The Fractional Prize-Collecting Steiner Tree Problem on Trees*

Extended Abstract

Gunnar W. Klau[1], Ivana Ljubić[2], Petra Mutzel[2], Ulrich Pferschy[3], and René Weiskircher[2]

[1] Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB), Germany, klau@zib.de
[2] Institute of Computer Graphics and Algorithms, Vienna University of Technology, Austria
ljubic|mutzel|weiskircher@ads.tuwien.ac.at
[3] Department of Statistics and Operations Research University of Graz, Austria, pferschy@uni-graz.at

Abstract. We consider the fractional prize-collecting Steiner tree problem on trees. This problem asks for a subtree T containing the root of a given tree $G = (V, E)$ maximizing the ratio of the vertex profits $\sum_{v \in V(T)} p(v)$ and the edge costs $\sum_{e \in E(T)} c(e)$ plus a fixed cost c_0 and arises in energy supply management. We experimentally compare three algorithms based on parametric search: the binary search method, Newton's method, and a new algorithm based on Megiddo's parametric search method. We show improved bounds on the running time for the latter two algorithms. The best theoretical worst case running time, namely $O(|V| \log |V|)$, is achieved by our new algorithm. A surprising result of our experiments is the fact that the simple Newton method is the clear winner of the tested algorithms.

1 Introduction

The recent complete deregulation of the Austrian natural gas market leads to new competitive situations for the energy companies. This and the increasing use of biomass for heat generation has lead to a rapid expansion of district heating networks. The planning of these heating networks is one of the major challenges in this area [10]. Here, the input is a set of potential customers with known or estimated heat demands, and a potential network for laying the pipelines (which is usually identical to the street network of the district or village). The costs of the network are dominated by labor and right-of-way charges for laying the pipes and the costs for building the heating plant. The problem for a profit oriented company is to choose a subset of the potential customers and a sub-network of the potential network in order to maximize the profit generated by running the network.

We can formulate this problem as an optimization problem on a graph $G = (V, E)$ where the vertices are associated with profits and the edges with costs. The graph in this application corresponds to the local street map, with the edges representing street segments and vertices representing street intersections and the location of potential customers. The profit associated with a vertex is an estimate of the potential gain of revenue if that customer is connected and therefore receives the service. Vertices corresponding to street intersections have profit zero. The costs associated with an edge are the costs of establishing the connection – laying the pipe on the corresponding street segment. There is a special vertex with profit zero that represents the heating plant and has to be contained in every feasible solution.

* Partly supported by the DFG research center "Mathematics for key technologies" (FZT 86) in Berlin and by the Doctoral Scholarship Programme of the Austrian Academy of Sciences (DOC).

The problem described above is a variant of the well-studied *Steiner tree problem in graphs*, namely the *prize-collecting Steiner tree problem*. This problem is first mentioned by Segev [15] where it appears as a special case of the node-weighted Steiner tree problem and is called the *Single Point Weighted Steiner Tree problem*. The author proves NP-hardness of the problem, presents integer linear programming formulations and uses Lagrangean relaxation and heuristics to compute lower and upper bounds for these formulations, respectively. In [5], Duin and Volgenant relate the node-weighted (and thus also the prize-collecting) variant to the classical Steiner tree problem. They adapt reduction techniques and show how the rooted prize-collecting Steiner tree problem can be transformed into the directed version of the classical Steiner tree problem.

In [7], Fischetti studies the facial structure of a generalization of the problem, the so called *Steiner arborescence problem*. Goemans studies the polyhedral structure of the node-weighted Steiner tree problem [8] and shows that his characterization is complete in case the input graph is series-parallel. Approximation results are given by Bienstock *et al.* [1] and by Goemans and Williamson [9]; the latter present a purely combinatorial $O(n^2 \log n)$-time primal-dual $(2 - \frac{1}{n-1})$-approximation algorithm, where n denotes the number of vertices in the graph and the objective is to minimize the edge costs plus the prizes of the nodes not spanned. For the more realistic objective to maximize the sum of the profits minus the costs, Feigenbaum *et al.* [6] prove that it is NP-hard to approximate the problem to a constant factor.

In this paper, we look at the special case where the potential network is a tree. The reason is that in the real world instances of the problem that we looked at, the periphery of the potential network always consisted of large trees while the center had higher connectivity. Since the problem is NP-hard for general graphs and polynomial time solvable for trees, it makes sense to solve the trees on the periphery first and then solve the core of the network using methods like (fractional) integer linear programming or heuristics.

The energy companies are interested in the fractional version of the problem which maximizes the ratio of the sum of the profits and the sum of the (fixed and variable) costs, since this corresponds to the "return on investment" factor. In order to use a parametric formulation, we need to be able to solve the same problem with linear objective function. In the linear formulation of the problem, the total revenue - the sum of the profits connected to the network minus the sum of the costs - needs to be maximized.

Section 2 contains some preliminaries including the description of a linear time algorithm for optimizing the linear objective function. In Section 3, we present three different algorithms that use the parametric formulation: a binary search algorithm, Newton's method and our new variant based on parametric search. We show a worst case running time of Newton's method of $O(|V|^2)$, and of our new algorithm of $O(|V| \log |V|)$. In Section 4, we report on extensive computational experiments. Surprisingly for us, our experiments show that Newton's method, although having worst case running time of $O(|V|^2)$, outperforms the two other methods on our benchmark set. Finally, in Section 5 we summarize the results.

