Algorithmica
DOI 10.1007/s00453-015-0080-0

A 2-Approximation Algorithm for Finding a Spanning Tree with Maximum Number of Leaves

Roberto Solis-Oba[1] · **Paul Bonsma**[2] · **Stefanie Lowski**[3]

Received: 9 December 2013 / Accepted: 5 October 2015

Abstract We study the problem of finding a spanning tree with maximum number of leaves. We present a simple, linear time 2-approximation algorithm for this problem, improving on the previous best known algorithm for the problem, which has approximation ratio 3.

Keywords Approximation · Algorithm · Graphs · Optimization

1 Introduction

1.1 Results

In this paper, we study the problem of finding in a given connected graph a spanning tree with maximum number of leaves (MAX- LEAVES SPANNING TREE). This number is called the *max leaf number* of the graph. This problem has applications in the design of (wireless) communication networks [21,23,31], circuit layouts [34], and in distributed

An extended abstract of this paper appeared in the proceedings of ESA 1998 [32].

Roberto Solis-Oba: Research of this author partially supported by Grant 227829-2009 from the Natural Sciences and Engineering Research Council of Canada.

✉ Roberto Solis-Oba
solis@csd.uwo.ca

1 Department of Computer Science, The University of Western Ontario, London, ON N6A 5B7, Canada

2 Faculty of EEMCS, University of Twente, PO Box 217, 7500 AE Enschede, The Netherlands

3 Mathematics Department, Humboldt University Berlin, Unter den Linden 6, 10099 Berlin, Germany

Published online: 13 October 2015

systems [28]. In addition, for graphs with low max leaf number, many problems can be solved efficiently [15]. MAX- LEAVES SPANNING TREE is known to be APX-hard [18], even for cubic graphs [5], so it admits no polynomial time approximation scheme unless P = NP. In this paper, we present a 2-approximation algorithm for MAX-LEAVES SPANNING TREE, improving on the previous best approximation ratio of 3, achieved by algorithms of Lu and Ravi [26,27]. In addition, we improve the running time to linear. This yields our main result:

Theorem 1 *There exists a linear time approximation algorithm for* MAX- LEAVES SPANNING TREE *with approximation ratio 2.*

The algorithm we study here (in Sect. 2) is actually the algorithm presented by the first author in a conference paper from 1998 [32]. However, in [32] no complete proof of the approximation ratio is given. A more detailed, but also relatively long and complex proof appears in the unpublished technical report [33]. In this paper, we present a new, complete and relatively concise proof of the approximation ratio of this algorithm (Sect. 3), inspired by the proof ideas from [32]. In addition, we describe how the algorithm can be implemented to run in linear time (Sect. 4), and discuss tightness of the analysis (Sect. 5). Hence the current paper may be viewed as the full version of [32]. It is surprising that this is still the best known approximation algorithm for MAX- LEAVES SPANNING TREE, considering the amount of research that has been done on this problem, especially in the last 15 years; see below for a selection of the most relevant results.

1.2 Techniques

Lu and Ravi [26] presented the first constant-factor approximation algorithm for MAX-LEAVES SPANNING TREE on arbitrary graphs. Using local-improvement techniques they designed approximation algorithms with approximation ratios 5 and 3. Later [27] they introduced the concept of *leafy forest* that allowed them to design a more efficient 3-approximation algorithm for the problem. For the construction of a leafy forest, there are two conflicting objectives: informally, (1) the number of leaves of the forest should be at least a constant fraction of the max leaf number, and (2) the number of components k should be small, since connecting them into a spanning tree may cost up to $2k - 2$ leaves.

We improve the approximation ratio to 2 by using an algorithm based on *expansion rules*. This is similar to the techniques that have been used to obtain lower bounds for the max leaf number in e.g. [9,19,24]. However, we assign priorities to the rules, and use them to build a forest instead of a tree as in earlier papers [19,24]. The leafy forest F^* built by our rules satisfies the above objectives in a way that allows proving the approximation ratio.

Informally, the priority of a rule reflects the number of leaves that the rule adds to the forest F^*. Hence it is desirable to use only 'high' priority rules to build F^*. The 'low' priority rules are needed though to ensure that the number of components of F^* can be kept small enough. The key idea that allows us to prove the approximation ratio of 2 is an upper bound for the max leaf number that takes into account the number of times that 'low' priority rules need to be used to build the forest F^*.

