),$$

whereby $\text{proj}_z(\mathcal{P}_D)$ *is the projection of* $\mathcal{P}_D$ *onto the* (z, y) *variable space with* $z_{\{i,j\}} = x_{(i,j)} + x_{(j,i)}$ *for all* $\{i, j\} \in E$.

Proof. We prove equality by showing mutual inclusion:

• $\text{proj}_z(\mathcal{P}_D) \subseteq \mathcal{P}_G$: Any $(\bar{z}, \bar{y}) \in \text{proj}_z(\mathcal{P}_D)$ satisfies (3.10) by definition, and (3.11) by (2.3) and Lemma 2.1. Let $\bar{x}$ be the vector from which we projected the vector $\bar{z}$, and consider some $S \subseteq V$ with $|S| \geq 2$ and some vertex $t \in S$. We show that $(\bar{z}, \bar{y})$ also satisfies the corresponding gsec-constraint (3.9):

$$\bar{z}(E(S)) = \bar{x}(A(S)) = \sum_{v \in S} \bar{x}(\delta^-(v)) - \bar{x}(\delta^-(S))$$
$$\stackrel{(2.3)}{=} \bar{y}(S) - \bar{x}(\delta^-(S)) \stackrel{(2.2)}{\leq} \bar{y}(S) - \bar{y}_t.$$

• $\mathcal{P}_G \subseteq \text{proj}_z(\mathcal{P}_D)$: Consider any $(\bar{z}, \bar{y}) \in \mathcal{P}_G$ and a set

$$X := \{ \quad x \in \mathbb{R}_{\geq 0}^{|A \cup A_r|} \mid x \text{ satisfies } (2.5)$$
$$\text{and } x_{ij} + x_{ji} = \bar{z}_{\{ij\}} \; \forall (i, j) \in A \quad \}.$$

Every such projective vector $\bar{x} \in X$ clearly satisfies (2.4). In order to generate the dcut-inequalities (2.2) for the corresponding $(\bar{x}, \bar{y})$, it is sufficient to

show that we can always find an $\hat{x} \in X$, which together with $\bar{y}$ satisfies the indegree-constraints (2.3). Since then, for any $S \subseteq V$ and $t \in S$:

$$\hat{x}(\delta^-(S)) = \sum_{v \in S} \hat{x}(\delta^-(v)) - \hat{x}(A(S))$$
$$\stackrel{(2.3)}{=} \bar{y}(S) - \bar{z}(E(S)) \stackrel{(3.9)}{\geq} \bar{y}_t.$$

We show the existence of such an $\hat{x}$ using a proof technique similar to [20, proof of Claim 2], where it was used for the Steiner tree problem.

An $\hat{x} \in X$ satisfying (2.3) can be interpreted as the set of feasible flows in a bipartite transportation network (N, L), with $N := (E \cup \{r\}) \cup V$. For each undirected edge $e = (u, w) \in E$ in G, our network contains exactly two outgoing arcs $(e, u), (e, w) \in L$. Furthermore, L contains all arcs of A_r. For all nodes $e \in E$ in N we define a supply $s(e) := \bar{z}_e$; for the root r we set $s(r) := 1$. For all nodes $v \in V$ in N we define a demand $d(v) := \bar{y}_v$.

Finding a feasible flow for this network can be viewed as a capacitated transportation problem on a complete bipartite network with capacities either zero (if the corresponding edge does not exist in L) or infinity. Note that in our network the sum of all supplies is equal to the sum of all demands, due to (3.10) and (3.11). Hence, each feasible flow in such a network will lead to a feasible $\hat{x} \in X$. Such a flow exists if and only if for every set $M \subseteq N$ with $\delta^+_{(N,L)}(M) = \emptyset$ the condition

$$(3.13) \qquad s(M) \leq d(M)$$

is satisfied, whereby $s(M)$ and $d(M)$ are the total supply and the total demand in M, respectively, cf. [16, 20]. In order to show that this condition holds for (N, L), we distinguish between two cases; let $U := E \cap M$:

$r \in M$: Since r has an outgoing arc for every $v \in V$ and $\delta^+_{(N,L)}(M) = \emptyset$, we have $V \subset M$. Condition (3.13) is satisfied, since $s(r) = 1$ and therefore:

$$s(M) = s(r) + \bar{z}(U) \leq s(r) + \bar{z}(E)$$
$$= \bar{z}(E) + 1 \stackrel{(3.10),(3.11)}{=} \bar{y}(V) = d(M).$$