2 Preliminaries

In this section, we provide some basic definitions and describe a linear time algorithm for solving the linear version of the prize-collecting Steiner tree problem (PCST problem). A closely related dynamic programming algorithm can also be found in [16] (where trees with only node-weights are considered).

Let $G = (V, E)$ be an undirected graph, $r \in V$ a root vertex of G, $p : V \to \mathbb{R}^+ \cup \{0\}$ a profit function on the vertices and $c : E \to \mathbb{R}^+ \cup \{0\}$ a cost function on the edges.
The *Fractional Prize Collecting Steiner Tree problem* consists of finding a connected sub-

graph $T = (V', E')$ of G with $r \in V'$ that maximizes the ratio of the profits and the costs:

$$\text{profit}(T) = \frac{\sum_{v \in V'} p(v)}{c_0 + \sum_{e \in E'} c(e)} \ ,$$

where $c_0 > 0$ represents the cost of the root node, e.g. the fixed costs of the heating plant.

The *Linear Prize Collecting Steiner Tree problem* consists of finding a connected subgraph $T = (V', E')$ of G with $r \in V'$ that maximizes the difference of the profits and the costs:

$$\text{profit}(T) = \sum_{v \in V'} p(v) - \sum_{e \in E'} c(e) \ .$$

Note that fixed costs are irrelevant if we optimize a linear objective function.

If $T = (V, E)$ is a tree with root r, then the function parent(v) assigns every vertex $v \in V \setminus \{r\}$ a unique vertex u which is the vertex following v on the path from v to r. The *subtree* rooted at v consists of all vertices and edges reachable from v without passing the vertex parent(v). The set $C(v)$ of *children* of v is the set that contains all vertices u with parent$(u) = v$. A subtree of T is *optimal*, if there is no other subtree of T with a higher profit. We recursively define a value $l(v)$ and a subtree $T(v)$ for each vertex $v \in V$.

$$l(v) = p(v) + \sum_{u \in C(v)} \max\{0, l(u) - c(u, v)\} \ . \tag{1}$$

The subtree $T(v) = (V(v), E(v))$ with profit $l(v)$ is defined in the following way:

$$V(v) = \{v\} \cup \bigcup_{u \in C(v)} \{V(u) \mid l(u) - c(u, v) \geq 0\}$$
$$E(v) = \bigcup_{u \in C(v)} \{(u, v) \cup E(u) \mid l(u) - c(u, v) \geq 0\} \ .$$

If $c(u, v) > l(u)$ for a vertex u with parent$(u) = v$ it does not pay off to include the subtree rooted at u via edge (u, v) (the only possible connection towards r), and we decide to cut off the edge (u, v) together with the corresponding subtree. This decision can be made locally, as soon as the value $l(u)$ is known. Thus, our algorithm starts by labeling all leaves and ends up at the root vertex r (see Algorithm 1 for an outline).

It is easy to see that the optimal subtree rooted at v is $T(v)$ with $l(v)$ as its profit (the correctness of this algorithm follows easily by induction). Note that even in the case when $l(u) - c(u, v) = 0$ we decide to include the edge (u, v) in the solution, in order to obtain the optimal subtree with the maximum number of vertices and edges. Depending on the application one may decide to exclude these edges in the solution and thus derive an optimal solution with the minimum number of vertices.

Lemma 1. *Algorithm* LinearTree *solves the rooted PCST on trees in* $O(|V|)$ *time.*

When solving the fractional PCST on trees, in contrast to the linear case, we cannot make local decisions anymore without looking at the whole problem. In order to decide if the inclusion of a certain subtree improves our solution, we need to know the profit to cost ratio of the rest of the solution. The following section presents the parametric formulation of the problem that allows us to decide in linear time if a given value t is smaller, equal or greater than the value of an optimal solution to the fractional PCST problem.

Data : A tree $T = (V, E)$ with a fixed root $r \in V$, a profit function p on the vertices and a costs function c on the edges
Result : For each $v \in V$ an optimal subtree $T(v) = (V(v), E(v))$ and a label $l(v) = \text{profit}(T(v))$

```
G' = (V', E') = G;
for v ∈ V do
    l(v) = p(v); V(v) = {v}; E(v) = ∅;
end
repeat
    L = {v ∈ V \ {r} | deg_{G'}(v) = 1};
    for v ∈ L do
        u = parent(v);
        l(u) = l(u) + max{0, l(v) − c(u, v)};
        if l(v) ≥ c(u, v) then
            V(u) = V(u) ∪ V(v);
            E(u) = E(u) ∪ {(u, v)} ∪ E(v);
        end
    end
    Remove the vertices of L from G';
until V' = {r};
```

Algorithm 1: Algorithm LINEARTREE for labeling the vertices in V

3 Algorithms Based on Parametric Formulation

To solve the fractional problem, we first formulate the linear problem with an additional parameter. Then we show how this enables us to solve the fractional problem using our algorithm for the linear problem. The connection between a parametric formulation and the fractional version of the same problem has already been established by Dinkelbach [4].

