Theoretical Computer Science 410 (2009) 5273–5284

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Approximating the Maximum Internal Spanning Tree problem

Gábor Salamon

Department of Computer Science and Information Theory, Budapest University of Technology and Economics, 1117 Budapest, Magyar tudósok körútja 2., Hungary

ARTICLE INFO

Keywords:
Approximation algorithm
Spanning tree leaves
Hamiltonian path

ABSTRACT

Given an undirected connected graph G we consider the problem of finding a spanning tree of G which has a maximum number of internal (non-leaf) vertices among all spanning trees of G. This problem, called Maximum Internal Spanning Tree problem, is clearly NP-hard since it is a generalization of the Hamiltonian Path problem. From the optimization point of view the Maximum Internal Spanning Tree problem is equivalent to the Minimum Leaf Spanning Tree problem. However, the two problems have different approximability properties. Lu and Ravi proved that the latter has no constant factor approximation – unless P = NP –, while Salamon and Wiener gave a linear-time 2-approximation algorithm for the Maximum Internal Spanning Tree problem.

In this paper, we improve this approximation ratio by giving an $\mathcal{O}(|V(G)|^4)$-time 7/4-approximation algorithm for graphs without pendant vertices. Our approach is based on the successive execution of local improvement steps. We use a linear programming formulation and a primal–dual technique to prove the approximation ratio. We also investigate the vertex-weighted case, that is to find a spanning tree of a vertex-weighted graph G in which the weight sum of internal vertices is maximal among all spanning trees of G. For this problem we present a $(2\Delta(G) - 3)$-approximation algorithm, where $\Delta(G)$ is the maximum vertex-degree of G. A slight modification of this algorithm ensures a 2-approximation whenever the input graph is claw-free. Both algorithms run in $\mathcal{O}(|V(G)|^4)$ time for graphs with no pendant vertices.

1. Introduction

The design process of different networks generally uses graph theory as a convenient 0 approach. In these models the connectivity requirements are often represented in such a way that a spanning tree of the network graph must be found as a feasible solution. The objective function varies from one design problem to other thus arising a family of combinatorial optimization problems. For a survey on spanning tree optimization, the reader is referred to [1].

In many cases, the value or goodness of a solution (to be either minimized or maximized) is a function of the degree distribution of the spanning tree itself. We refer such problems as *degree-based spanning tree optimization problems.*

In this paper we consider the Maximum Internal Spanning Tree (MaxIST) problem which is to find a spanning tree of a given graph with a maximum number of internal vertices (non-leaves). Formally

Problem 1 (Maximum Internal Spanning Tree (MaxIST)).
Input: An undirected connected graph G.
Goal: Find a spanning tree T of G such that T has a maximum number of internal vertices among all spanning trees of G. (A vertex of T is internal if its degree in T is at least 2.)

E-mail address: gsala@cs.bme.hu.

0304-3975/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2009.08.029

Clearly, the MaxIST problem is a degree-based spanning tree optimization problem. Besides, it is a generalization of the Hamiltonian Path problem as Hamiltonian paths of G are exactly its spanning trees with $|V(G)| - 2$ internal vertices. This directly implies that the problem is NP-hard. Notice that from an optimization point of view, the MaxIST problem is equivalent to the Minimum Leaf Spanning Tree problem. The later is to find a spanning tree with a minimum number of leaves (1-degree vertices).

From an approximation point of view, however, the two problems behave differently. On one hand, Lu and Ravi [2] proved that there is no constant factor approximation for the Minimum Leaf Spanning Tree problem unless P = NP. On the other hand, complementing the cost function, and thus counting internal vertices instead of leaves, leads to a better situation. By slightly modifying the depth-first search algorithm, one can get a spanning tree that is either a Hamiltonian path or a tree with independent leaves. Such a tree always has at least half as many internal vertices as the optimal one. This yields a linear-time 2-approximation for the MaxIST problem [3].

In this paper we improve this approximation ratio to 7/4. The algorithm can work on arbitrary graphs, however, the above approximation ratio is guaranteed only when the input graph has no pendant (1-degree) vertex. The removal of this extra condition from the analysis is subject of further research.

The proof of the approximation ratio is based on a primal–dual linear programming technique. The main idea is to define a primal program such that any spanning tree is a feasible solution with a value equals to the number of internal vertices. We apply this to the optimal spanning tree. Then the dual problem is used to upper bound the optimum solution of the MaxIST problem by the means of the value of our algorithmic solution. Primal–dual techniques of this kind were already used for approximation spanning tree optimization problems (see e.g. [4]). Degree-based problems, however, were more likely approximated with local improvement techniques [2,5]. Our work combines the two approaches.

We also consider the vertex-weighted case, when the input graph has positive weights on its vertices and the aim is to find a spanning tree maximizing the weighted sum of internal vertices. Formally the Maximum Weighted Internal Spanning Tree (MaxWIST) problem is:

Problem 2 (Maximum Weighted Internal Spanning Tree (MaxWIST)).
Input: An undirected connected graph G with vertex-weights $c : V \longrightarrow \mathbf{Q}_+$
Goal: Find a spanning tree T such that $\sum_{v \in I(T)} c(v)$ is maximized among all spanning trees of G, where $I(T)$ stands for the set of internal vertices of T.

We present two algorithms to solve this problem. The first one provides a $(2\Delta - 3)$-approximation for general graphs, while the second yields a 2-approximation whenever the input graph is claw-free, that is, it has no induced subgraph isomorphic to $K_{1,3}$. (Here, Δ stands for the maximum degree of the input graph.)

All our algorithms are based on the technique of local improvement rules. Each of them basically starts by building an arbitrary spanning tree. Then small local changes (defined by improvement rules) are applied successively as far as possible. When no more rule is applicable, we obtain a spanning tree with specific properties. These properties then can be used to prove the approximation ratio.

The local improvement technique has already turned to be very efficient in case of different NP-hard degree-based spanning tree optimization problems. The Minimum Degree Spanning Tree problem is to find a spanning tree of the input graph in which the maximum degree is as small as possible. Fürer and Raghavachari [5] gave a local improvement based approximation algorithm for this problem. Their algorithm is the best possible (unless P = NP) as it finds a spanning tree whose maximum degree is at most one more than the optimum solution.

The Maximum Leaf Spanning Tree problem is to find a spanning tree with a maximum number of leaves. Lu and Ravi used the local improvement technique to give an $\mathcal{O}(n^7)$-time 3-approximation algorithm for this problem. Later they used a technique that builds the spanning tree according to the local structure of the underlying graph (instead of applying local improvements on a pre-built spanning tree) [6]. This way they reduced the running time to be almost linear while preserved the approximation factor of 3. The currently known best approximation factor is 2, and is achieved by Solis-Oba [7] whose algorithm is based on the local improvement idea in [6].

Finally we mention another degree-based spanning tree optimization problem which is also a generalization of the Hamiltonian Path problem. A vertex of a spanning tree is called branching if its degree in the spanning tree is at least 3. The Minimum Branching Spanning Tree problem is to find a spanning tree T of the input graph G such that T has a minimum number of branchings among all spanning trees of G. Gargano et al. [8] considered this problem and proved that it is NP-complete to decide whether a spanning tree with at most k branchings exists (for any fixed k). They also gave an algorithm that finds a single-branching spanning tree if each 3-element independent set of the input graph G has a degree sum of at least $|V(G)| - 1$. Salamon [9] gave a few approximability results concerning this problem. A standard approximation preserving reduction from Set Cover to Minimum Branching Spanning Tree shows that any approximation factor better than $\mathcal{O}(\log |V(G)|)$ would imply that $NP \subseteq DTIME(n^{\mathcal{O}(\log \log n)})$. Salamon also gave an algorithm that finds a spanning tree with $\mathcal{O}(\log |V(G)|)$ branchings whenever each vertex of the input graph has a degree of $\Omega(n)$. However, the general approximability properties of the Minimum Branching Spanning Tree problem are still undiscovered. Our local improvement algorithm for the MaxIST problem can be used as a heuristic to minimize the number of branchings, but no approximation ratio has been proven.