$r \notin M$: Let $S := V \cap M$. We then have $U \subseteq E(S)$. If $|S| \leq 1$ we have $U = \emptyset$ and therefore (3.13) is automatically satisfied. For $|S| \geq 2$, the condition is also satisfied, since for every $t \in S$ we have:

$$s(M) = \bar{z}(U) \leq \bar{z}(E(S)) \stackrel{(3.9)}{\leq} \bar{y}(S) - \bar{y}_t$$
$$\leq \bar{y}(S) = d(M). \qquad \square$$

3.1 Other approaches.

3.1.1 Multi-Commodity Flow. One can formulate a multi-commodity-flow based ILP for KCA (MCF) as it was done for the prize-collecting Steiner tree problem (PCST) [22], and augment it with cardinality inequalities. Analogously to the proof in [22], which shows the equivalence of DCUT and MCF for PCST, we can obtain:

LEMMA 3.1. *The LP-relaxation of* MCF *for KCA is equivalent to* GSEC *and* DCUT.

Nonetheless, we know from similar problems [12, 23] that directed-cut based approaches are usually more efficient than multi-commodity flows in practice.

3.1.2 Undirected Cuts for Approximation Algorithms. In [17], Garg presents an approximation algorithm for KCT, using an ILP for lower bounds (GU-CUT). It is based on undirected cuts and has to be solved $|V|$ times, once for all possible choices of a root node r.

LEMMA 3.2. DCUT *is stronger than* GUCUT.

Proof. Clearly, each feasible point in $\mathcal{P}_D$ is feasible in the LP-relaxation of GUCUT using the projection proj_z. On the other hand, using a traditional argument, assume a complete graph on 3 nodes is given, where each vertex variable is set to 1, and each edge variable is set to 0.5. This solution is feasible for the LP-relaxation of GUCUT, but infeasible for DCUT. □

4 Branch-and-Cut Algorithm

Based on our DCUT formulation, we developed and implemented a Branch-and-Cut algorithm. For a general description of the Branch-and-Cut scheme see, e.g., [27]: Such algorithms start with solving an *LP relaxation*, i.e., the ILP without the integrality properties, only considering a certain subset of all constraints. Given the *fractional solution* of this partial LP, we perform a *separation routine*, i.e., identify constraints of the full constraint set which the current solution violates. We then add these constraints to our current LP and reiterate these steps. If at some point we cannot find any violated constraints, we have to resort to *branching*, i.e., we generate two disjoint subproblems, e.g., by fixing a variable to 0 or 1. By using the LP relaxation as a lower bound, and some heuristic solution as an upper bound, we can prune irrelevant subproblems.

In [13], a Branch-and-Cut algorithm based on the GSEC formulation has been developed. Note that the dcut-constraints are sparser than the gsec-constraints, which in general often leads to a faster optimization

in practice. This conjecture was experimentally confirmed, e.g., for the similar prize-collecting Steiner tree problem [23], where a directed-cut based formulation was compared to a GSEC formulation. The former was both faster in overall running time and required less iterations, by an order of 1–2 magnitudes. Hence we can expect our DCUT approach to have advantages over GSEC in practice. In Section 4.2 we will discuss the formal differences in the performances between the DCUT and the GSEC separation algorithms.

4.1 Initialization. Our algorithm starts with the constraints (2.3), (2.4), and (2.5). We prefer the indegree constrains (2.3) over the node-cardinality constraint (2.7), as they strengthen the initial LP and we do not require to separate dcut-constraints with $|S| = 1$ later.

For the same reason, we add the orientation-constraints

$$(4.14) \qquad x_{ij} + x_{ji} \leq y_i \quad \forall i \in V, \forall \{i, j\} \in E$$

to our initial ILP. Intuitively, these constraints ensure a unique orientation for each edge, and require for each selected arc that both incident nodes are selected as well. These constraints do not actually strengthen the DCUT formulation as they represent the gsec-constraints for all two-element sets $S = \{i, j\} \subset V$. From the proof of Theorem 3.1, we know that these inequalities can be generated with the help of (2.3) and (2.2). Nonetheless, as experimentally shown in [22] for PCST and is also confirmed by our own experiments, the addition of (4.14) speeds up the algorithm tremendously, as they do not have to be separated explicitly by the Branch-and-Cut algorithm.