Let $\mathcal{T}$ be the set of all connected subgraphs $T = (V', E')$ of G that contain the root. We are looking for a graph in $\mathcal{T}$ that maximizes the expression

$$\frac{\sum_{v \in V'} p(v)}{c_0 + \sum_{e \in E'} c(e)} .$$

Now consider the following function $o(t)$:

$$o : \mathbb{R}^+ \to \mathbb{R},\ o(t) = \max_{T=(V',E') \in \mathcal{T}} \sum_{v \in V'} p(v) - t(c_0 + \sum_{e \in E'} c(e)).$$

If we have $o(t^*) = 0$, then it follows that

$$t^* = \frac{\sum_{v \in V^*} p(v)}{c_0 + \sum_{e \in E^*} c(e)}$$

for a certain tree $T^* = (V^*, E^*) \in \mathcal{T}$. We claim that T^* is an optimal solution of the fractional PCST on G. It is obvious that the objective value of T^* for the fractional PCST is t^*. Now assume that there is another tree $T_1 = (V_1, E_1) \in \mathcal{T}$ that has a higher objective value t_1 than t^* with respect to the fractional PCST. Then we have

$$t_1 = \frac{\sum_{v \in V_1} p(v)}{c_0 + \sum_{e \in E_1} c(e)} > t^* \Leftrightarrow 0 < \sum_{v \in V_1} p(v) - t^*(c_0 + \sum_{e \in E_1} c(e)) \ .$$

But this is a contradiction against $o(t^*) = 0$.

Let t^* be the value of the optimal solution of the PCST problem on G and $t \in \mathbb{R}$. Then we have:

$$o(t) = 0 \Leftrightarrow t = t^*, \qquad o(t) < 0 \Leftrightarrow t > t^*, \qquad o(t) > 0 \Leftrightarrow t < t^* \ .$$

Using LinearTree, we can test for any t in linear time if it is smaller, equal or greater than the optimal solution for the fractional PCST. This fact can be used to construct different search algorithms that solve the problem.

There is also a geometric interpretation of our problem. Let $\mathcal{T}$ be again the set of all non-empty subtrees of G. Each $T = (V_T, E_T) \in \mathcal{T}$ defines a linear function $f_T : \mathbb{R}^+ \to \mathbb{R}$ in the following way:

$$f_T(t) = \sum_{v \in V_T} p(v) - t(c_0 + \sum_{e \in E_T} c(e)) \ .$$

Since all vertex profits and edge costs are non-negative, and c_0 is positive, all these linear functions have negative slope. In this geometric interpretation, the function o defined above is the maximum of these functions. Hence it is a piecewise linear, convex, monotonously decreasing function. What we are looking for is the point where o crosses the x-axis. The functions f_T that contain this point correspond to optimal subtrees for the given profits and costs.

3.1 Binary Search

An easy way of building an algorithm for the fractional PCST problem that uses the parametric formulation of the previous section is binary search. We start with an interval (t_l, t_h) that contains t^*. Then we test the mid point t of this interval using the algorithm for the linear problem. This will give us either a proof that t equals t^* or a new upper or lower bound and halve the size of the interval.

Each linear piece of the function $o(t)$ corresponds to a certain subtree T_i of G that is the optimal subtree for a certain range of t. We call the t-values where the optimal tree changes the *breakpoints* of o. For each positive value t, we call the point $p_t = (t, o(t))$ in two-dimensional space the *representative* of t. The following two terminating conditions cover all possible cases and work well in practice:

1. The representatives of the last three values for t are on a straight line. The intersection of this line with the x-axis is t^*. This condition covers the case where t^* is not a breakpoint of o.
2. The line through the representatives of the last two values for t_l crosses the line through the last two representatives of t_h at $(t', 0)$ and $o(t') = 0$. It follows that $t^* = t'$. This covers the case where t^* is a breakpoint of t.

The running time of the algorithm depends on the quality of the initial bounds for t^* and on the density of the straight line segments of o around t^*. This density is not only influenced by the size of the graph but also to a great degree by the values for the profits and costs. Therefore, a good upper bound for the worst case running time that depends only on the size of the input graph cannot be given.

3.2 Newton's Method

We use the adaptation of Newton's iterative method already described by Radzik [13]. Let $\mathcal{T}$ be the set of all subtrees of G that contain the root. We start with $t_0 = 0$. In iteration i, we compute

$$o(t_i) = \max_{T=(V',E') \in \mathcal{T}} \sum_{v \in V'} p(v) - t_i(c_0 + \sum_{e \in E'} c(e))$$

together with the optimal tree $T_i = (V_i, E_i)$ for parameter t_i using the linear algorithm from Section 2. As long as $o(t_i)$ is not zero, we compute t_{i+1} as the fractional objective value of T_i. So we have:

$$t_{i+1} = \frac{\sum_{v \in V_i} p(v)}{c_0 + \sum_{e \in E_i} c(e)} \ .$$

In the course of this algorithm, t_i increases monotonically until t^* is reached. Let l be the index with $t_l = t^*$. Radzik shows in [14] for general fractional optimization problems where all weights are non-negative that $l = O(p^2 \log^2 p)$ where p is the size of the problem (in our case the number of vertices of the problem graph G).

For our specific problem, we can prove a stronger bound for l:

Theorem 1. *Newton's method applied to the fractional prize-collecting Steiner tree problem with fixed costs takes at most $n+2$ iterations where n is the number of vertices of the input tree T.*

Proof. Let t_i be the value of the parameter in iteration i. So we have $t_0 = 0$ and $t_l = t^*$ for the last iteration l. Algorithm LinearTree for solving the linear version of the problem *cuts* certain edges, which means that they are not added to any set $E(v)$. These edges are not part of the optimal subtree of any vertex.

We argue that if an edge is cut in iteration i, it will be cut in every iteration $j > i$ and that in every iteration $0 < i < l$ an edge is cut that was not cut before. It follows that there can be at most $n+2$ iterations.