The rest of the paper is organized as follows. Section 2 introduces our notations and gives the basic definitions. Section 3 presents our new 7/4-approximation algorithm for the MaxIST problem. Section 4 deals with the weighted case considering both general and claw-free graphs. Section 5 provides some implementation details and a brief running time analysis of our algorithms.

2. Notation and basic definitions

By a graph $G = (V, E)$ we mean an undirected simple connected graph. Throughout this paper we suppose that G has no pendant vertices. (A vertex is *pendant* if its degree is 1.) Let $T = (V, E')$ be a spanning tree of G. The elements of E' are called *tree edges*, the edges in $E \setminus E'$ are called *non-tree edges*. We say that a vertex x is a *G-neighbor* of a vertex y if $(x, y) \in E$, and that x is a *T-neighbor* of y if (x, y) is a tree edge. A set $X \subseteq V$ is *G-independent* if it spans no edges of G, and a vertex x is *G-independent from* a set $Y \subseteq V$ if x has no G-neighbors in Y. We use $d_G(x)$ and $d_T(x)$ to denote the number of G-neighbors and T-neighbors of x, respectively. A vertex x is a *leaf* of T, a *forwarding vertex* of T, or a *branching* of T if $d_T(x) = 1$, $d_T(x) = 2$, or $d_T(x) \geq 3$, respectively. Forwarding vertices and branchings are called *internal vertices* of T. We denote the set of leaves by $L(T)$ and the set of internal vertices by $I(T)$. For vertices x and y, we denote by $P_T(x, y)$ the unique path in T between vertices x and y. The *branching $b(l)$ of a leaf l* is the branching being closest to l in T. Note that if T is not a Hamiltonian path then each leaf l has a unique branching $b(l)$. The path $P_T(l, b(l))$ is called the *branch* of l and is denoted by $br(l)$. The vertex $b^-(l)$ is the T-neighbor of $b(l)$ being in the branch of l. If l is a leaf and (l, x) is a non-tree edge then $x^{\to l}$ is the predecessor of x and $b(l)^{\to x}$ is the successor of $b(l)$ along the path $P_T(l, x)$. The branch of l is *short* if $b^-(l) = l$, and *long* otherwise. A leaf l is called short (long) if its branch is short (long). $L_s(T)$ stands for the set of short leaves and $L_g(T)$ for the set of long leaves. A leaf l is called *(x-)supported* if there is a non-tree edge (l, x) with $x \notin br(l)$. If l is a long leaf and there exists a non-tree edge (l, x) such that $x \in br(l)$ then $x^{\to l}$ is called an *l-leafish vertex*. In this case, the vertex x is called the *base* of $x^{\to l}$. The set of l-leafish vertices is denoted by $F(l)$. Note that not every long leaf has a leafish vertex in its branch. We denote by $L_p(T)$ the set of long leaves of T having no leafish vertices in their branch. For a set $X \subseteq V$, the graph $G[X]$ is the subgraph of G spanned by X. The *trunk* of tree T is $T[V \setminus \cup_{l \in L(T)}(br(l) \setminus \{b(l)\})]$. The maximum vertex-degree in G is denoted by Δ. For the sake of simplicity, we use $X + x$, and $X - x$ instead of $X \cup \{x\}$, and $X \setminus \{x\}$, respectively.

3. Maximizing the number of internal vertices

In this section we present a local search based algorithm for finding a spanning tree with many internal vertices. The algorithm starts by creating a DFS-tree. Then it executes local improvement rules as long as possible in order to reduce the number of leaves. Finally we obtain either a Hamiltonian path or a locally optimal spanning tree (a LOST). We show that a LOST is a 7/4-approximation for the MaxIST problem.

To prove the approximation ratio we construct a linear programming problem formulation of which a spanning tree T' is always a feasible solution with value $|I(T')|$. This implies that the solution corresponding to the optimal tree T^* has a value of $|I(T^*)|$. Suppose that we can algorithmically construct some spanning tree T. To establish some approximation ratio of α, it is enough to give a particular dual solution with a value of $\alpha|I(T)|$ corresponding to T, as in this case

$$|I(T^*)| \leq \text{ LP-optimum } \leq \alpha|I(T)|$$

and so

$$\frac{|I(T^*)|}{|I(T)|} \leq \alpha.$$

In fact, our algorithm builds a spanning tree T by a successive application of improvement rules. Then, to obtain the approximation ratio, we give two different dual solutions both corresponding to T. The first solution bounds the approximation factor if T has many short branches, and the second one does this job if T has many long branches. Both dual solutions are based on a set S of forwarding vertices of T that are G-independent from the leaves.

The structure of the improvement rules is as follows. A precondition part determines when the particular rule can be executed. An action part defines the replacement of some (1 or 2) edges of T by non-tree edges to obtain a spanning tree T' having less leaves than T.

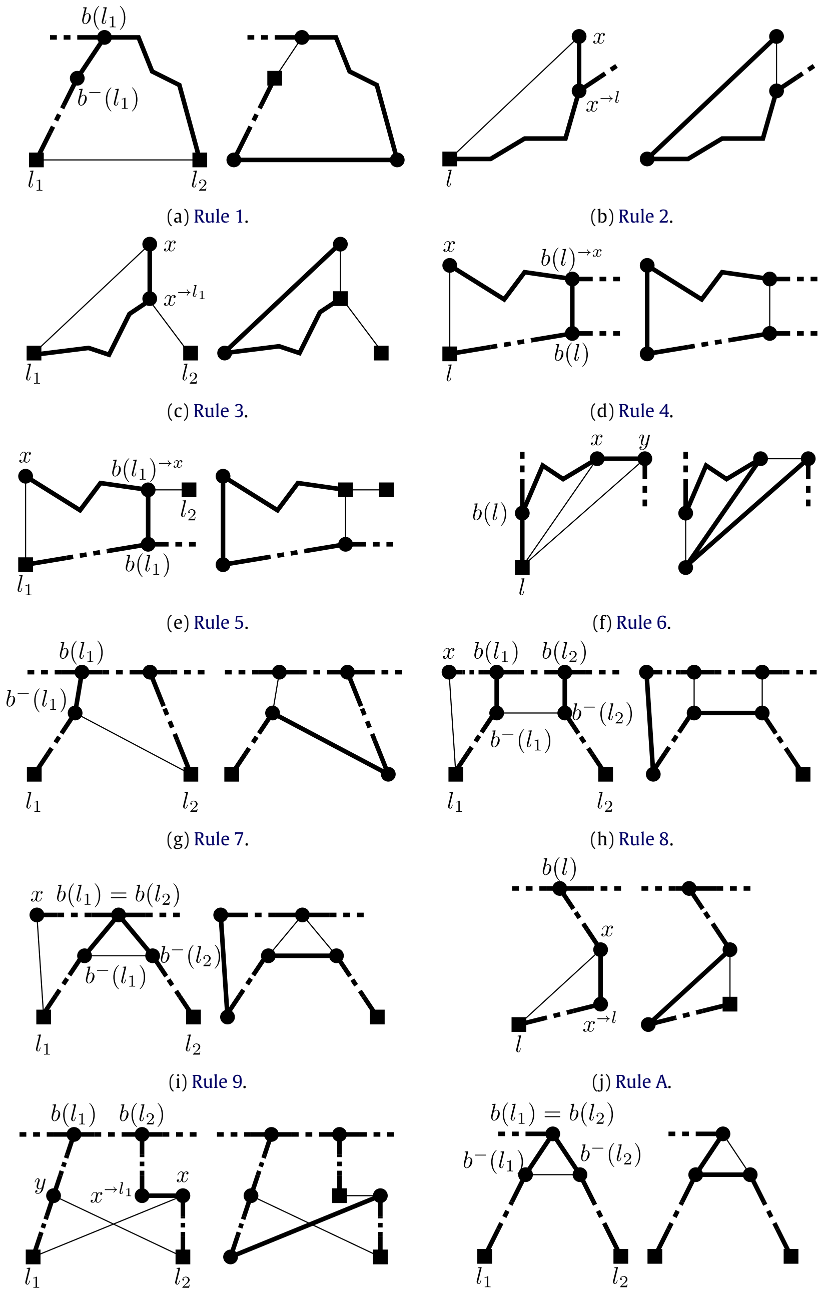
We define a LOST to be a spanning tree T in which none of these improvement rules can be applied, that is, the preconditions of the rules are not satisfied. If T has many short branches then the violated preconditions of Rules 1–6, if T has many long branches then the violated preconditions of Rules 1 and 7–14 ensure a set S that is G-independent from the leaves and whose size is big enough to yield the approximation ratio via a suitable solution of the dual LP-problem. Our local improvement rules are as follows.