1.3 Related Results

From an optimization viewpoint, the MAX- LEAVES SPANNING TREE problem is equivalent to the problem of finding a minimum connected dominating set (MIN CONNECTED DOMINATING SET): for a graph G on at least three vertices and any $L \subseteq V(G)$, G admits a spanning tree where all vertices in L are leaves if and only if $V(G)\backslash L$ is a connected dominating set of G. However, from an approximation viewpoint, the problems are very different. The best known approximation algorithm for MIN CONNECTED DOMINATING SET is given by Ruan et al. [30], with an approximation ratio of $2 + \ln \Delta(G)$, where $\Delta(G)$ denotes the maximum degree of G, improving the approximation ratio given in [21]. In addition, Guha and Khuller [21] showed that this ratio cannot be significantly improved, unless $\mathrm{NP} \subseteq \mathrm{DTIME}[n^{O(\log \log n)}]$.

Approximation algorithms for the restriction of MAX- LEAVES SPANNING TREE to cubic graphs have been well-studied [8,10,25]. The current best approximation ratio is 3/2 [8], which gives a 4/3-approximation algorithm for MIN CONNECTED DOMINATING SET on cubic graphs. For cubic graphs, these problems are still APX-hard [5]. A self-stabilizing 3-approximation algorithm for MAX- LEAVES SPANNING TREE has been presented in [23]. This is a certain type of distributed algorithm, which is very useful in wireless networks. The generalization of MAX- LEAVES SPANNING TREE to directed graphs has also been well-studied in recent years. In this variant, the objective is to find a spanning subgraph T that is an oriented tree, with all edges oriented away from a chosen root. The number of vertices of T without out-neighbors (leaves) should be maximized. The first approximation algorithm for this problem was given in [13], and the first constant factor approximation algorithm was given in [12], with an approximation ratio of 92. For the special case of directed acyclic graphs, this has been improved to 2 [31].

There are several papers that give tight lower bounds in terms of the number of vertices n for the max leaf number for certain graph classes, such as [9,19,20,24,34]. In particular, for graphs with minimum degree at least 3, the max leaf number is at least $n/4 + 2$ [19,24]. After excluding two special subgraphs, this can be improved to $n/3+4/3$ [9] (this results uses and generalizes the result for cubic graphs by Griggs et al. [19]). For minimum degree 4 and 5, the bound can be improved to $2n/5+8/5$ [20,24] and $n/2 + 2$ [20], respectively. These results easily give approximation algorithms with ratios 3, 5/2 and 2, but only for graphs with minimum degree at least 3, 4 and 5, respectively. Similar bounds for directed graphs appear in [7,12].

Lower bounds of this type have also been used to obtain fixed parameter tractable algorithms for the parameterized version of MAX- LEAVES SPANNING TREE (see e.g. [6,7]). This is a decision problem that asks, for a given connected graph G on n vertices and integer k, whether the max leaf number of G is at least k. When choosing k as parameter, an algorithm for this problem is *fixed parameter tractable* (*FPT*) if its complexity can be bounded by $f(k) \cdot \mathrm{poly}(n)$, where $\mathrm{poly}(n)$ denotes a polynomial function in n, and $f(k)$ an arbitrary computable function of k. The first FPT algorithm for the undirected version of this problem appears in [4], and for the directed version in [7] (extending the algorithm introduced in [1]). After a long sequence of improvements, the current best algorithms have complexity $O(3.4575^k)\mathrm{poly}(n)$ [29] and $O(3.72^k)\mathrm{poly}(n)$ [11], for the undirected and directed version, respectively.

Another way to obtain and improve both FPT and approximation algorithms is based on *kernelization*. Informally, the goal here is to give a preprocessing algorithm for the parameterized (decision) version of the problem that returns an equivalent instance (the kernel) on at most $f(k)$ vertices (the *kernel size*). The best known kernelization algorithm has kernel size $3.75k$ [14]. It can be verified that this result gives a 3.75-approximation algorithm for MAX- LEAVES SPANNING TREE. For the vertex-weighted version, with vertex weights at least one, a kernelization algorithm with kernel size $5.5k$ was given in [22] (for this result, the parameter k is a lower bound for the sum of leaf *weights*). Although the directed version of the problem admits no polynomial kernel unless the polynomial hierarchy collapses to the third level [3], the version of the problem where a root is specified does [3]. The current best kernel for this rooted directed version has kernel size $O(k^2)$ [12].

The best known exact algorithms for undirected and directed MAX- LEAVES SPANNING TREE have running times $O(1.8966^n)$ [16] and $O(1.9044^n)$ [2], respectively. This latter can be improved to $O(1.8139^n)$ if exponential space is allowed [2]. Integer linear programming formulations of the problem have been studied in e.g. [17].