So assume that edge $e = (u, v)$ was cut in iteration i and that $u = \text{parent}(v)$. It follows that the value $l_i(v) - t_i c(u, v)$ was negative ($l_i(v)$ is the label of vertex v in iteration i). Since $t_{i+1} > t_i$ and $l_{i+1}(v) \leq l_i(v)$ (can be easily shown using induction on the longest distance form v to a leaf), e will also be cut in iteration $i+1$.

Let T_i be the optimal subtree in iteration i. We know that all edges cut in iteration i will also be cut in iteration $i+1$. This implies that T_{i+1} does not contain any vertices that are not contained in T_i. To show that at least one additional edge is cut, we note that t_{i+1} is the objective function value of the optimal tree $T_i = (V_i, E_i)$ for iteration i:

$$t_{i+1} = \frac{\sum_{v \in V_i} p(v)}{c_0 + \sum_{e \in E_i} c(e)} \tag{2}$$

We assume that no new edges are cut in iteration $i+1$. It follows that $T_{i+1} = T_i$ and we have

$$o(t_{i+1}) = \sum_{v \in V_i} p(v) - t_{i+1}(c_0 + \sum_{e \in E_i} c(e)) \tag{3}$$

Putting (2) and (3) together we get:

$$o(t_{i+1}) = \sum_{v \in V_i} p(v) - \frac{\sum_{v \in V_i} p(v)}{c_0 + \sum_{e \in E_i} c(e)}(c_0 + \sum_{e \in E_i} c(e)) = 0 \ .$$

It follows that iteration $i+1$ is the last iteration and that Newton's method is finished. So a new edge is cut in every iteration but the first and the last and it follows that there can be at most $n+2$ iterations. □

Since we can solve the problem for the linear objective function in linear time using the algorithm from Section 2, Newton's Method has a worst case running time of $O(|V|^2)$ for our problem.

Figure 1 shows an example where this worst case running time is reached. If we define the fixed costs $c_0 = 1$, we can show by a coarse estimation of the objective function value for each path starting at r that the solution of Newton's method shrinks only by one vertex in every iteration and that the optimal solution is the root together with vertex v_{n-1}. Therefore, the algorithm executes $n-1$ iterations and since each iteration has linear running time, the total running time of Newton's method on this example is $\Theta(n^2)$.

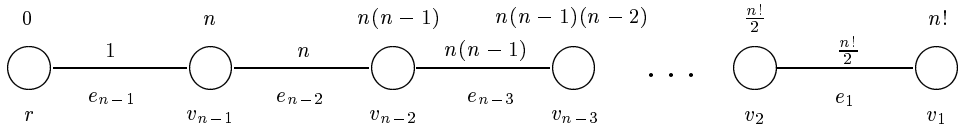

Fig. 1. Worst case example for Newton's Method. The edge costs and vertex profits are above the path while the names of the vertices and edges are below

3.3 A New Algorithm Based on Megiddo's Parametric Search

In this section, we present our new algorithm for the fractional PCST problem which is a variant of parametric search introduced by Megiddo [12]. Furthermore, we suggest an improvement that guarantees a worst case running time of $O(n \log n)$ for any tree G with n vertices.

The idea of the basic algorithm is to simulate the execution of algorithm LinearTree on the unknown edge cost parameter t^* (the objective value of an optimal solution). During the simulation, we keep an interval (t_l, t_h) that contains t^* and is initialized to $(0, \infty)$. Whenever LinearTree has to decide if a certain edge (u, v) is included in the solution, this decision is based on the evaluation of the maximum in (1) on page 3 and depends on the root r_d of a linear function in t given by $l(u) - t\, c(u, v)$.

The decision is clear if r_d is outside (t_l, t_h). Otherwise, we multiply all edge costs of the tree with r_d and execute LinearTree on the resulting problem. The sign of the linear objective function value $o(r_d)$ determines the decision (which enables us to continue the simulation of LinearTree) and r_d either becomes the new upper or lower bound of (t_l, t_h).

There are two possibilities for the algorithm to terminate. The first is that one of the roots we test is t^*. In this case, we can stop without completing the simulation of LinearTree. If we have to simulate LinearTree completely, we end up with an interval for t^*. In this case, we perform depth first search on the edges that we have marked during the simulation to obtain an optimal subtree.

Just as in the algorithm for the linear problem, our algorithm assigns labels to the vertices, but these labels are now linear functions that depend on the parameter t. The algorithm uses a copy G' of the problem tree G. In each phase, all leaves of G' are deleted after the necessary information has been propagated to the parents of the leaves. When the algorithm starts, the label of every vertex is set to the constant function equal to its profit. In the course of the algorithm, these labels change and will correspond to linear functions over the parameter t. We also initialize the interval for t^* to $(0, \infty)$.

When we look at a certain leaf v with label $f_v(t)$ during a phase we compute the linear function $\bar{f}_v(t) = f_v(t) - t \cdot c(e_v)$ where e_v is the edge incident to v. Let r_v be the root of $\bar{f}_v(t)$. For all current leaves, we collect the values r_v, sort them and perform binary search on the roots using the linear algorithm to decide if the value t^* is smaller, greater or equal than a certain root. Note that we do not have to include the roots in the binary search that are outside the current interval for t^*. If there are roots that are inside the current interval, we either find t^* or we end up with a smaller interval.

After the binary search, we know for each leaf v if its root r_v is smaller or greater than t^* (if it is equal, we have already found the solution and the algorithm has stopped). We delete all leaves whose root is smaller than t^* from G'. For all other leaves v, we add the function $\bar{f}_v(t)$ to the label of its parent and delete it, too. Now the next phase of the algorithm starts with the vertices that have become leaves because of the deletion of the current leaves.