Rule 1. Precondition: *T has two leaves l_1 and l_2 such that $(l_1, l_2) \in E(G)$.* **Action:** Let $E(T') = E(T) + (l_1, l_2) - (b(l_1), b^-(l_1))$. (See Fig. 1(a).)

Rule 2. Precondition: *T has an x-supported leaf l such that $d_T(x^{\to l}) > 2$.* **Action:** Let $E(T') = E(T) + (l, x) - (x, x^{\to l})$. (See Fig. 1(b).)

Rule 3. Precondition: *T has an x-supported leaf l_1 and a leaf l_2 such that $d_T(x^{\to l_1}) = 2$, and that $(l_2, x^{\to l_1})$ is a non-tree edge.* **Action:** Let $E(T') = E(T) + (l_1, x) - (x, x^{\to l_1})$. Apply Rule 1 on leaves $l_2, x^{\to l_1}$. (See Fig. 1(c).)

Rule 4. Precondition: *T has an x-supported leaf l such that $d_T(b(l)^{\to x}) > 2$.* **Action:** Let $E(T') = E(T) + (l, x) - (b(l), b(l)^{\to x})$. (See Fig. 1(d).)



(a) Rule 1. (b) Rule 2. (c) Rule 3. (d) Rule 4. (e) Rule 5. (f) Rule 6. (g) Rule 7. (h) Rule 8. (i) Rule 9. (j) Rule A. (k) Rule 13. (l) Rule 14.

Fig. 1. Local improvement steps for creating a LOST (squares represent leaves, circles represent internal vertices).

Rule 5. Precondition: T has an x-supported leaf l_1 and a leaf l_2 such that $d_T(b(l_1)^{\to x}) = 2$, and that $(l_2, b(l_1)^{\to x})$ is a non-tree edge. **Action:** Let $E(T') = E(T) + (l_1, x) - (b(l_1), b(l_1)^{\to x})$. Apply Rule 1 on leaves $l_2, b(l_1)^{\to x}$. (See Fig. 1(e).)

Rule 6. Precondition: T has a short leaf l, and an edge (x, y) such that (l, x) and (l, y) are both non-tree edges. **Action:** Let $E(T') = E(T) + (l, x) + (l, y) - (x, y) - (l, b(l))$. (See Fig. 1(f).)

Rule 7. Precondition: T has a long leaf l_1 and a leaf l_2 such that $(b^-(l_1), l_2) \in E(G)$. **Action:** Let $E(T') = E(T) + (b^-(l_1), l_2) - (b^-(l_1), b(l_1))$. (See Fig. 1(g).)

Rule 8. Precondition: T has an x-supported long leaf l_1 and a long leaf l_2 such that $x \notin br(l_2) - b(l_2)$, $b(l_1) \neq b(l_2)$, and $(b^-(l_1), b^-(l_2)) \in E(G)$. **Action:** Let $E(T') = E(T) + (l_1, x) + (b^-(l_1), b^-(l_2)) - (b(l_1), b^-(l_1)) - (b(l_2), b^-(l_2))$. (See Fig. 1(h).)

Rule 9. Precondition: T has an x-supported long leaf l_1 and a long leaf l_2 with $b(l_1) = b(l_2)$ such that $d_T(b(l_1)) \geq 4$, $x \notin br(l_2)$, and $(b^-(l_1), b^-(l_2)) \in E(G)$. **Action:** Let $E(T') = E(T) + (l_1, x) + (b^-(l_1), b^-(l_2)) - (b(l_1), b^-(l_1)) - (b(l_2), b^-(l_2))$. (See Fig. 1(i).)

The following rule differs from the above ones as – while applied on the long branch of a leaf l – it changes neither the trunk nor any other branch of T. Only the branch of l is modified such that one of its leafish vertices becomes leaf and l becomes an internal vertex.

Rule A. Precondition: T has a leaf l and an l-leafish vertex $x^{\to l}$ with base x. **Action:** Let $E(T') = E(T) + (l, x) - (x, x^{\to l})$. (See Fig. 1(j).)

We can use Rule A to decrease the number of leaves as follows.

Rule 10. Precondition: T has two leaves l_1 and l_2, and an l_1-leafish vertex u such that (u, l_2) is a non-tree edge. **Action:** Apply Rule A on u to make it a leaf and l_1 an internal vertex. Then apply Rule 1 on leaves l_2, u.

Rule 11. Precondition: T has two leaves l_1 and l_2, an l_1-leafish vertex u, and an l_2-leafish vertex v such that (u, v) is a non-tree edge. **Action:** Apply Rule A on u and on v to make both of them a leaf while l_1 and l_2 an internal vertex. Then apply Rule 1 on leaves u, v.

Rule 12. Precondition: T has two leaves l_1 and l_2, and an l_1-leafish vertex u such that $(u, b^-(l_2))$ is a non-tree edge. **Action:** Apply Rule A on u to make it a leaf and l_1 an internal vertex. Then apply Rule 7 on leaves u, l_2.

The following rules do not change the number of leaves. They can be applied only on pairs of branches that contain no leafish vertices.

Rule 13. Precondition: T has two leaves $l_1, l_2 \in L_p(T)$ and two non-tree edges (l_1, x) and (l_2, y) such that $x \in br(l_2)$ and $y \in br(l_1)$. **Action:** Let $E(T') = E(T) + (l_1, x) - (x, x^{\to}(l_1))$. (See Fig. 1(k).)

Rule 14. Precondition: T has two leaves $l_1, l_2 \in L_p(T)$ such that $b(l_1) = b(l_2)$, $d_T(b(l_1)) = 3$ and $(b^-(l_1), b^-(l_2))$ is a non-tree edge. **Action:** Let $E(T') = E(T) + (b^-(l_1), b^-(l_2)) - (b(l_2), b^-(l_2))$. (See Fig. 1(l).)

Although, the number of leaves remains the same when applying Rules 13 and 14 the sum $\sum_{l \in L_p} |br(l)|$ is always decreased. This is a crucial point of our running time analysis that can be found in Section 5.

Definition 1. A spanning tree T is a *locally optimal spanning tree (LOST)* if none of Rules 1–14 can be applied on it.

Now we build an approximation algorithm for the MaxIST problem using the above improvement rules.
Algorithm LOST. Create a DFS-tree. Then apply Rules 1–14 as long as possible. If several rules can be applied, execute the one with the lowest number.
Now we can state the main theorem of this section.

Theorem 2. *Algorithm LOST is an $\mathcal{O}(|V|^4)$-time 7/4-approximation for the MaxIST problem in graphs that have no pendant vertices.*

The proof of the running time is postponed to Section 5. The approximation ratio is established by the following lemma.