We also tried *asymmetry constraints* [22] to reduce the search space by excluding symmetric solutions:

$$(4.15) \qquad x_{rj} \leq 1 - y_i \quad \forall i, j \in V, i < j.$$

They assure that for each KCA solution, the vertex adjacent to the root is the one with the smallest possible index. Anyhow, we will see in our experiments that the quadratic number of these constraints becomes a hindrance for large graphs and/or small k in practice.

4.2 Separation. The dcut-constraints (2.2) can be separated in polynomial time via the traditional maximum-flow separation scheme: we compute the maximum-flow from r to each $v \in V$ using the edge values of the current solution as capacities. If the flow is less than y_v, we extract one or more of the induced minimum (r, v)-cuts and add the corresponding constraints to our model. In order to obtain more cuts

with a single separation step we also use nested- and back-cuts [21, 23]. Indeed, using these additional cuts significantly speeds up the computation.

Recall that in a general separation procedure we search for the most violated inequality of the current LP-relaxation. In order to find the most violated inequality of the DCut formulation, or to show that no such exists, we construct the flow network only once and perform at most $|V|$ maximum-flow calculations on it. This is a main reason why the DCut formulation performs better than Gsec in practice: a single separation step for Gsec requires $2|V| - 2$ maximum-flow calculations, as already shown by Fischetti et al. [15]. Furthermore, the corresponding flow network is not static over all those calculations, but has to be adapted prior to each call of the maximum-flow algorithm.

Our test sets, as described in Section 5, also contain grid graphs. In such graphs, it is easy to detect and enumerate all 4-cycles by embedding the grids into the plane and traversing all faces except for the single large one. Note that due to our transformation, all 4-cycles are bidirected. Let $\mathcal{C}_4$ be the set of all bidirected 4-cycles; a cycle $C \in \mathcal{C}_4$ then consists of 8 arcs and $V[C]$ gives the vertices on C. We use a separation routine for gsec-constraints on these cycles:

$$(4.16) \quad \sum_{a \in C} x_a \leq \sum_{i \in V[C] \setminus \{v\}} y_i \quad \forall C \in \mathcal{C}_4, \forall v \in V[C].$$

4.3 Upper Bounds and Proving Optimality. In the last decade, several heuristics and metaheuristics have been developed for KCT. See, e.g., [4, 5, 8, 11] for an extensive comparison. Traditional Branch-and-Cut algorithms allow to use such algorithms as primal heuristics, giving upper bounds which the Branch-and-Cut algorithm can use for bounding purposes when branching. The use of such heuristics is two-fold: (a) they can be used as start-heuristics, giving a good initial upper bound before starting the actual Branch-and-Cut algorithm, and (b) they can be run multiple times during the exact algorithm, using the current fractional solutions as an additional input, or *hint*, in order to generate new and tighter upper bounds on the fly.

Let h be a primal bound obtained by such a heuristic. Mathematically, we can add this bound to our LP as

$$\sum_{a \in A} c(a) \cdot x_a \leq h - \Delta.$$

Thereby, $\Delta := \min\{c(a) - c(b) \mid c(a) > c(b), \ a, b \in A\}$ denotes the minimal difference between any two cost values. If the resulting ILP is found to be infeasible, we have a proof that h was optimal, i.e., the heuristic solution was optimal.

As our experiments reveal, our algorithm is already very successful without the use of any such heuristic. Hence we compared our heuristic-less Branch-and-Cut algorithm (DC^-) with one using a *perfect heuristic*: a (hypothetical) algorithm that requires no running time and gives the optimal solution. We can simulate such a perfect heuristic by using the optimal solution obtained by a prior run of DC^-. We can then measure how long the algorithm takes to discover the infeasibility of the ILP. We call this algorithm variant DC^+. If the runtime performance of DC^- and DC^+ are similar, we can conclude that using any heuristic for bounding is not necessary.

5 Experimental results

We implemented our algorithm in C++ using CPLEX 9.0 and LEDA 5.0.1. The experiments were performed on 2.4 GHz AMD Opteron with 2GB RAM per process. We tested our algorithm on all instances of the KCTLIB [6] which consists of the following benchmark sets:

(BX) The set by Blesa and Xhafa [2] contains 35 4-regular graphs with 25–1000 nodes. The value of k is fixed to 20. The results of [8] have already shown that these instances are easy, which was confirmed by our experiments: our algorithm needed on average 1.47 seconds per instance to solve them to optimality, the median was 0.09 seconds.