Algorithm 2 shows the procedure in more detail. For convenience, all data is represented by linear functions over t during this section. In particular, profit values $p(v)$ are represented by $p(v) + t \cdot 0$ and edge costs $c(u, v)$ are given as $0 - t \cdot c(u, v)$. Addition and subtractions of

linear functions are defined in the obvious way. Note that in this way subtracting an edge cost is done by *adding* the corresponding linear function.

Data : A tree $G = (V, E)$ with fixed root r, a non-negative profit function p on the vertices and a positive cost function c on the edges
Result : The value t^* of an optimal subtree of G
$G' = (V', E') = G$;
for $v \in V'$ **do** $l(v) = p(v)$;
$t_l = 0$, $t_h = \infty$;
while $V' \neq \{r'\}$ **do**
  $L = \{v \in V' \setminus \{r'\} \mid \deg(v) = 1\}$;
  $B = \{t_l, t_h\}$;
  for $v \in L$ **do**
    $u = \text{parent}(v)$;
    $t_v = \text{root of } (l(v) + c(u, v))$;
    if $t_v \in (t_l, t_h)$ **then** $B = B \cup \{t_v\}$;
  end
  Perform binary search on B using the linear algorithm and update t_l and t_h;
  if $t_l \equiv t_h$ **then return** t_l;
  for $v \in L$ **do**
    $u = \text{parent}(v)$;
    if $t_v \geq t_h$ **then** $l(u) = l(u) + l(v) + c(u, v)$;
  end
  Remove the vertices of L from G';
end
Perform depth first search on the marked edges to construct a subtree with value t^*;
return t^*;

Algorithm 2: Algorithm FractionalTree for the fractional PCST

The correctness of FractionalTree follows from the general principle of Meggido's method [12]. The running time of the algorithm is dominated by the calls to the linear algorithm. The binary search is performed by solving $O(\log(|B|))$ instances of a linear PCST with profits and costs determined by the parameter t. Since it may happen that the graph contains only one leaf in every iteration (G may be a path) the number of iterations can be n. The worst case example for Newton's method in Section 3.2 is also a worst case example for this algorithm. Thus the overall running time of FractionalTree is $O(|V|^2)$.

Improvement Through Path Contraction. If there is no vertex in G with degree two, FractionalTree already has a running time of $O(n \log n)$ for a tree with n vertices: In this case we delete at least half the vertices of the graph in every iteration by deleting all leaves. It will follow from the proof of Theorem 2 that this property is sufficient for the improved running time.

We will remove the remaining obstacles in the graph, namely vertices of degree two, by performing a reduction of all paths in the tree. This must be done in every iteration since the removal of all leaves at the end of the previous iteration may generate new paths. The idea of the reduction is based on the fact that the subtree situated at the end of a path can only contribute to the optimal solution if the complete path is also included. Otherwise, only a connected subset of the path can be in the optimal solution.

More formally, a subset of V is a path denoted by $P := \{v_0, v_1, \ldots, v_m, v_{m+1}\}$ if v_0 has degree greater two or is the root, v_{m+1} does not have degree two and all other vertices are of degree two. To fix the orientation we assume that v_0 is included in the path from v_1 to

r. Since we want to contract the m vertices of the path to a single vertex, trivial cases can be excluded by assuming $m \geq 2$. In an optimal solution either there exists a vertex $v_q \in P$ such hat $v_1, \ldots, v_q$ are the only vertices of P in the solution, or P is completely contained in the solution and connects a possible subtree rooted at v_{m+1} to r.

The procedure ContractPath determines the best possible candidate for v_q and contracts the path by adding an artificial edge from v_0 to v_q with cost equal to the value of the complete subpath including $v_1, \ldots, v_{q-1}$, and a second artificial edge from v_q to v_{m+1} that models the cost of traversing the vertices $v_{q+1}, \ldots, v_m$. The path contraction is invoked at the beginning of every iteration in Algorithm FractionalTree (see the appendix for a detailed description of ContractPath).

The main theoretical result of this paper is stated in the following theorem:

Theorem 2. *The running time of Algorithm* FractionalTree *with* ContractPath *is in* $O(n \log n)$.

Proof. (*Sketch*) To find v_q, we need to compute the maximum of m linear functions, which can be done in time $O(m \log m)$ (see [2] for a proof). The resulting piecewise linear function has at most m breakpoints. In every iteration there is a number of breakpoints from ContractPath and a number of leaves with corresponding root values to be considered. We use binary search in each iteration to find a new interval (t_l, t_h) including neither breakpoints nor roots thus resolving the selection of v_q and the final decision on all leaves.

If k is the size of the graph at the beginning of an iteration, then the binary search performs a logarithmic number of calls to the algorithm that solves the linear PCST. Therefore, a single iteration takes time $O(k \log k)$.