Lemma 3. *Let T be a LOST of a graph G that has no pendant vertices, and let T^* be a spanning tree of G with a maximum number of internal vertices. Then*

$$\frac{|I(T^*)|}{|I(T)|} \leq 7/4.$$

Proof. First observe some basic properties of T which are immediate consequences of the definition of a LOST.

Property 1. *$L(T)$ forms a G-independent set. (As Rule 1 is no more applicable on T.)*

Property 2. *Let l be an x-supported leaf. Then $d_T(x^{\to l}) = d_T(b(l)^{\to x}) = 2$. Furthermore, both $x^{\to l}$ and $b(l)^{\to x}$ are G-independent from the set $L(T) - l$. (As Rules 2–5 are no more applicable on T.)*

Property 3. *Let l be a short leaf. Then no two G-neighbors of l are T-neighbors. (As Rule 6 is no more applicable on T.)*

Property 4. *Let l_1 be a long leaf. Then no leaf $l_2 \neq l_1$ is a G-neighbor of $b^-(l_1)$. (As Rule 7 is no more applicable on T.)*

Property 5. *Let l_1 and l_2 be leaves. Then l_2 is G-independent from the set of l_1-leafish vertices. (As Rule 10 is no more applicable on T.)*

Property 6. *Let l_1 and l_2 be leaves. Then every l_1-leafish vertex is G-independent from the set of l_2-leafish vertices. (As Rule 11 is no more applicable on T.)*

Property 7. *Let l_1 and l_2 be leaves. Then $b^-(l_2)$ is G-independent from the set of l_1-leafish vertices. (As Rule 12 is no more applicable on T.)*

Property 8. *Let l_1 and l_2 be long leaves such that their branches do not contain leafish vertices, and $(b^-(l_1), b^-(l_2))$ is a non-tree edge. Then $b(l_1) = b(l_2)$ and $d_T(b(l_1)) = 3$. (As Rules 8, 9 and 13 are no more applicable on T.) Moreover, as Rule 14 is no more applicable, T must have exactly 3 leaves. From this point we suppose that T has at least 4 leaves. (Since it is easy to see that a LOST with 3 leaves satisfies the approximation factor of 7/4.) As a result, $l_1, l_2 \in L_p(T)$ implies $(b^-(l_1), b^-(l_2)) \notin E(G)$.*

To prove the approximation ratio we use a primal–dual linear programming approach. Let us recall a formulation of the spanning tree polyhedron [10]:

$$\mathcal{SP}(G) = \{x \mid \forall S \subseteq V : x(S) \le |S| - 1, -x(V) \le -(|V| - 1), \forall e \in E : 0 \le x(e)\},$$

where $x(S) = \sum_{e \in E(G[S])} x(e)$ is the sum of x over all edges spanned by S.

Let us consider the following linear program ($\delta(v)$ is the set of edges incident to v):

$$\begin{array}{lll} \text{maximize} & \sum_{v \in V} z(v) & \\ \text{subject to} & x \in \mathcal{SP}(G) & \\ & -\sum_{e \in \delta(v)} x(e) + z(v) \le -1 & \text{for all } v \in V \\ & 0 \le z(v) \le 1 & \text{for all } v \in V. \end{array}$$

Now let T^* be a spanning tree which is an optimal solution of the MaxIST problem. Then from T^* we create a solution **P** of this primal problem as follows. We set $x(e) = 1$ when e is an edge of T^*, and $z(v) = 1$ when v is an internal vertex of T^*. All other variables are set to 0. Since T^* is a spanning tree, it is easy to see that this solution is feasible and has a value of $\text{val}(\mathbf{P}) = |I(T^*)|$. If **OPT** is the optimal solution of the LP-problem itself then we obtain

$$|I(T^*)| = \text{val}(\mathbf{P}) \le \text{val}(\mathbf{OPT}). \tag{1}$$

The dual of the above program is:

$$\begin{array}{lll} \text{minimize} & \sum_{S \subseteq V} (|S| - 1) y(S) - (|V| - 1)t - \sum_{v \in V} w(v) + \sum_{v \in V} r(v) & \\ \text{subject to} & \sum_{e \in E(G[S])} y(S) - t - \sum_{e \in \delta(v)} w(v) \ge 0 & \text{for all } e \in E \qquad (2) \\ & w(v) + r(v) \ge 1 & \text{for all } v \in V \\ & y(S), t, w(v), r(v) \ge 0 & \text{for all } S \subseteq V, v \in V. \end{array}$$

Let T be the LOST created by our algorithm. We consider two different dual solutions corresponding to T. The first one is used in case of many short branches, and the second one is used in case of many long branches.

Let l be a short leaf. We define the set

$$Q(l) = \left\{x^{\to l} \mid \exists x : (l, x) \in E(G) \setminus E(T)\right\} \cup \left\{b(l)^{\to x} \mid \exists x : (l, x) \in E(G) \setminus E(T)\right\}.$$

Note that $Q(l) \ne \emptyset$, since $d_G(l) \ge 2$. Let

$$Q = \bigcup_{l \in L_s(T)} Q(l).$$

The first dual solution $\mathbf{D}_1$ is constructed as follows. Let $y(V) = 1, y(Q) = 1, w(v) = 1$ for each $v \in L(T) \cup Q$, and $r(v) = 1$ for each $v \in V \setminus (L(T) \cup Q)$. All other variables are set to 0.

To see the feasibility of this solution, it is enough to check (2) for all edges of G. As $y(V) = 1$, only the edges of $G[L(T) \cup Q]$ could violate the inequality. However, by Property 1, there is no edge spanned by $L(T)$, and by Properties 2 and 3, there is no edge between $L(T)$ and Q. Thus $y(Q) = 1$ ensures the feasibility.

Let us define

$$c_1 = \frac{|Q|}{|I(T)|}.$$

The value of this dual solution is

$$\begin{aligned} \text{val}(\mathbf{D}_1) &= |V| - 1 + |Q| - 1 - |L(T)| - |Q| + |V| - |L(T)| - |Q| \\ &= 2(|I(T)| - 1) - |Q| < (2 - c_1)|I(T)|. \end{aligned}$$

To construct the second dual solution $\mathbf{D}_2$ we denote the set of l-leafish vertices by $F(l)$, and define the set of leafish vertices to be

$$F = \bigcup_{l \in L_g(T)} F(l).$$

Recall that $L_p(T)$ denotes the set of long leaves that have no leafish vertex in their branch. We use B_p^- to denote the set $\cup_{l \in L_p(T)} b^-(l)$. We immediately obtain $|B_p^-| = |L_p(T)|$.

Dual variables are set as: $y(V) = 1$, $y(F(l) + l) = 1$ for each $l \in L_g(T) \setminus L_p(T)$, $y(\{l, b^-(l)\}) = 1$ for each $l \in L_p(T)$, $w(v) = 1$ for each $v \in (L(T) \cup F \cup B_p^-)$, $r(v) = 1$ for each $v \in V \setminus (L(T) \cup F \cup B_p^-)$. All other variables are set to 0.

To see the feasibility of this solution, it is enough to check (2) for all edges of G. As $y(V) = 1$, only the edges of $G[(L(T) \cup F \cup B_p^-)]$ could violate the inequality. However, by Properties 1 and 4–8, the graph $G[(L(T) \cup F \cup B_p^-)]$ has no edge between different branches of T. On the other hand, the edges of $G[(L(T) \cup F \cup B_p^-)]$ within a single branch of T are also covered by some set S with $y(S) = 1$.

Let us define

$$c_2 = \frac{|V| - |L_s(T)|}{|I(T)|}.$$

Then as $|I(T)| = |V| - |L_s(T)| - |L_g(T)|$ we obtain $|L_g(T)| = (c_2 - 1)|I(T)|$.