(BB) The set by Blesa and Blum [8] is divided into four subsets of dense, sparse, grid and 4-regular graphs, respectively, with different sizes of up to 2500 nodes. Each instance has to be solved for different values of k, specified in the benchmark set: these are k_{rel} of $n = |V|$, for $k_{rel} = \{10\%, \ldots, 90\%\}$[1], and additionally $k = 2$ and $k = n - 2$. Note that the latter two settings are rather insignificant for our analysis, as they can be solved optimally via trivial algorithms in quadratic time.

The most successful known metaheuristics for (BB) are the hybrid evolutionary algorithm (HyEA) [4] and the ant colony optimization algorithm (ACO) [11].

(UBM) The set by Urošević et al. [25] consists of large 20-regular graphs with 500–5000 nodes which were originally generated randomly. The values for k are defined as for (BB) by using $k_{rel} = \{10\%, \ldots, 50\%\}$. In [25] a variable neighborhood decomposition search (VNDS) was presented, which is still the best known metaheuristic for this benchmark set.

---

[1]For the grid instances, the values k_{rel} differ slightly.

# of nodes	500	1000	1500	2000	3000	4000	5000
avg. time in sec.	7.5	48.3	107.4	310.7	1972	5549	15372.2
avg. gap of BKS	1.5%	0.1%	0.1%	0.2%	0.2%	0.3%	0.3%

Table 1: Average running times and average gap to the BKS provided in [5, 7, 25] for (UBM).
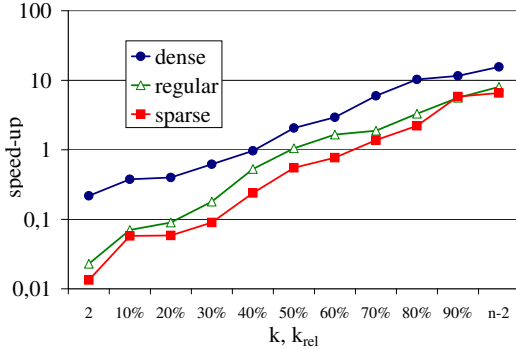

Figure 1: Speed-up factors for dense, regular and sparse graphs with $|V| < 2000$ obtained when asymmetry constraints (4.15) are included in the initial LP
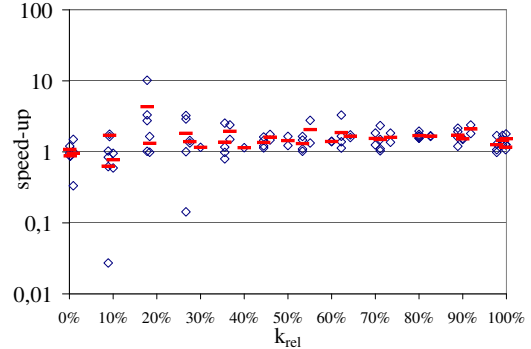

Figure 2: Speed-up factors for the grid instances of (BB) when gsec-constraints (4.16) are separated. For each instance and k_{rel} value there is a diamond-shaped datapoint; the short horizontal bars denote the average speed-up per k_{rel}.

Our computational experiments on (UBM) show that all instances with up to 3000 nodes can be solved to optimality within two hours. We are also able to solve the graphs with 4000 and 5000 nodes to optimality, although only about 50% of them in less than two hours. Note that for these large instances the VNDS metaheuristic of [25] is faster than our algorithm, however they thereby could not reach optimal solutions. Table 1 gives the average running times and the differences between the optimal solutions and the previously best known solutions (BKS).

In the following we will concentrate on the more common and diversified benchmark set (BB), and compare our results to those of HyEA and ACO. Unless specified otherwise, we always report on the DC$^-$ algorithm, i.e., the Branch-and-Cut algorithm without using any heuristic for upper bounds.