2 Algorithm and Lower Bound

All graphs we consider are simple and undirected. A *leaf* of a graph G is a vertex of degree 1. By $L(G)$ we denote the set of leaves of G. For a vertex v of a graph G, by $N_G(v)$ we denote the neighborhood of v in G, and by $d_G(v) = |N_G(v)|$ the degree of v. For a vertex set $S \subseteq V(G)$, we denote $\overline{S} = V(G) \backslash S$. A *path from u to v* in a graph G is a sequence $x_0, \ldots, x_n$ of distinct vertices such that $u = x_0$, $v = x_n$, and for all i, $x_i x_{i+1} \in E(G)$.

Let F be a (possibly empty) subgraph of a graph G. The operation of *expanding a vertex $v \in V(G)$* consists of

- adding v to F, in case $v \notin V(F)$, and
- for every $w \in N_G(v) \backslash V(F)$: adding vertex w and edge vw to F.

This operation yields a new subgraph F' of G, which is a supergraph of F. If F is a forest, then F' is also a forest. If $v \in V(F)$, then F and F' have an equal number of components, otherwise F' has one more component.

Our algorithm takes as input a connected, undirected graph G, with maximum degree at least 3. (Note that the case where G has maximum degree at most 2 can trivially be solved optimally.) In the *first phase* of the algorithm, a forest F^* is constructed by starting with the empty subgraph and iteratively expanding this using expansion operations of the types specified below until no such operation can be applied anymore, while maintaining a forest subgraph F of G throughout. In the *second phase* of the algorithm, first a spanning forest subgraph of G is obtained by starting with $F = F^*$ and applying arbitrary expansion operations to vertices of F as long as possible. Next, arbitrary edges are added, without introducing cycles, until a connected spanning subgraph T^* of G is obtained, which is therefore a spanning tree. This spanning tree T^* is returned by the algorithm. The algorithm is summarized in Algorithm 1.

We now focus only on the first phase of the algorithm, where the forest F^* is constructed by iteratively expanding a forest F. From the definition of expansion

Algorithm 1 A 2-approximation algorithm for MAX- LEAVES SPANNING TREE

INPUT: A connected simple undirected graph G with maximum degree at least 3.
OUTPUT: A spanning tree T^* of G.

// First phase:
$F :=$ the empty graph.
while one of the Expansion Rules 1–4 can be applied to F **do**
    Apply the expansion rule with the highest priority to F.
$F^* := F$
// Second phase:
while F is not spanning **do**
    Choose a vertex $v \in V(F)$ with $N_G(v) \backslash V(F) \neq \emptyset$, and expand v.
while F is not connected **do**
    Choose an edge of G between different components of F, and add it to F.
$T^* := F$
Return T^*

operation, it follows that at any time the only vertices in F that can have neighbors in $\overline{V(F)}$ are leaves of F (i.e. unexpanded vertices). We will use this observation implicitly throughout. We use the following four expansion rules, where the given order defines the priorities. That is, for any $i < j$, if Rule i can be applied, then Rule j may not be applied (see also Fig. 1).

Rule 1: If F contains a vertex v with at least two neighbors in $\overline{V(F)}$, then expand v.
Rule 2: If F contains a vertex v with only one neighbor w in $\overline{V(F)}$, which in turn has at least three neighbors in $\overline{V(F)}$, then first expand v, and next expand w.
Rule 3: If F contains a vertex v with only one neighbor w in $\overline{V(F)}$, which in turn has two neighbors in $\overline{V(F)}$, then first expand v, and next expand w.
Rule 4: If $\overline{V(F)}$ contains a vertex v with at least three neighbors in $\overline{V(F)}$, then expand v.

Note that since G has maximum degree at least 3, Rule 4 can be applied at least once, and thus F^* is nonempty. Rule 4 is the only expansion rule that increases the number of components of F. Furthermore, the priorities ensure that when Rule 4 is applied, Rules 1, 2, and 3 cannot be applied to any component of F. Applying further expansion

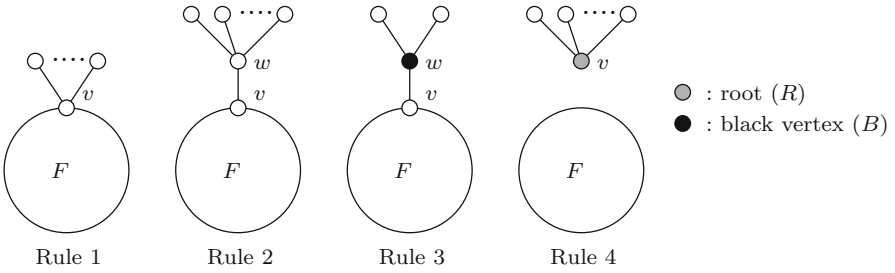

Fig. 1 The four expansion rules. *Ellipses* indicate possible additional neighbors

rules cannot change this, so once a new component is introduced, existing components of F will not be modified anymore.