The value of the solution is

$$\begin{aligned}\mathrm{val}(\mathbf{D}_2) &= |V| - 1 + |F| + |L_p(T)| - |L(T)| - |F| - |L_p(T)| + |V| - |L(T)| - |F| - |L_p(T)| < 2|I(T)| - |F| - |L_p(T)| \\ &\le 2|I(T)| - |L_g(T)| = (3 - c_2)|I(T)|.\end{aligned}$$

Here we have used that $|L_g(T)| \le |F| + |L_p(T)|$.

Using the duality theorem of linear programming we get

$$\mathrm{val}(\mathbf{OPT}) \le \min\left(\mathrm{val}(\mathbf{D}_1), \mathrm{val}(\mathbf{D}_2)\right).$$

Then (1) implies that

$$|I(T^*)| \le \min\left(\mathrm{val}(\mathbf{D}_1), \mathrm{val}(\mathbf{D}_2)\right).$$

As a result we have

$$\frac{|I(T^*)|}{|I(T)|} \le \min\left(2 - c_1, 3 - c_2\right). \qquad (3)$$

Now let $N(L_s(T))$ be the set of G-neighbors of short leaves. Observe that by the definition of Q, each element of $N(L_s(T))$ has a T-neighbor in Q. Moreover, by Property 2, all vertices of Q are forwarding vertices of T. Thus

$$|N(L_s(T))| \le 2|Q| = 2c_1|I(T)|. \qquad (4)$$

Let us remark that the condition $d_G(l) \ge 2$ is used here to ensure that the set $Q(l)$ is not empty, for each leaf l. This latter fact is necessary to upper bound $|N(L_s(T))|$ by a function of $|Q|$.

Let us recall that the *scattering number* [11] of a non-complete graph is

$$\mathrm{sc}(G) = \max\left\{\mathrm{comp}(G[V \setminus X]) - |X| : \emptyset \subset X \subset V, \mathrm{comp}(G[V \setminus X]) \ge 2\right\}, \qquad (5)$$

where $\mathrm{comp}(G[V \setminus X])$ is the number of components of $G[V \setminus X]$.

Salamon and Wiener [12] proved that each spanning tree of G has at least $\mathrm{sc}(G) + 1$ leaves. This yields

$$|I(T^*)| \le |V| - \mathrm{sc}(G) - 1. \qquad (6)$$

As the short leaves of T are G-independent, $G[V - N(L_s(T))]$ has at least $|L_s(T)|$ components implying that $\mathrm{sc}(G) \ge |L_s(T)| - |N(L_s(T))|$.

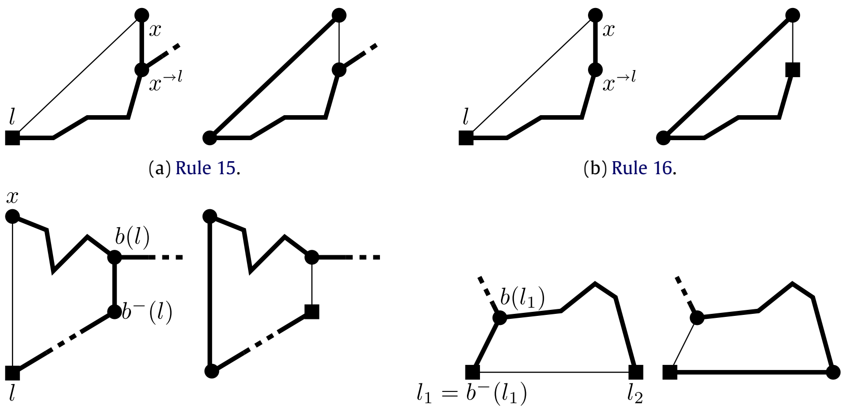
Thus, by (4) and (6), and the definition of c_2 we have

$$|I(T^*)| < |V| - \mathrm{sc}(G) \le |V| - |L_s(T)| + |N(L_s(T))| \le (c_2 + 2c_1)|I(T)|. \qquad (7)$$

If $c_1 \ge 1/4$ or $c_2 \ge 5/4$ then by (3), otherwise by (7), we obtain $\frac{|I(T^*)|}{|I(T)|} \le 7/4$. $\square$

4. Maximum Weighted Internal Spanning Tree

Let $G = (V, E)$ be a graph without pendant vertices and with a positive weight-function $c : V \longrightarrow \mathbf{Q}_+$ on its vertices. The MaxWIST problem aims to find a spanning tree T of G that maximizes the weight sum of internal vertices, that is $c(I(T)) = \sum_{v \in I(T)} c(v)$. Obviously, this problem is NP-hard, as it contains the unweighted version, the MaxIST problem, as a special case. In this section we present a $(2\Delta - 3)$-approximation algorithm for general graphs which is further refined to get a 2-approximation algorithm for claw-free graphs. The main idea of the algorithms is similar to the one of unweighted case. We use local improvement steps to obtain a locally optimal tree. Then we prove that certain properties of such a tree guarantee the desired approximation ratio. The proof is, however, different from the one of Section 3. Instead of using linear programming, it is based on a mapping that maps every leaf of the spanning tree into a greater-weight internal vertex.



(a) Rule 15. (b) Rule 16. (c) Rule 17. (d) Rule 18.

Fig. 2. Local improvement rules for creating a WLOST.

4.1. General graphs

Let us consider an arbitrary spanning tree T of G. In order to get a good approximation we apply local improvement rules as long as possible. Each such rule either turns a leaf into an internal vertex or it replaces a leaf with an other one of strictly smaller weight. The weighted sum of leaves decreases in both cases. Similarly to the unweighted case, each of the following rules has a precondition and an action part of the same semantics.

Rule 15. Precondition: T has an x-supported leaf l such that $d_T(x^{\to l}) > 2$. **Action:** Let $E(T') = E(T) + (l, x) - (x, x^{\to l})$ (Fig. 2(a)).

Rule 16. Precondition: T has a leaf l, and a non-tree edge (l, x) such $d_T(x^{\to l}) = 2$, and $c(x^{\to l}) < c(l)$. **Action:** Let $E(T') = E(T) + (l, x) - (x, x^{\to l})$ (Fig. 2(b)).

Rule 17. Precondition: T has an x-supported leaf l such that $c(b^-(l)) < c(l)$ **Action:** Let $E(T') = E(T) + (l, x) - (b(l), b^-(l))$ (Fig. 2(c)).

Rule 18. Precondition: T has a short leaf l_1 and a leaf l_2 such that $(l_1, l_2) \in E(G)$. **Action:** Let $E(T') = E(T) + (l_1, l_2) - (b(l_1), b^-(l_1))$ (Fig. 2(d)).

We can construct an approximation algorithm for the MaxWIST problem using the above rules. First we have a definition.

Definition 4. A spanning tree T is a *weighted locally optimal spanning tree (WLOST)* if none of Rules 15–18 can be applied on it.

Algorithm WLOST. Create an arbitrary spanning tree. Then apply Rules 15–18 as long as possible. If several rules can be applied, execute the one with the lowest number.

Theorem 5. *Algorithm WLOST is an $\mathcal{O}(|V|^4)$-time $(2\Delta - 3)$-approximation for the MaxWIST problem in graphs that have no pendant vertices.*

The proof of the running time is postponed to Section 5. The following lemma establishes the approximation ratio.

Lemma 6. *Let T be a WLOST of a vertex-weighted graph $G = (V, E, c)$ that has no pendant vertices. Then $(2\Delta - 3)c(I(T)) \geq c(V)$.*

Proof. First, observe some basic properties of T that are immediate consequences of the definition of a WLOST.