It can be shown that applying the procedure ContractPath to every non trivial path guarantees that FractionalTree with ContractPath deletes at least one third of the vertices in each iteration. Note that if there is no vertex in G with degree two, FractionalTree already deletes half the vertices of the graph by deleting all leaves. Since the size of the graph is reduced by a constant fraction after each iteration, the total running time sums up to $O(n \log n)$. See the appendix for a detailed proof. □

4 Computational Experiments

We generated two different test sets of graphs to test the performance of the algorithms presented in Section 3. The first set consists of randomly generated trees where every vertex has at most two children while the second set contains random trees where each vertex can have up to ten children. In both sets, the costs of each edge and the profit of each vertex is a random integer from the set $\{1, 2, \ldots, 10.000\}$. Both sets contain 100 trees for each number of vertices from 1,000 to 10.000 in steps of 500 vertices. The fixed costs for all problem instances was chosen as 1000 times the number of vertices in the graph. This produced solution that contained around 50% of all vertices for the graphs where each vertex has at most 10 children. For the graphs where each vertex has at most two children, the percentage was around 35%. To execute the three algorithms on the test sets as a documented and repeatable experiment and for analyzing the results, we used the tool set `ExpLab` [11].

Figure 2 shows the average number of calls over all trees with the same number of vertices for the three algorithms and the two benchmark sets. The number of calls grows very slowly with the size of the graphs for all three algorithms. In fact, the number of calls barely grows with the number of vertices in the graph for Newton's method.

Our variant of Megiddo's method needs more calls than the other two methods. For the leaves of the tree, the algorithm behaves just like binary search. The reason why the number of calls is higher than for binary search is that our new algorithm not only executes calls at the leaf level but also higher up in the tree. These are usually very few and not on every

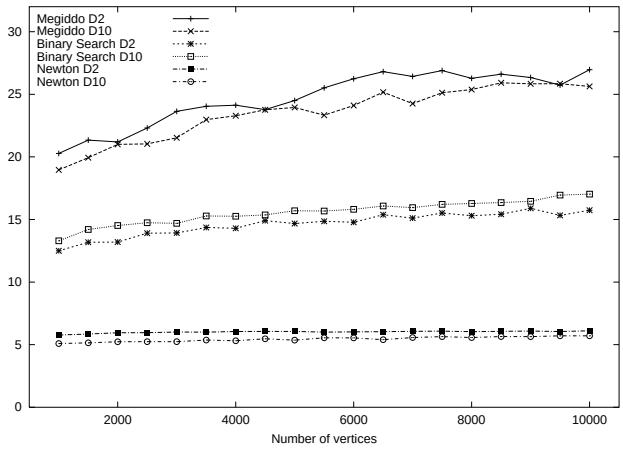

Fig. 2. The average number of calls to the linear algorithm executed by the three algorithms on the benchmark set with maximum degree 2 and maximum degree 10

level. So on a level where additional calls have to be made, there are usually only one or two open decisions. Therefore, the binary search in our new algorithm can not effectively be used except at the leaf level. Because of this fact, the pure binary search algorithm can "jump" over some decisions that parametric search has to make on higher levels.

The reason why Newton's method needs less calls than the binary search method is the random nature of our problem instances. Binary search starts with a provable upper bound for t^* which in our case is the sum of all vertex profits divided by the fixed costs. This upper bound is far away from the objective value of the optimal solution. After the first iteration of Newton's method, the value t is the objective function value of the whole tree which is a good lower bound for the optimal solution because the profits and costs are random and with the fixed costs we have chosen, the optimal tree contained 35-50% of all vertices. Therefore, Newton's method needs only a small number of steps to reach the optimal solution and the number of calls grows only very slowly with the size of the graphs.

Figure 3 shows that the number of calls to the linear algorithm determines the running time: our new algorithm is the slowest and Newton's method the fastest. The running times grow slightly faster than linear with the size of the graphs. Since each call to the algorithm for the linear problem needs linear time, the fact that the number of calls grows with the size of the graph (albeit very slowly) is the reason for this behavior. We executed the experiments on a PC with a 2.8 GHz Intel Processor with 2GB of memory running Linux. The running time of all three algorithms grows linearly with the size of the graphs and even for the graphs with 10.000 vertices, the problems can be solved in less than 1.8 seconds.

We also executed an experiment where we used only the 100 graphs of the test set with maximum degree 10 that have 10.000 vertices. We increased the fixed costs c_0 exponentially and ran all three algorithms on the 100 graphs for each value of c_0. We started with $c_0 = 100$ (where the solution contained only a few vertices) and multiplied the fixed costs by 10 until we arrived at 10^{11} (where the optimal solution consisted almost always of the whole tree).

Figure 4 shows how the time needed by the three algorithms depends on fixed costs. It is remarkable that for small fixed costs, binary search is faster than Newton's method but for fixed costs of more than 10.000, Newton's method is faster. The reason is the same we have already given for the better performance of Newton's method in our first experiments. For large fixed costs, the percentage of the vertices contained in an optimal solution rises and so the value of the first solution that Newton's method tests, which is the value of the whole graph, is already very close to the optimal value. Binary search has to approach the

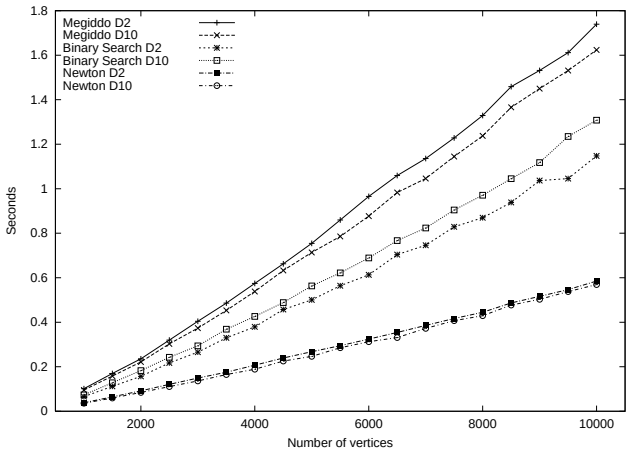

Fig. 3. The average time used by the three algorithms on the two benchmark sets

optimum solution from the provable upper bound for the objective function value which is far away from the optimal solution when this solution is large and therefore contains many edges.