Denote by $T_0, \ldots, T_k$ the $k+1$ (tree) components of F^*, in the order in which they are constructed. So for any $i \in \{0, \ldots, k\}$, the graph consisting of the components $T_0, \ldots, T_i$ occurs as graph F at some point during the construction of F^* (we will call this *the graph F as it is after the construction of T_i*), and Rules 1, 2 and 3 cannot be applied to this graph. We will often use this fact in the proofs below.

Whenever Rule 3 is applied, the vertex w referred to in this rule's definition is called a *black vertex*. By $B \subset V(F^*)$ we denote the set of all black vertices, so $|B|$ is the number of times Rule 3 has been applied during the construction of F^*. Whenever Rule 4 is applied, the vertex v referred to in this rule is called a *root*. By $R \subseteq V(F^*)$ we denote the set of roots; these are numbered so $R = \{r_0, \ldots, r_k\}$, and r_i is the root of component T_i. Figure 2a shows an example of a forest F^* constructed by Algorithm 1, with two components T_0 and T_1. First, T_0 is constructed by applying Rules 4 and 2. Next, T_1 is constructed, by applying Rules 4, 3, 1 and 3, in this order. So F^* has two black vertices, and two roots (which are all shown in black in the figure).

We can now give a simple lower bound for the number of leaves $|L(T^*)|$ of any spanning tree T^* returned by the algorithm. (The proof technique is based on [19,24].)

Lemma 2 *Let F^* and T^* be the forest and spanning tree of G constructed by* Algorithm 1*, respectively. Let $B \subseteq V(F^*)$ be the set of black vertices, and let $k+1$ be the number of components of F^*. Then*

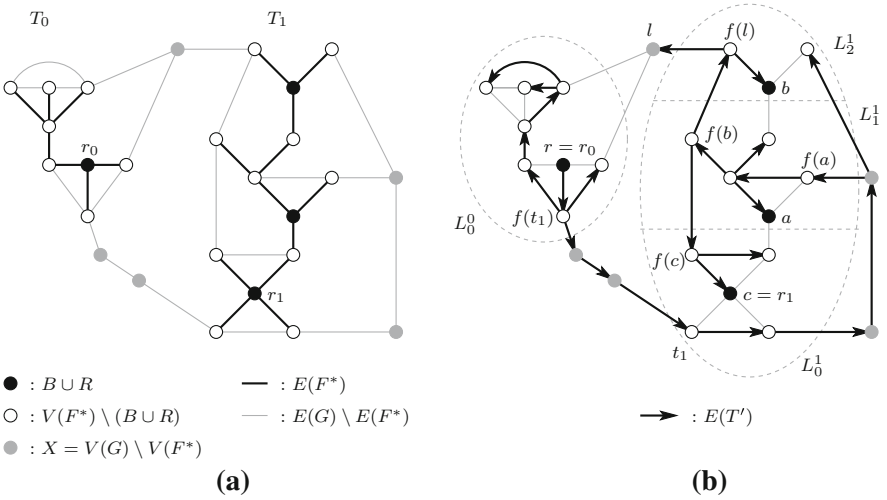

Fig. 2 **a** An example of a forest F^* constructed by Algorithm 1, consisting of two components T_0 and T_1, with roots r_0 and r_1. **b** An illustration of the proof of Lemma 9: *bold arrows* shows an arbitrary spanning tree T', rooted at $r = r_0$. This defines an injective mapping f from $S_1 = \{t_1\}$, $S_2 = \{l\}$ and $S_3 = \{a, b, c\}$ to $V(F^*) \setminus L(T')$

$$|L(T^*)| \geq \frac{|V(F^*)| - |B|}{2} - k + 1$$

Proof For any forest subgraph F of G constructed during the first phase of the algorithm, denote by $B(F)$ the current set of black vertices, by $\mathrm{cc}(F)$ the number of components, and define the *potential* of F as

$$\mathcal{P}(F) = 2|L(F)| - |V(F)| + |B(F)| - 2\mathrm{cc}(F).$$

We first argue that at any point during Phase 1 of the algorithm, it holds that $\mathcal{P}(F) \geq 0$. Clearly, $\mathcal{P}(F) \geq 0$ holds if F is the empty graph. Now suppose an expansion rule is applied to a forest F, which yields the forest F'. Denote $\Delta\mathcal{P} = \mathcal{P}(F') - \mathcal{P}(F)$. It suffices to show that $\Delta\mathcal{P} \geq 0$.