Property 9. *If l is a leaf of T and (l, x) is a non-tree edge then $d_T(x^{\to l}) = 2$ and $c(x^{\to l}) \geq c(l)$. (As Rules 15 and 16 are no more applicable on T.)*

Property 10. *If l is a supported leaf of T then $c(b^-(l)) \geq c(l)$. (As Rule 17 is no more applicable on T.)*

Property 11. *If l is a short leaf of T then l is G-independent from the set $L(T)$. (As Rule 18 is no more applicable on T.)*

We define a mapping $f : L(T) \longrightarrow I(T)$ as follows: for a leaf l, let x be the vertex such that (l, x) is a non-tree edge and the length of path $P_T(l, x)$ is the maximum possible. Such an x always exists as $d_G(l) \geq 2$. If $br(l)$ is a short branch or $x \in br(l)$ then let $f(l) = x^{\to l}$. Otherwise, let $f(l) = b^-(l)$. This means that each long leaf l is mapped to an internal vertex of its own branch $br(l)$. Thus the images of long leaves are disjoint. By the above properties, it is easy to see that for every leaf l, we have $c(f(l)) \geq c(l)$, and $d_T(f(l)) = 2$.

The mapping f can be used to establish the approximation ratio. Unfortunately, several short leaves can be mapped to the same vertex y. The following proposition is used to upper bound the number of such leaves.

Proposition 7. *For any vertex $y \in V$, we have $|\{l : y = f(l)\}| \leq 2(\Delta - 2)$.*

Proof. Let $l_1, l_2, \ldots, l_r$ be leaves of T with $f(l_i) = y$ $(1 \leq i \leq r)$. As shown above, $d_T(y) = 2$. Let y_1 and y_2 be the T-neighbors of y. Recall that a long leaf is mapped to a vertex of its own branch. This implies that neither y_1 nor y_2 is a leaf of T. Supposing the contrary, namely e.g. $d_T(y_1) = 1$, y must be in the long branch of y_1, and at least $r - 1$ of the branches $br(l_i)$ $(1 \leq i \leq r)$ must be short. Moreover, by the definition of the mapping f, graph G must have edges (l_i, y_1) that is a contradiction to Property 11. Hence, at least 2 edges incident to y_1 (and y_2, respectively) are tree edges, and so at most $\Delta - 2$ are non-tree edges. This shows that $r \leq 2(\Delta - 2)$, yielding the proposition. □

Using Proposition 7, and the fact that the images of long leaves are G-independent, we obtain

$$\begin{aligned}\sum_{v\in L(T)} c(v) &= \sum_{l\in L_g(T)} c(l) + \sum_{l\in L_s(T)} c(l)\\ &\leq \sum_{l\in L_g(T)} c(f(l)) + \sum_{l\in L_s(T)} c(f(l)) \leq 2(\Delta - 2) \sum_{v\in I(T)} c(v). \qquad (8)\end{aligned}$$

Hence adding $\sum_{v\in I(T)} c(v)$ to both sides we obtain

$$\sum_{v\in V} c(v) \leq (2\Delta - 3) \sum_{v\in I(T)} c(v)$$

which proves the approximation ratio. □

4.2. Claw-free graphs

In this subsection we extend the above algorithm by an additional improvement step in order to get a 2-approximation algorithm for the MaxWIST problem in claw-free graphs—that is, graphs without induced $K_{1,3}$. Observe that (8) would guarantee an approximation ratio of 2 if we could find a WLOST without short branches. The new improvement rule does exactly this job, namely it converts short branches to long ones. Throughout this subsection, G is supposed to be claw-free. First we point out a property of WLOSTs of claw-free graphs.

Let T be a WLOST of G, and l be a short leaf of T. Furthermore, let $l, x_1, x_2, \ldots, x_k$ are the T-neighbors of $b(l)$. Then by Property 9, none of $x_1, x_2, \ldots, x_k$ is a G-neighbor of l. Thus, vertices $x_1, x_2, \ldots, x_k$ must span a complete subgraph of G, since otherwise $b(l), l, x_i, x_j$ would induce a $K_{1,3}$ for some i, j. As a result – using Property 9 – all the vertices x_i are internal vertices of T for $i \geq 1$. Thus we have

Property 12. *If l is a short leaf of T and the T-neighbors of $b(l)$ are $l, x_1, \ldots, x_k$, then $x_1, x_2, \ldots, x_k$ are all internal vertices of T and they induce a complete subgraph of G.*

Using this property, we can now give the additional improvement rule to decrease the number of short branches, while not changing the set of leaves.

Rule 19. Precondition: T has a short leaf l, and the T-neighbors of $b(l)$ are $l, x_1, \ldots, x_k$ such that for some $1 \leq i \leq k$ the vertex x_i is a branching, or x_i has a T-neighbor $v_i \neq b(l)$ which is an internal vertex. **Action:** Let $E(T') = E(T) \setminus \{(b(l), x_j)\}_{j=1..k, j\neq i} \cup \{(x_i, x_j)\}_{j=1..k, j\neq i}$.

Definition 8. A WLOST is called a *refined WLOST (RWLOST)* if Rule 19 cannot be applied on it.

Using the above Rule 19, we can improve Algorithm WLOST to obtain a better approximation ratio for claw-free graphs.
Algorithm RWLOST. Create an arbitrary spanning tree. Then apply Rules 15–19 as long as possible. If several rules can be applied, execute the one with the lowest number.

Theorem 9. *Algorithm WLOST is an $\mathcal{O}(|V|^4)$-time 2-approximation for the MaxWIST problem in claw-free graphs that have no pendant vertices.*

Proof. Let T be an RWLOST of a claw-free graph $G = (V, E, c)$ that has no pendant vertices. We prove that in this case $c(I(T)) \geq \frac{1}{2}c(V)$. This yields the approximation ratio of 2. At first, we show that T has no short branches. Suppose the contrary, namely let l be a short leaf. Let $l, x_1, \ldots, x_k$ be the T-neighbors of $b(l)$. As T is an RWLOST, Rule 19 cannot be applied on it. Hence, each vertex x_i is a forwarding vertex with T-neighbors $b(l)$ and v_i, where v_i is a leaf of T (for $1 \leq i \leq k$). As a result, G has only $2k+2$ vertices (l, $b(l)$, x_i's, and v_i's). Then Properties 9 and 11 imply that l is not a G-neighbor of x_i's or v_i's, respectively. This gives $d_G(l) = 1$, a contradiction. Therefore all branches of T are long. Now recall (8) and reformulate it for a WLOST without short branches.

$$\sum_{v\in L(T)} c(v) = \sum_{l\in L_g(T)} c(l) \leq \sum_{l\in L_g(T)} c(f(l)) \leq \sum_{v\in I(T)} c(v).$$

Again adding $\sum_{v\in I(T)} c(v)$ to both sides we conclude that

$$\sum_{v\in V} c(v) \leq 2 \sum_{v\in I(T)} c(v). \quad \square$$

5. Running time analysis

In this section we give some implementation details and a brief running time analysis of our algorithms. We suppose that graph G is given by its adjacency matrix. For all of our algorithms, we maintain some additional data structures representing the current spanning tree T. The purpose of these data structures is to ease the precondition-checking, that is finding the next rule to execute. The data structures are updated after each rule application step.

Besides the adjacency matrix of G we use the following data structures while running our algorithms.

(i) We maintain the edge-list of the current spanning tree T. This represents the solution to be constructed. This list is built in $\mathcal{O}(|V|^2)$ time during the initial traversal. As each rule adds and removes only a constant number of edges, the list can be updated in $\mathcal{O}(|V|)$ time after each rule execution step.

(ii) We have a list of leaves of T and in addition for each leaf l we store:
   (a) $b(l)$
   (b) the vertex $x^{\to l}$ for every vertex x
   (c) the vertex $b(l)^{\to x}$ for every vertex x.
   This *leaf-information* structure is a navigation tool which is useful when an improvement step is based on a non-tree edge incident to a leaf. It can be built (and rebuilt after each rule execution) in $\mathcal{O}(|V|^2)$ time by executing a traversal from each leaf. (The maintenance of the leaf-list itself needs only constant time after each improvement step.) Let us note that most of the rules would enable a minor and so faster update of this data structure. However, several rules heavily modify the structure. For this reason, we always rebuild the leaf-information structure to make our analysis less complex.