Algorithmic Behaviour. Figure 1 illustrates the effectiveness of the asymmetry constraints (4.15) depending on increasing relative cardinality k_{rel}. Therefore we measured the speed-up by the quotient $\frac{t_\emptyset}{t_{asy}}$, whereby t_{asy} and $t_\emptyset$ denote the running time with and without using (4.15), respectively. The constraints allow a speed-up by more than an order of magnitude for sparse, dense and regular graphs, but only for large cardinality $k > \frac{n}{2}$. Our experiments show that for smaller k, a variable x_{ri}, for some $i \in V$, is quickly set to 1 and stays at this value until the final result. In these cases the constraints cannot help and only slow down

the algorithm. Interestingly, the constraints were never profitable for the grid instances. For graphs with more than 2000 nodes using (4.15) is not possible due to memory restrictions, as the $\mathcal{O}(|V|^2)$ many asymmetry constraints are too much to handle. Hence, we ommitted these graphs in our figure.

We also report on the experiments with the special gsec-constraints (4.16) within the separation routine for the grid graphs. The clear advantage of these constraints is shown in Figure 2, which shows the obtained speed-up factor $\frac{t_\emptyset}{t_{gsec}}$ by the use of these constraints.

Based on these results we choose to include the assymmetry constraints for all non-grid instances with less then 2000 nodes and $k > \frac{n}{2}$, in all the remaining experiments. For the grid instances we always separate the gsec-constraints (4.16).

In Table 2, we show that the computation time is not only dependent on the graph size, but also on the density of the graph. Generally, we leave table cells empty if there is no problem instance with according properties.

As described in Section 4.3, we also investigate the influence of primal heuristics on our Branch-and-Cut algorithm. For the tested instances with 1000 nodes the comparison of the running times of DC$^+$ and DC$^-$ is shown in Figure 3. In general, our experiments show
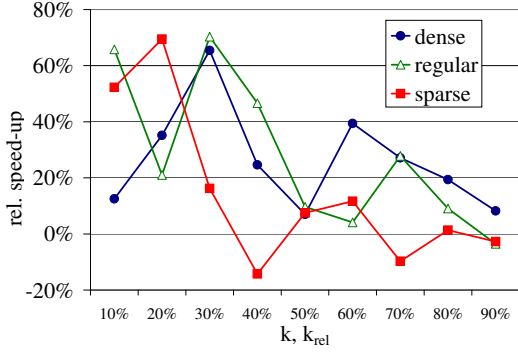
Figure 3: Relative speed-up $\frac{(t_{DC^-} - t_{DC^+})}{t_{DC^-}}$ (in percent) of DC^+ compared to DC^- for the instances with 1000 nodes.

Figure 4: Dependancy of the BKS quality on k_{rel}, for selected instances. The vertical axis gives the percentage of the tested instances for which the BKS provided in [6] are optimal.

avg. deg	set	500 nodes	1000 nodes
2.5	(BB)	1	8.1
4	(BB)	0.9	15.7
10	(BB)	2.6	25
20	(UBM)	7.5	48.4
36.3	(BB)	10.7	—

Table 2: Average CPU time (in seconds) over k_{rel} values of 10%, 20%, ..., 50%, sorted by the average degree of the graphs.

that DC^+ is only 10–30% percent faster than DC^- on average, even for the large graphs. Hence, we can conclude that a bounding heuristic is not crucial for the success of our algorithm.

Runtime Comparison. Table 3 summarizes the average and median computation times of our algorithm, sorted by size and categorized according to the special properties of the underlying graphs. We can observe that performance does not differ significantly between the sparse, regular and dense graphs, but that the grid instances are more difficult and require more computational power. This was also noticed in [9, 10, 14].

The behaviour of DC^- also has a clear dependency on k, see Figures 5(a), 5(c) and 5(d): for the sparse, dense and regular instances the running time increases with increasing k. In contrast to this, solving the grid instances (cf. Figure 5(b)) is more difficult for the relatively small k-values.

The original experiments for HyEA and ACO were performed on an Intel Pentium IV, 3.06 GHz with 1GB RAM and a Pentium IV 2.4 GHz with 512MB RAM, respectively. Using the well-known SPEC performance evaluation [26], we computed scaling factors of both machines to our computer: for the running time comparison we divided the times given in [4] and [11] by 1.5
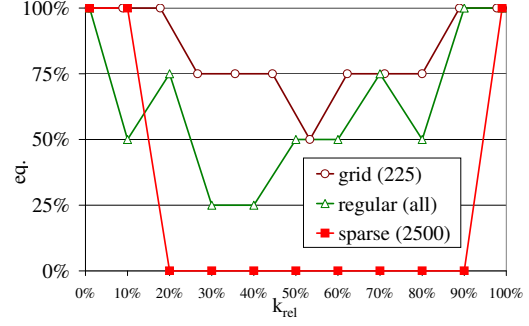
and 2, respectively. Anyhow, note that these factors are elaborate guesses and are only meant to help the reader to better evaluate the relative performance.