If Rule 4 is applied to a vertex v with $x \geq 3$ neighbors in $\overline{V(F)}$, then $|L(F)|$ increases by x, $|V(F)|$ increases by $x+1$, $|B(F)|$ is unchanged, and $\mathrm{cc}(F)$ increases by one. Since $x \geq 3$, $\Delta\mathcal{P} = 2x - (x+1) - 2 = x - 3 \geq 0$. If Rule 1 is applied to a vertex v with $x \geq 2$ neighbors in $\overline{V(F)}$, then $|L(F)|$ increases by $x-1$ (note that v will not be a leaf anymore), and $|V(F)|$ increases by x. The other values remain unchanged. Since $x \geq 2$, it holds that $\Delta\mathcal{P} = 2(x-1) - x = x - 2 \geq 0$. If Rule 2 is applied, expanding vertices v and w, where w has $x \geq 3$ neighbours in $\overline{V(F)}$ then $|L(F)|$ increases by $x-1$, and $|V(F)|$ increases by $x+1$. The other values remain unchanged. Since $x \geq 3$, it holds that $\Delta\mathcal{P} = 2(x-1) - (x+1) = x - 3 \geq 0$. Finally, if Rule 3 is applied, then $|L(F)|$ increases by 1, $|V(F)|$ increases by 3, and $|B(F)|$ increases by 1. So $\Delta\mathcal{P} = 2 - 3 + 1 = 0$.

Inductively, this shows that $\mathcal{P}(F^*) \geq 0$. Because $\mathrm{cc}(F^*) = k+1$ and $B(F^*) = B$, this yields $2|L(F^*)| - |V(F^*)| + |B| - 2(k+1) \geq 0$, hence

$$|L(F^*)| \geq \frac{|V(F^*)| - |B|}{2} + (k+1).$$

In the second phase of the algorithm, using arbitrary expansion operations to turn F^* into a spanning forest does not decrease the number of leaves. To turn the spanning forest into a spanning tree, only k edges need to be added. This process can destroy at most $2k$ leaves, so for the resulting spanning tree T^* we have:

$$|L(T^*)| \geq \frac{|V(F^*)| - |B|}{2} + (k+1) - 2k.$$

□

3 Upper Bound and Approximation Ratio

Throughout this section, F^* again denotes the forest subgraph of G constructed by Algorithm 1, with black vertex set B, and components $T_0, \ldots, T_k$, numbered in order of construction, which have roots $R = \{r_0, \ldots, r_k\}$. We denote $X = \overline{V(F^*)}$. Before we can prove our main upper bound for the max leaf number, we first need to prove

four lemmas characterizing the local structure of F^*. Essentially, these all state that the various types of vertices of G cannot have neighbors in too many different components of F^*, and cannot have too many neighbors in X.

Lemma 3 *Let u be a nonleaf vertex of F^* in component T_i. Then $N_G(u) \subseteq V(T_i)$.*

Proof Since u is a nonleaf, it has been expanded at some point during the construction of F^*, so $N_G(u) \subset V(F^*)$. Suppose u has a neighbor v in another component T_j, $j \neq i$. Then $j < i$, otherwise uv would have been added when expanding u. Since u is a nonleaf vertex of T_i, its degree in T_i is at least 2, so u has at least two neighbors in T_i which are not in the forest F as it is after the construction of T_j. So at this point, instead of using Rule 4 to introduce a new component, Rule 2 or 3 could have been applied to v; a contradiction. □

Lemma 4 *Let $u \in X$ with $d_G(u) \geq 3$. Then there is a component T_i of F^* that contains at least $d_G(u) - 1$ neighbors of u.*

Proof If u has no neighbors in F^*, then Rule 4 can be applied; a contradiction. So now we may choose i to be the lowest index such that $N_G(u) \cap V(T_i) \neq \emptyset$, and choose $v \in N_G(u) \cap V(T_i)$. If u has at least two neighbors not in T_i, then these neighbors are not in the forest F as it is after T_i has been constructed and so Rule 2 or 3 could have been applied to v; a contradiction. □

Lemma 5 *Let $u \in V(T_i)$ be a vertex with $|N_G(u) \setminus V(T_i)| \geq 2$. Then there is a tree T_j such that $N_G(u) \subseteq V(T_i) \cup V(T_j)$.*