(iii) For each forwarding vertex v we store the leaf whose branch contains v. (A special indicator is used instead when v is a trunk-vertex.) This structure is created and maintained together with the leaf-information structure without any extra time cost.

As a consequence, we obtain that the update of the used data structures needs $\mathcal{O}(|V|^2)$ time after each rule execution.

Some of the preconditions require to determine the list of short (or long) leaves. This can be done in $\mathcal{O}(|V|)$ time by iterating through the list of leaves and checking for each leaf l whether $b^-(l) = l$. If yes then l is short otherwise l is long. Note that $b^-(l)$ can be found in constant time using the leaf-information structure as $b^-(l) = b(l)^{\to l}$.

5.1. *Algorithm LOST*

Leafish vertices play an important role in Algorithm LOST. Their list can be built in $\mathcal{O}(|V|)$ time. To this aim, firstly we check for each long leaf l whether $(l, b(l)) \in E(G)$. If yes then we add $b^-(l)$ to the list of leafish vertices. Secondly, for each forwarding vertex x we determine the branch that contains x. If $x \in br(l)$ for some l and $(x, l) \in E(G)$ then we add $x^{\to l}$ to the list of leafish vertices. When building the list of leafish vertices, we can also create a list to represent the leaves in $L_p(T)$. Initially this list contains all leaves of T. Then when a vertex $x \in br(l)$ is found to be leafish then l is removed from the list. As a result, we can build the list representing $L_p(T)$ in $\mathcal{O}(|V|)$ time.

The precondition part of the individual rules is tested as follows.

Rule 1: We consult the leaf-list and for every pair l_1, l_2 of leaves we check in the adjacency matrix whether $(l_1, l_2) \in E(G)$. This needs $\mathcal{O}(|V|^2)$ time.

Rule 2: For each leaf l we get the list of non-tree edges from the adjacency matrix. Then for each non-tree edge (l, x), we check $x \notin br(l)$ and $d_T(x^{\to l}) > 2$. We use the leaf-information structure to get $x^{\to l}$ from x. The degree is checked in the edge-list of T. Thus we need constant time for a given (l, x) pair, and $\mathcal{O}(|V|^2)$ time in total.

Rule 3: For each leaf l_1 we get the list of non-tree edges from the adjacency matrix. Then for each non-tree edge (l_1, x), we check $x \notin br(l)$ and $d_T(x^{\to l_1}) = 2$. Unfortunately, if we directly check the adjacency between $x^{\to l_1}$ and each leaf l_2 then we need $\mathcal{O}(|V|^3)$ time in total. Therefore instead of doing this check directly, we build a list from vertices $x^{\to l_1}$. This can be done in $\mathcal{O}(|V|^2)$ time. Note that each vertex is contained at most once in this list. The rule can be executed if there is a leaf l_2 and an element v of the list such that $(l_2, v) \in E(G)$. This check can be done in $\mathcal{O}(|V|^2)$ time which is also the total time requirement for this rule.

Rule 4: For every leaf l, and for every non-tree edge (l, x) we check $x \notin br(l)$ then determine $b(l)^{\to x}$ and check $d_T(b(l)^{\to x}) > 2$ in constant time using the leaf-information structure. The total time needed is $\mathcal{O}(|V|^2)$.

Rule 5: For every leaf l_1, and for every non-tree edge (l_1, x) we check $x \notin br(l_1)$ then determine $b(l_1)^{\to x}$ and check $d_T(b(l_1)^{\to x}) = 2$ in constant time using the leaf-information structure. Then, analogously to the case of Rule 3, we use an extra data structure to find a leaf l_2 that is neighboring to $b(l_1)^{\to x}$. The total time requirement is $\mathcal{O}(|V|^2)$.

Rule 6: For every tree edge (x, y), and for every short leaf l, we check whether both (x, l), and (y, l) are edges of G. This needs $\mathcal{O}(|V|^2)$ time.

Rule 7: For every pair of long leaves l_1, l_2 we check whether $(l_1, b^-(l_2)) \in E(G)$. This needs $\mathcal{O}(|V|^2)$ time.

Rule 8: We need some extra consideration at this rule to avoid the unnecessary increase of time complexity. First we build a $|L_g(T)| \times |L_g(T)|$-matrix M to indicate the pairs of long leaves for which the rule is possibly applicable. $M(l_1, l_2)$ is set to 1 whenever $(b^-(l_1), b^-(l_2)) \in E(G)$ and $b(l_1) \neq b(l_2)$. All other elements of M are 0. This can be done in $\mathcal{O}(|V|^2)$ time. Then for each leaf l_1 we consider each non-tree edge (l_1, x) for which $x \notin br(l_1)$. There are three cases: (i) there is a non-tree edge (l_1, x) with x being a trunk-vertex; (ii) there are two non-tree edges (l_1, x) and (l_1, y) with x and y in different branches; (iii) all non-tree edges (l_1, x) end in the same branch of l_2. In the first two cases, the rule can be executed to l_1 and any other leaf. In the third case we set $M(l_1, l_2)$ to 0 indicating that l_1 and l_2 cannot be used together to execute the rule. This checking process needs $\mathcal{O}(|V|^2)$ time. Finally, we look for a pair (l_1, l_2) with $M(l_1, l_2) = 1$, and execute the rule on them. Applying the extra data structure (M) results that the total time requirement remains $\mathcal{O}(|V|^2)$.

Rule 9: We apply the idea of Rule 8. The only difference is in the initial construction of matrix M. Here, we set $M(l_1, l_2) = 1$ if $d_T(b(l_1)) \geq 4$, $(b^-(l_1), b^-(l_2)) \in E(G)$ and $b(l_1) = b(l_2)$. The total time needed is again $\mathcal{O}(|V|^2)$.

Rule 10: For each leaf l and leafish vertex u we check whether $(l, u) \in E(G)$. This needs $\mathcal{O}(|V|^2)$ time. The total time requirement after the implied execution of Rule 1 remains $\mathcal{O}(|V|^2)$.

Rule 11: For every pair u, v of leafish vertices we check that u and v are in different branches and that $(u, v) \in E(G)$. This is done in $\mathcal{O}(|V|^2)$ time. The implied application of Rule 1 does not increase this time requirement.

Rule 12: For each leaf l and for each leafish vertex u we check $u \notin br(l)$ and $(u, b^-(l)) \in E(G)$. This needs $\mathcal{O}(|V|^2)$ time. This time complexity is not increased by the implied application of Rule 7.

Rule 13: Firstly we create an $|L_p(T)| \times |L_p(T)|$-matrix M and for each leaf $l_1 \in L_p(T)$ we set $M(l_1, l_2)$ to 1 if there is a non-tree edge (l_1, x) such that $x \in br(l_2)$. All other elements of M are 0. This is done in $\mathcal{O}(|V|^2)$ time. Then we check whether there exist two leaves $l_1, l_2 \in L_p(T)$ such that $M(l_1, l_2) = M(l_2, l_1) = 1$. If yes then the rule can be applied to l_1 and l_2. The total time requirement is $\mathcal{O}(|V|^2)$.

Rule 14: For each pair of leaves $l_1, l_2 \in L_p(T)$ we check in constant time the followings: $b(l_1) = b(l_2)$, $d_T(b(l_1)) = 3$ and $(b^-(l_1), b^-(l_2)) \in E(G)$. The total time requirement is $\mathcal{O}(|V|^2)$.