Table 3 additionally gives the average factor of $\frac{t_{HyEA}}{t_{DC^-}}$, i.e., the running time of our algorithm compared to (scaled) running time of HyEA. Analogously, Figure 5 shows the CPU time in (scaled) seconds of HyEA, ACO and our algorithm.

We observe that our DC^- algorithm performs better than the best metaheuristics in particular for the medium values of k, i.e., $40 - 70\%$ of $|V|$, on all instances with up to 1089 nodes, except for the very dense graph `le450_15a.g` with 450 nodes and 8168 edges, where HyEA was slightly faster. Interestingly, the gap between the heuristic and the optimal solution tended to be larger especially for medium values of k (cf. next paragraph and Figure 4 for details).

Solution Quality. For each instance of the sets (BX) and (BB) we compared the previously best known solutions, see [6], with the optimal solution obtained by our algorithm, in order to assess their quality. Most of the BKS were found by HyEA, followed by ACO. Note that these solutions where obtained by taking the best solutions over 20 independent runs per instance. In Table 4 we show the number of instances for which we proved that BKS was in fact not optimal, and give the corresponding average gap $gap_{bks} := \frac{BKS - OPT}{OPT}$ (in percent), where OPT denotes the optimal objective value obtained by DC^- and BKS denotes the best known solution obtained by either ACO or HyEA. Analogously, we give the average gaps $gap_{avg} := \frac{AVG - OPT}{OPT}$ (in percent), AVG denotes the average solution obtained by a metaheuristic. We observe that—concerning the solution quality—metaheuristics work quite well on instances with up to 1000 nodes and relatively small k.

| # nodes	500			1000–1089			2500		
group	**avg/med**	$\frac{t_{\text{HyEA}}}{t_{\text{DC}^-}}$		**avg/med**	$\frac{t_{\text{HyEA}}}{t_{\text{DC}^-}}$		**avg/med**	$\frac{t_{\text{HyEA}}}{t_{\text{DC}^-}}$	
sparse	1.7/2.0	2.2		15.2/20.2	2.6		923.2/391.5	0.1	
regular	1.7/1.5	3.1		22.2/21.4	5.7		—	—	
dense	7.5/7.9	2.2		25.5/27.9	2.7		—	—	
grid	11.7/1.2	0.1		124.8/98.7	1.1		3704.1/2800.1	0.1	

Table 3: Average/median CPU time (in seconds) and the average speed-up factor of DC^- to HyEA for the instance set (BB). Cells are left empty if there exists no instance matching the given criteria.
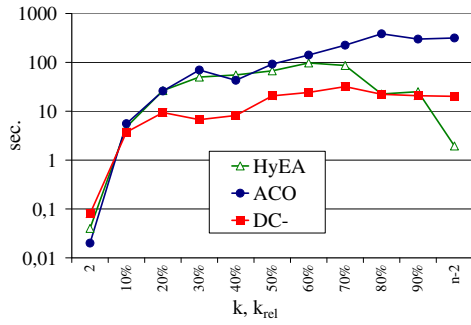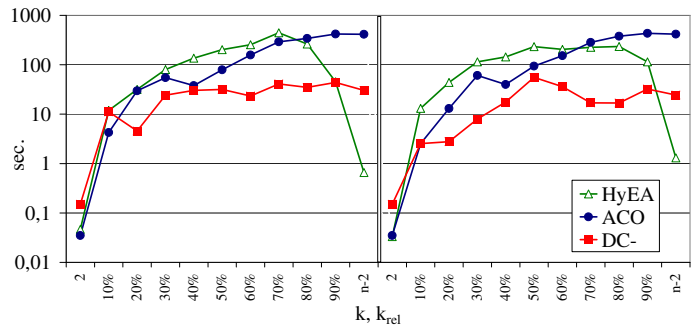