Proof Because Rule 1 cannot be applied to u when the construction of T_i is completed, u has at least one neighbor in a tree component T_j with $j < i$. Choose j to be the minimum index j with this property, and choose $v \in N_G(u) \cap V(T_j)$. Choose x to be a neighbor of u in T_i. If u has at least one neighbor w that is neither in T_j nor in T_i, then by choice of j, for the forest F as it is after the construction of T_j, it holds that $\{u, w, x\} \cap V(F) = \emptyset$. So Rule 2 or 3 can be applied to v at this point; a contradiction. Therefore, all neighbors of u are either in T_i or in T_j. □

Lemma 6 *Let $x_0, \ldots, x_p$ and $y_0, \ldots, y_q$ be two paths in G with*

- $x_0 = y_0 \in V(F^*)$,
- $x_i \notin V(F^*)$ *for* $1 \leq i \leq p - 1$,
- $y_i \notin V(F^*)$ *for* $1 \leq i \leq q - 1$, *and*
- $x_j \neq y_j$ *for some* j.

Then two of the vertices x_0, x_p and y_q are part of the same component of F^*.

Proof Let j be the lowest index such that $x_j \neq y_j$. If $j = 1$, then by Lemma 5 either

- x_0 and x_1 are in the same component of F^*; note that then $x_1 = x_p$,
- x_0 and y_1 are in the same component of F^*; in this case $y_1 = y_q$,
- x_1 and y_1 are in the same component of F^*; thus $x_1 = x_p$ and $y_1 = y_q$.

So now we may assume $j \geq 2$. Then $x_{j-1} = y_{j-1}$ is a vertex in X with degree at least 3; it has distinct neighbors x_{j-2}, x_j and y_j. By Lemma 4, at least two neighbors of x_{j-1} are part of the same component of F^*:

– if x_j and y_j are in the same component of F^*, then $x_j = x_p$ and $y_j = y_q$,
– if x_{j-2} and x_j are part of the same component of F^*, then $x_{j-2} = x_0$ and $x_j = x_p$,
– if x_{j-2} and y_j are in the same component of F^*, then $x_{j-2} = x_0$ and $y_j = y_q$. □

We remark that from this point onward, our proof is very different from the one in [32], until we obtain the same approximation ratio in Theorem 11. For the next lemma, we partition the vertices of every tree component T_i of F into *layers* $L_0^i, \ldots, L_p^i$, where $p = |B \cap V(T_i)|$. The first layer L_0^i contains all vertices that are added to T_i before the first black vertex is added to it, if any (i.e. before the first application of Rule 3). For $j \geq 1$, the layer L_j^i contains the vertices in T_i that are not in $L_0^i, \ldots, L_{j-1}^i$, which are added before the $(j+1)$th black vertex is added to T_i. (Figure 2b shows the layers for the forest F^* shown in Fig. 2a. Here, T_0 has only one layer, and T_1 has three layers.) For two vertices $u \in L_j^i$ and $v \in L_\ell^i$, we say that u lies in a *lower layer* than v (or v lies in a *higher layer* than u) if $j < \ell$. Note that this expression implies that u and v are part of the same component T_i of F^*. Observe that every layer of T_i either contains exactly one black vertex, or contains the root r_i of T_i. The following observation follows easily, since Rule 1 is always preferred above Rule 3.

Proposition 7 *No vertex in F^* has more than one neighbor in a higher layer.*

Lemma 8 *Let $u \in V(T_i)$ have at least one neighbor not in T_i. Then u has no neighbor in a higher layer.*

Proof Let $w \in N_G(u) \backslash V(T_i)$, and suppose that u has at least one neighbor v in a higher layer (so $v \in V(T_i)$). Considering Rule 1, which has priority over the Rule 3 that introduces a new layer, we conclude that $w \in V(T_j)$, for some $j < i$. But u also has at least one neighbor x in the same layer (since for every expansion rule, every vertex added has at least one neighbor in the same layer). So for the forest F as it is after the construction of T_j, $\{u, v, x\} \cap V(F) = \emptyset$. This shows that at that point, Rule 2 or 3 can be applied, a contradiction. □

We can now prove the main lemma of this section. In Fig. 2b, the following proof is illustrated.

Lemma 9 *Let T' be a spanning tree of G, and let F^* be the forest constructed by Algorithm 1, with $k+1$ components, black vertex set B, and $X = \overline{V(F^*)}$. Then*

$$|V(F^*) \backslash L\left(T'\right)| \geq 2k + |X \cap L\left(T'\right)| + |B| - 1$$

Proof We will view T' as a rooted tree, with root $r = r_0$ (the root of T_0). For a vertex $v \in V(G) \backslash \{r\}$, the unique vertex $w \in N_{T'}(v)$ that lies on the path P from r to v in T' is called the *T'-parent* of v, and v is called a *T'-child* of w. The *T'-predecessors* of v are all vertices on this path P except for v itself. The *closest* T v-predecessor of v that satisfies some property is the one at minimum distance from v, where the distance is taken with respect to T'. A vertex $v \in V(T_i)$ is called an *in-portal* if its T'-parent is not part of T_i. It is called an *out-portal* if it has a T'-child outside of T_i. Together, in-portals and out-portals are called *portals*.