Parametric search is not much slower than binary search for high fixed costs. As the plot shows, the reason is not that parametric search performs significantly better for higher fixed costs but that the performance of binary search deteriorates for the reasons given in the last paragraph.

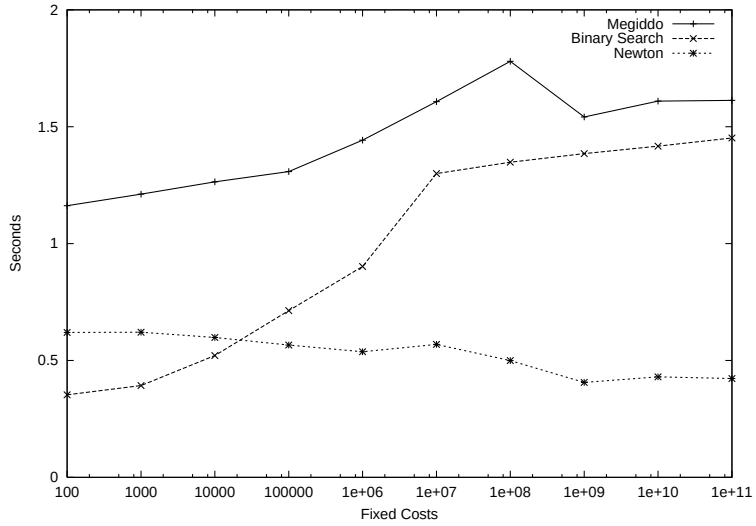

Fig. 4. Time used by the three algorithms for growing fixed costs (logarithmic x-axis)

5 Conclusions

In this paper, we have presented three algorithms for solving the fractional prize-collecting Steiner tree problem (PCST problem) on trees $G = (V, E)$. We have shown that Newton's

algorithm has a worst case running time of $O(|V|^2)$. We have also presented a variant of parametric search and proved that the worst case running time of this new algorithm is $O(|V| \log |V|)$. Our computational results show that Newton's method performs best on randomly generated problems while a simple binary search approach and our new method are considerably slower. For all three algorithms, the running time grows slightly faster than linear with the size of our test instances.

Acknowledgments

We thank Günter Rote and Laurence Wolsey for giving us useful pointers to the literature.

References

[1] D. Bienstock, M. X. Goemans, D. Simchi-Levi, and D. Williamson. A note on the prize collecting traveling salesman problem. *Mathematical Programming*, 59:413–420, 1993.
[2] J. D. Boissonnat and M. Yvinec. *Algorithmic Geometry*. Cambridge University Press, 1998.
[3] H. Davenport and A. Schinzel. A combinatorial problem connected with differential equations. *American J. Mathematics*, 87:684–694, 1965.
[4] W. Dinkelbach. On nonlinear fractional programming. *Management Science*, 13:492–498, 1967.
[5] C. W. Duin and A. Volgenant. Some generalizations of the Steiner problem in graphs. *Networks*, 17(2):353–364, 1987.
[6] J. Feigenbaum, C. H. Papadimitriou, and S. Shenker. Sharing the cost of multicast transmissions. *Journal of Computer and System Sciences*, 63(1):21–41, 2001.
[7] M. Fischetti. Facets of two Steiner arboresence polyhedra. *Mathematical Programming*, 51:401–419, 1991.
[8] M. X. Goemans. The Steiner tree polytope and related polyhedra. *Mathematical Programming*, 63:157–182, 1994.
[9] M. X. Goemans and D. P. Williamson. The primal-dual method for approximation algorithms and its application to network design problems. In D. S. Hochbaum, editor, *Approximation algorithms for NP-hard problems*, pages 144–191. P. W. S. Publishing Co., 1996.
[10] J. Hackner, 2002. personal communication.
[11] S. Hert, L. Kettner, T. Polzin, and G. Schäfer. Explab. http://explab.sourceforge.net, 2002.
[12] N. Megiddo. Combinatorial optimization with rational objective functions. *Mathematics of Operations Research*, 4(4):414–424, 1979.
[13] T. Radzik. Newton's method for fractional combinatorial optimization. In *Proceedings of 33rd Annual Symposium on Foundations of Computer Science*, pages 659–669, 1992.
[14] T. Radzik. Fractional combinatorial optimization. In D. Z. Du and P. Pardalos, editors, *Handbook of Combinatorial Optimization*, pages 429–478. Kluwer, 1998.
[15] A. Segev. The node-weighted Steiner tree problem. *Networks*, 17:1–17, 1987.
[16] L. A. Wolsey. *Integer Programming*. John Wiley, New York, 1998.

6 Appendix: The Path Contraction Method

The following pseudo code gives a detailed description of the algorithm ContractPath.