(a) sparse (1000 nodes, 1250 edges)
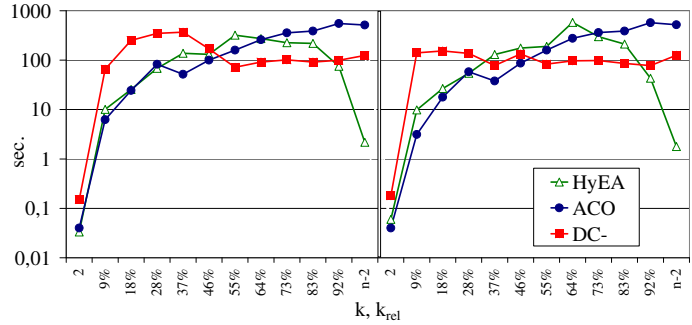
(b) grid 33x33 (1089 nodes, 2112 edges)
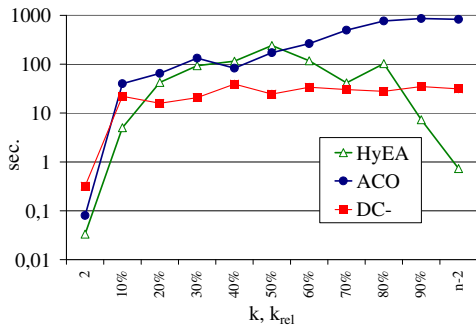
(c) dense (1000 nodes, 1250 edges)

(d) 4-regular (1000 nodes, 2000 edges)

Figure 5: Running times of DC^-, HyEA, and ACO (in seconds) for instances of (BB) with $\sim$1000 nodes, depending on k. The figures for the grid and regular instances show the times for two different instances of the same type, respectively.

instance	$(|V|,|E|)$	eq.	gap$_{bks}$	gap$_{avg}$ ACO	gap$_{avg}$ HyEA
regular g400-4-1.g	(400,800)	10/11	0.09	0.07	0.04
regular g400-4-5.g	(400,800)	8/11	0.19	0.31	0.35
regular g1000-4-1.g	(1000,2000)	7/11	0.07	0.65	0.12
regular g1000-4-5.g	(1000,2000)	3/11	0.08	0.45	0.35
sparse steinc5.g	(500,625)	11/11	–	0.97	0.06
sparse steind5.g	(1000,1250)	11/11	–	0.48	0.11
sparse steine5.g	(2500,3125)	3/11	0.13	n/a	0.23
dense le450a.g	(450,8168)	11/11	–	n/a	0.04
dense steinc15.g	(500,2500)	11/11	–	0.36	0.02
dense steind15.g	(1000,5000)	10/11	0.22	0.38	0.04
grid 15x15-1	(225,400)	13/13	–	1.27	0.18
grid 15x15-2	(225,400)	13/13	–	2.04	0.12
grid 45x5-1	(225,400)	4/13	0.54	n/a	1.22
grid 45x5-2	(225,400)	10/13	0.08	n/a	0.13
grid 33x33-1	(1089,2112)	3/12	0.31	1.70	0.57
grid 33x33-2	(1089,2112)	3/12	0.39	2.48	0.49
grid 50x50-1	(2500,4900)	2/11	0.95	n/a	1.27
grid 50x50-2	(2500,4900)	2/11	0.55	n/a	0.82

Table 4: Quality of previously best known solutions (BKS) provided in [6] for selected instances. "**eq.**" denotes the number of instances for which the BKS was optimal. For the other instances where BKS was not optimal, we give the average relative gap (**gap**$_{bks}$) between OPT and BKS. For all instances we also give the average relative gap (**gap**$_{avg}$) between the average solution of the metaheuristic and OPT. All gaps are given in percent. Cells marked as "n/a" cannot be computed as the necessary data for ACO is not not available.

In particular, for $k = 2$ and $k = n - 2$ they always found an optimal solution.

Acknowledgements. We would like to thank Christian Blum and Dragan Urošević for kindly providing us with test instances and sharing their experience on this topic.

References

[1] S. Arora and G. Karakostas. A $(2 + \epsilon)$-approximation algorithm for the k-MST problem. In *Proc. of the eleventh annual ACM-SIAM symposium on Discrete algorithms (SODA'00)*, pages 754–759, 2000.

[2] M. J. Blesa and F. Xhafa. A C++ Implementation of Tabu Search for k-Cardinality Tree Problem based on Generic Programming and Component Reuse. In c/o tranSIT GmbH, editor, *Net.ObjectDsays 2000 Tagungsband*, pages 648–652, Erfurt, Germany, 2000. Net.ObjectDays-Forum.