To prove the statement, we consider three disjoint vertex sets S_1, S_2 and S_3, with sizes k, $|X \cap L(T')|$ and $k + |B| - 1$ respectively, and construct an injective mapping $f : S_1 \cup S_2 \cup S_3 \to V(F^*) \backslash L(T')$. These sets are defined as follows:

- For every component T_i of F^*, except the component T_0 that contains the root r of T', we choose a vertex $t_i \in V(T_i)$ that has no T'-predecessors in $V(T_i)$. Then $S_1 = \{t_1, \ldots, t_k\}$.
- $S_2 = X \cap L(T')$.
- $S_3 = B \cup (R \backslash \{r\})$ (where r is the root of T' and of T_0).

(In the example in Fig. 2b, we have $S_1 = \{t_1\}$, $S_2 = \{l\}$, and $S_3 = \{a, b, c\}$.) Note that these sets are indeed disjoint; $S_1 \subseteq V(F^*)$ contains only portals, $S_3 \subseteq V(F^*)$ contains no portals by Lemma 3, and $S_2 \cap V(F^*) = \emptyset$. The injective mapping f is constructed as follows: For each $u \in S_1 \cup S_2$, let v be the closest T'-predecessor of u that is in $V(F^*)$. Set $f(u) = v$. (Note that since $r \in V(T_0)$, and S_1 contains no vertex in T_0, every vertex in $S_1 \cup S_2$ has indeed a T'-predecessor in $V(F^*)$.) For every vertex $b \in S_3$, let $w \in V(T_i)$ be the closest T'-predecessor of b that

- (Case A) is an in-portal, or
- (Case B) is in a lower layer, or
- (Case C) has its T'-parent in a higher layer.

Set $f(b) = w$. (In the example in Fig. 2b, Cases A, B and C apply to the vertices a, b and c, respectively. The mapping f is indicated.) Note that this mapping is well-defined: because $r \notin S_3$, every vertex in S_3 has a T'-predecessor in a different layer, or outside of its F^*-component. In addition, Lemma 3 shows that vertices in S_3 cannot be portals, so if a vertex in S_3 has a T'-predecessor outside of its F^*-component, then it has a T'-predecessor that is an in-portal (which lies in the same F^*-component). Because all images of f are T'-predecessors of some vertex, they are all nonleaves of T', and they are chosen in $V(F^*)$. Thus, to prove the lemma, it only remains to show that f is injective. Consider $u, v \in S_1 \cup S_2 \cup S_3$. We distinguish six cases.

Case 1: $\{u, v\} \subseteq S_1 \cup S_2$.
Let $x_0, \ldots, x_p$ be the T'-path from $f(u)$ to u, and let $y_0, \ldots, y_q$ be the T'-path from $f(v)$ to v. Suppose that $f(u) = f(v)$. Then Lemma 6 applies to these paths, so either u and v are part of the same component of F^*, or w.l.o.g. u and $f(u)$ are part of the same component of F^*. Neither case is possible by the definitions of S_1 and S_2.

Case 2: $u \in S_1 \cup S_2$, and $v \in S_3$.
By definition, $f(u)$ is an out-portal. If Case B or C applies to v, then $f(v)$ has a neighbor in a higher layer, so it cannot be a portal (Lemma 8). So we may assume that Case A applies, and therefore $f(v)$ is an in-portal. Suppose that $f(u) = f(v)$. Then $f(u)$ is both an in-portal and an out-portal, so it has at least two neighbors outside of its F^*-component T_i. By Lemma 5, all neighbors of $f(u)$ outside of T_i are part of the same F^*-component T_j. Note that then u cannot belong to S_2 (which contains no vertices of F^*), as $f(u)$ is the closest T'-predecessor of u in $V(F^*)$. So $u \in S_1$, and u is in fact one of these neighbors of $f(u)$ in T_j [a T'-child of $f(u)$]. However, the neighbor of $f(u)$ in T_j that is a T'-predecessor of $f(u)$ is then also a T'-predecessor of u, a contradiction with the definition of S_1.

In the remaining cases, w.l.o.g. it holds that $\{u, v\} \subseteq S_3$.

Case 3: Case A applies to u but not to v.
Then $f(u)$ is an in-portal, and $f(v)$ has a neighbor in a higher layer. By Lemma 8, $f(v)$ is therefore not an in-portal, so $f(u) \neq f(v)$.