Data : A labeled tree $T = (V, E)$ with fixed root r;
a path in T $v_0, v_1, \ldots, v_m, v_{m+1}$, $m > 2$
Result : A labeled tree $T = (V, E)$ with fixed root r
$end[1] = 0$;
for $j = 1$ *to* m **do**
$end[j] := end[j-1] + l(v_j) + c(v_{j-1}, v_j)$;
end
$f(t) = \max_{j=1}^{m} end[j]$;
$B = \{t \in (t_l, t_h) \mid t \text{ is breakpoint of } f(t)\} \cup \{t_l, t_h\}$;
Perform binary search on B using the modified linear algorithm and update t_l and t_h;
choose q s.t. $end[q] = f(t)$ for $t \in (t_l, t_h)$;
$c(v_0, v_q) := \sum_{k=1}^{q-1}(l(v_k) + c(v_{k-1}, v_k)) + c(v_{q-1}, v_q)$;
$c(v_q, v_{m+1}) = \sum_{k=q+1}^{m}(l(v_k) + c(v_{k-1}, v_k)) + c(v_m, v_{m+1})$;
Remove vertices $v_1, \ldots, v_{q-1}, v_{q+1}, \ldots, v_m$ from T;

Algorithm 3: Algorithm ContractPath to remove all nontrivial paths from a tree

The main difficulty in ContractPath is the computation of the maximum value over all subpaths from v_0 to some vertex $v_j \in P$. Clearly, the total values of all subpaths $end[j]$ are linear (decreasing) functions and their maximum $f(t)$ is piecewise linear convex. It is known from elementary computational geometry [3] that $f(t)$ contains at most m breakpoints and can be computed in $O(m \log m)$ time. A simple algorithm for doing so can for example be found in [2]. After generating these $O(m)$ breakpoints, binary search is performed in the same way as in FractionalTree to find a smaller interval (t_l, t_h) containing t^*. Restricting $f(t)$ to this interval yields a single linear function and yields the desired vertex v_q.

Before analyzing the running time of FractionalTree we introduce an elementary lemma from graph theory. Let g_i denote the number of vertices in a graph with degree i.

Lemma 2. *If G is a tree then* $g_1 \geq \dfrac{n - g_2}{2} + 1$.

Proof. Consider the following elementary calculation:

Since in a tree with n vertices the total number of edges in every tree is $n - 1$ we get immediately

$$2(n-1) = \sum_{i=1}^{n} i\, g_i = g_1 + \sum_{i=2}^{n} i\, g_i \quad \wedge \quad \sum_{i=1}^{n} g_i = n \Longleftrightarrow g_1 = n - \sum_{i=2}^{n} g_i.$$

Plugging the two together we get

$$n - 2 = \sum_{i=2}^{n} (i-1) g_i \geq g_2 + \sum_{i=3}^{n} 2 g_i \quad \Rightarrow \quad \sum_{i=3}^{n} g_i \leq \frac{n - g_2}{2} - 1$$

which yields the result. □

Now we can give a proof of the $O(n \log n)$ running time of FractionalTree.

Proof. (*Theorem 2*) At the end of every iteration of FractionalTree the set L (i.e. all vertices with degree 1) are removed. However, some vertices with degree 2 (one from every

call to ContractPath and possible trivial paths) may remain. There will always be one such vertex followed by a vertex of degree ≥ 3. Let us now estimate the minimal cardinality of L in such a tree. Therefore, we will first consider a tree *without* vertices of degree 2 and then try to exchange a maximum number of edges with two new edges and a new vertex of degree 2 in-between them.

In a tree with $g_2 = 0$ the number of leaves can be bounded from Lemma 2 by $g_1 \geq n/2+1$. Of the $n-1$ edges of the tree there are exactly g_1 edges leading to leaves. The remaining $n-1-g_1$ "inner edges" may each be replaced by a a new vertex and two edges. The total number of vertices n' in this new graph is given by $n + (n-1-g_1) \leq 3n/2$. The number of leaves is still $g_1 \geq n'/3$. Hence, we have shown that in any graph with a structure resulting from path contraction at least one third of all vertices are leaves and are thus removed in every iteration. This bounds the number of iterations by $O(\log n)$. Roughly bounding the cardinality of B by n in every iteration, this would yield an overall running time of $O(n \log 2n)$. However, we can do better by taking into account that the linear time effort required for every call during the binary search is spent on instances which are getting smaller and smaller from one iteration to the other.

At the end of every iteration we have made a **final** decision for all removed vertices in L whether they should be included in a solution (from the point of view of their parent vertex) or not. This decision reduces the remaining problem by the amount of removed vertices and all linear PCST computations to resolve the breakpoints of the subsequent iterations have to solve only a problem of reduced size.

Denote by $T(n)$ the total running time of FractionalTree on a problem with n vertices and by $L(n)$ the number of vertices which are removed in the first iteration (all those with degree 1 and some path vertices in ContractPath). Considering the linear running time of the calls to LinearTree as performed in the binary search leads to a total running time of

$$T(n) \leq n \log(L(n)) + T(n - L(n)) < n \log n + T(n - L(n)).$$

We claim that $T(n)$ is in $O(n \log n)$. Obviously, the above expression is increasing with decreasing $L(n)$. But plugging in even the smallest possible value $L(n) = n/3$ yields

$$\begin{aligned} T(n) &< n \log n + T(2/3\, n) \\ &\leq n \log n + 2/3\, n \log(2/3\, n) + T(4/9\, n) \\ &\leq n \log n + 2/3\, n \log(2/3\, n) + 4/9\, n \log(4/9\, n) + T(8/27\, n) \\ &\leq \dots \\ &\leq \sum_{j=0}^{\log n} (2/3)^j n \, \log((2/3)^j n) < n \log n \sum_{j=0}^{\infty} (2/3)^j < 3n \log n. \end{aligned}$$

□

This shows that our new algorithm for the Prize Collecting Steiner Tree Problem on a rooted tree has indeed running time $O(n \log n)$.