As a result we can conclude that the precondition of every single rule can be checked in $\mathcal{O}(|V|^2)$ time. Thus, to establish the time requirement of $\mathcal{O}(|V|^4)$ for Algorithm LOST, it is enough to see that at most $\mathcal{O}(|V|^2)$ rules are applied before the algorithm stops and a LOST is found. Clearly, all of Rules 1–12 decrease the number of leaves as a result of their application. Therefore, the total number of their executions is $\mathcal{O}(|V|)$. Rules 13 and 14 do not change the number of leaves. They strictly decrease, however, the sum $\sum_{l \in L_p} |br(l)|$, that is the total length of branches having no leafish vertices. Without changing the number of leaves, this sum can be decreased $\mathcal{O}(|V|)$ times. (It can happen that the application of one of Rules 1–12 increases this sum, but in this case the number of leaves is decreased.) We conclude that there can be $\mathcal{O}(|V|^2)$ rule execution steps and so the running time of Algorithm LOST is $\mathcal{O}(|V|^4)$.

5.2. Algorithm WLOST and Algorithm RWLOST

The precondition part of each individual rule can be tested as follows.

Rule 15: This rule is the same as Rule 2 of Algorithm LOST. Checking its precondition needs $\mathcal{O}(|V|^2)$ time.

Rule 16: For every leaf l and for every non-tree edge (l, x) we check whether $c(x^{\to l}) < c(l)$. This requires $\mathcal{O}(|V|^2)$ time.

Rule 17: For every leaf l we check whether $c(b^-(l)) < c(l)$ and we look for a non-tree edge (l, x) such that $x \notin br(l)$. This needs $\mathcal{O}(|V|^2)$ time.

Rule 18: For every short leaf l_1 and for every leaf l_2 we check whether $(l_1, l_2) \in E(G)$. This requires $\mathcal{O}(|V|^2)$ time.

Rule 19 (only for Algorithm RWLOST): For every short leaf l it is enough to check the local neighborhood of $b(l)$. This can be done in $\mathcal{O}(|V|^2)$ time.

As a result, the precondition part of each particular rule can be checked in $\mathcal{O}(|V|^2)$ time. It remains to show that there are at most $\mathcal{O}(|V|^2)$ rule application steps before the algorithms stop.

For this purpose, let the vertices of G be sorted to the descending order of their weights, that is $c(v_1) \geq c(v_2) \geq \cdots \geq c(v_n) \geq 0$. Let T be the initial spanning tree that we have before rule applications. Let us use $|L(T)|$ markers to select the leaves of T from $v_1, v_2, \ldots, v_n$. When an improvement rule is executed in the current spanning tree, we change the position of the appropriate markers such that they always point to the leaves. The application of Rules 15 and 18 turns a leaf l into an internal vertex, that is, the marker of l is completely removed and is not used anymore. The application of Rules 16 and 17 changes a leaf l_1 to an other leaf l_2 such that $c(l_2) < c(l_1)$. This results that the marker of l_1 is moved to a later element (l_2) of the sequence $v_1, v_2, \ldots, v_n$. Observe that such a way, every marker is moved at most n times. Thus, at most $\mathcal{O}(|V|^2)$ rule execution steps are enough to obtain a WLOST.

In Algorithm RWLOST, we alternate the application of Rules 15–19. We use the above method to mark the leaves of the current spanning tree. Rule 19 does not change the set of leaves, and so the position of markers. However, it decreases the number of short branches. It is easy to see that Rules 15 and 18 do not increase the number of short branches, while Rules 16 and 17 increase it by at most one. Putting these facts together we conclude that Rule 15 is applied at most $|L_s(T)| + r$ times, where r is the total number of applications of Rules 16 and 17, that is $r = \mathcal{O}(|V|^2)$. This proves that after $\mathcal{O}(|V|^2)$ improvement steps we obtain an RWLOST.

As a result, we conclude that both algorithms use $\mathcal{O}(|V|^2)$ time for a single rule execution, and thus run in $\mathcal{O}(|V|^4)$ time.

6. Concluding remarks

In this paper we gave an $\mathcal{O}(n^4)$-time 7/4-approximation algorithm for the Maximum Internal Spanning Tree problem. The algorithm is based on the successive application of local improvement steps. We have used a primal–dual linear programming technique to obtain our approximation ratio which improves the previously known best ratio of 2. Although, there is no constant factor approximation for the Minimum Leaf Spanning Tree problem, our algorithm can be used as a heuristic for this problem, too.

We also presented the weighted version of the problem where each vertex has a positive value associated with it, and the aim is to find a spanning tree whose internal vertices have a maximum possible total weight. For this problem we gave an $\mathcal{O}(|V|^4)$-time $(2\Delta-3)$-approximation. We further refined this algorithm to ensure an approximation ratio of 2 for claw-free graphs.

Remark that the analysis of our algorithms are not tight. Implementing the most important local improvement steps, we experienced that in most cases they are very efficient in finding a many-internal-vertex spanning tree. For most of our input graphs we obtained near-optimal solutions, much better ones than the theoretic approximation factors.

A natural question arising is whether our local improvement based algorithms or our linear programming based proofs can be applied on closely related yet unsolved degree-based spanning tree optimization problems. Although, some of these problems have already approximation algorithms their approximability properties are not yet fully discovered. We hope that investigating these properties will lead to a better understanding of spanning tree optimization problems and will facilitate in giving efficient algorithms and heuristics for the related network design applications.

Acknowledgments

The author would like to thank Gábor Wiener, Dániel Marx and an anonymous referee for their valuable comments that helped a lot to improve the presentation of this paper.

The author's research was partially supported by the Hungarian National Research Fund and by the National Office for Research and Technology (Grant Number OTKA 67651).

References

[1] B.Y. Wu, K.-M. Chao, Spanning Trees and Optimization Problems, Chapman & Hall/CRC, 2004.
[2] H.-I. Lu, R. Ravi, The power of local optimization: Approximation algorithms for maximum-leaf spanning tree (DRAFT), Tech. Rep. CS-96-05, Department of Computer Science, Brown University, Providence, Rhode Island, 1996.
[3] G. Salamon, G. Wiener, On finding spanning trees with few leaves, Information Processing Letters 105 (2008) 164–169.
[4] M.X. Goemans, D.P. Williamson, A general approximation technique for constrained forest problems, in: Proceedings of SODA: ACM-SIAM Symposium on Discrete Algorithms, 1992, pp. 307–316.
[5] M. Fürer, B. Raghavachari, Approximating the minimum degree spanning tree to within one from the optimal degree, in: Proc. of the 3rd Annual ACM-SIAM Symp. on Discrete Algorithms, 1992, pp. 317–324.
[6] H.-I. Lu, R. Ravi, Approximation for maximum leaf spanning trees in almost linear time, Journal of Algorithms 29 (1) (1998) 132–141.
[7] R. Solis-Oba, 2-approximation algorithm for finding a spanning tree with maximum number of leaves, in: Proc. of 6th ESA Symposium, in: LNCS, vol. 1461, Springer, 1998, pp. 441–452.
[8] L. Gargano, P. Hell, L. Stacho, U. Vaccaro, Spanning trees with bounded number of branch vertices, in: Proc. of ICALP'02, in: LNCS, vol. 2380, Springer, 2002, pp. 355–365.
[9] G. Salamon, Spanning tree optimization problems with degree-based objective functions, in: The 4th Japanese–Hungarian Symposium on Discrete Mathematics and Its Applications, 2005, pp. 309–315.
[10] A. Schrijver, Shortest spanning trees, in: Combinatorial Optimization, Polyhedra and Efficiency. Vol. B, Springer-Verlag, 2003, pp. 855–876 (Chapter 50).
[11] H.A. Jung, On a class of posets and the corresponding comparability graphs, Journal of Combinatorial Theory Series B 24 (1978) 125–133.
[12] G. Salamon, G. Wiener, Leaves of spanning trees and vulnerability, in: The 5th Hungarian–Japanese Symposium on Discrete Mathematics and Its Applications, 2007, pp. 225–235.