[3] A. Blum, R. Ravi, and S. Vempala. A constant-factor approximation algorithm for the k-MST problem. In *ACM Symposium on Theory of Computing*, pages 442–448, 1996.

[4] C. Blum. A new hybrid evolutionary algorithm for the huge k-cardinality tree problem. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'06)*, pages 515–522, New York, NY, USA, 2006. ACM Press.

[5] C. Blum. Revisiting dynamic programming for finding optimal subtrees in trees. *European Journal of Operational Research*, 177(1):102–115, 2007.

[6] C. Blum and M. Blesa. KCTLIB – a library for the edge-weighted k-cardinality tree problem. http://iridia.ulb.ac.be/ cblum/kctlib/.

[7] C. Blum and M. Blesa. Combining ant colony optimization with dynamic programming for solving the k-cardinality tree problem. In *Proc. of Computational Intelligence and Bioinspired Systems, 8th International Work-Conference on Artificial Neural Networks, (IWANN'05)*, volume 3512 of *Lecture Notes in Computer Science*, Barcelona, Spain, 2005. Springer.

[8] C. Blum and M. J. Blesa. New metaheuristic approaches for the edge-weighted k-cardinality tree problem. *Computers & OR*, 32:1355–1377, 2005.

[9] C. Blum and M. Ehrgott. Local search algorithms for the k-cardinality tree problem. *Discrete Applied Mathematics*, 128(2–3):511–540, 2003.

[10] J. Brimberg, D. Urošević, and N. Mladenović. Variable neighborhood search for the vertex weighted k-cardinality tree problem. *European Journal of Operational Research*, 171(1):74–84, 2006.

[11] T. N. Bui and G. Sundarraj. Ant system for the k-cardinality tree problem. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'04)*, volume 3102 of *Lecture Notes in Computer Science*, pages 36–47, Seattle, WA, USA, 2004. Springer.

[12] M. Chimani, M. Kandyba, and P. Mutzel. A new ILP formulation for 2-root-connected prize-collecting

Steiner networks. In *Proc. of 15th Annual European Symposium on Algorithms (ESA'07)*, volume 4698 of *Lecture Notes in Computer Science*, pages 681–692, Eilat, Israel, 2007. Springer.

[13] M. Ehrgott and J. Freitag. K_TREE/K_SUBGRAPH: a program package for minimal weighted *k*-cardinality tree subgraph problem. *European Journal of Operational Research*, 1(93):214–225, 1996.

[14] M. Ehrgott, J. Freitag, H.W. Hamacher, and F. Maffioli. Heuristics for the *k*-cardinality tree and subgraph problem. *Asia Pacific J. Oper. Res.*, 14(1):87–114, 1997.

[15] M. Fischetti, W. Hamacher, K. Jornsten, and F. Maffioli. Weighted *k*-cardinality trees: Complexity and polyhedral structure. *Networks*, 24:11–21, 1994.

[16] D. Gale. A theorem on flows in networks. *Pacific J. Math*, 7:1073–1082, 1957.

[17] N. Garg. A 3-approximation for the minimum tree spanning *k* vertices. In *Proc. of the 37th Annual Symposium on Foundations of Computer Science (FOCS'96)*, pages 302–309, Washington, DC, USA, 1996. IEEE Computer Society.

[18] N. Garg. Saving an epsilon: a 2-approximation for the *k*-MST problem in graphs. In *Proc. of the thirty-seventh annual ACM symposium on Theory of computing (STOC'05)*, pages 396–402. ACM Press, 2005.

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

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

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

[22] I. Ljubić. *Exact and Memetic Algorithms for Two Network Design Problems.* PhD thesis, Technische Universität Wien, 2004.

[23] I. Ljubić, R. Weiskircher, U. Pferschy, G. Klau, P. Mutzel, and M. Fischetti. An algorithmic framework for the exact solution of the prize-collecting Steiner tree problem. *Mathematical Programming, Series B*, 105(2–3):427–449, 2006.

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

[25] D. Urošević, J. Brimberg, and N. Mladenović. Variable neighborhood decomposition search for the edge weighted *k*-cardinality tree problem. *Computers & OR*, 31(8):1205–1213, 2004.

[26] Standard performance evaluation corporation. http://www.spec.org/.

[27] L. A. Wolsey. *Integer Programming.* Wiley-Interscience, 1998.