Case 4: Case B applies neither to u nor to v.
Then $f(u)$ lies in the same layer as u, and $f(v)$ lies in the same layer as v. Since every layer contains at most one vertex of S_3, $f(u) \neq f(v)$.

Case 5: Case B applies to both u and v.
If $f(u) = f(v)$, then this vertex has two neighbors in a higher layer; one in the layer of u, and one in the layer of v. This is a contradiction with Proposition 7.

Case 6: Case C applies to u, and Case B applies to v.
If $f(u) = f(v)$, then this vertex has its T'-parent in a higher layer (because of Case C), and one T'-child in a higher layer (because of Case B). Hence it has two neighbors in a higher layer, a contradiction with Proposition 7.

W.l.o.g. this covers all cases. (Case 4 and 5 cover all cases except for those where Case B applies to one of the vertices but not to the other. Then if Case A applies to the other vertex, Case 3 can be applied. If Case C applies to the other vertex, Case 6 can be applied.) So we have proven that f is injective, which yields the lemma. □

Corollary 10 *Let T' be a spanning tree of G, and let F^* be the forest constructed by Algorithm 1, with $k+1$ components, and black vertex set B. Then*

$$|L\left(T'\right)| \leq |V(F^*)| - 2k - |B| + 1.$$

Proof Applying Lemma 9 yields

$$|L\left(T'\right)| = |X \cap L\left(T'\right)| + |V(F^*)| - |V(F^*) \backslash L\left(T'\right)| \leq |V(F^*)| - 2k - |B| + 1.$$

□

Now we can prove the approximation ratio of the algorithm.

Theorem 11 Algorithm 1 *is a 2-approximation algorithm for* Max- Leaves Spanning Tree.

Proof Let T^* be the spanning tree returned by the algorithm, and let T' be an optimal spanning tree. Combining Lemma 2 and Corollary 10 then yields

$$\frac{|L\left(T'\right)|}{|L(T^*)|} \leq \frac{|V(F^*)| - |B| - 2k + 1}{\left(|V(F^*)| - |B| - 2k + 2\right)/2} < 2.$$

□

4 Linear Time Implementation

We now describe a linear time implementation of the first phase of Algorithm 1. We may assume that the graph G is represented using adjacency lists. The main challenge is efficiently choosing the next vertex to expand. To this end, we maintain four lists R_i of expansion rule candidates; one list for every expansion rule. To be precise, at the beginning of every iteration of the first while-loop of Algorithm 1, we ensure that:

- R_1 contains all vertices in $v \in V(F)$ that have at least two neighbors in $\overline{V(F)}$.
- R_2 contains all vertices $w \in \overline{V(F)}$ that have at least one neighbor v in $V(F)$, and at least three neighbors in $\overline{V(F)}$.
- R_3 contains all vertices $w \in \overline{V(F)}$ that have at least one neighbor v in $V(F)$, and exactly two neighbors in $\overline{V(F)}$.
- R_4 contains all vertices $w \in \overline{V(F)}$ that have no neighbors in $V(F)$, and have $d_G(w) \geq 3$.

These are all implemented as doubly linked lists. For every vertex v, we keep track of the list it appears in (it appears in at most one of these lists), and maintain a pointer to its position in this list. This ensures that vertices can be added to and deleted from these lists in constant time. Note that R_2 and R_3 do not store the vertex $v \in V(F)$ that should be expanded first in the corresponding expansion rules, but instead store its (unique) neighbor $w \in N_G(v) \setminus V(F)$. This is no problem since we can simply expand w first, and add an edge vw for an arbitrary $v \in N_G(w) \cap V(F)$; this is equivalent to applying Rule 2 or 3 on v, respectively. To decide in constant time whether a vertex should be deleted from/added to one of these lists, we maintain for every vertex $v \in V(G)$ its *outside degree* $d^o(v) = |N_G(v) \setminus V(F)|$. Finally, we maintain a flag for every vertex v which allows us to test in constant time whether $v \in V(F)$. We now show that using these data structures, we can efficiently select the next expansion rule that should be applied (of highest priority), and apply this rule.

Lemma 12 *Consider one iteration of the while-loop in the first phase of Algorithm 1. Selecting the next vertex w to expand, applying the corresponding expansion rule, and updating all data structures, can be done in time $O(d_G(w) + \sum_{y \in N_G(w) \setminus V(F)} d_G(y))$.*

Proof In constant time, we can select a minimum i such that R_i is nonempty, and remove a vertex w from R_i. Expanding w (and in the case of Rule 2 or 3, adding an edge between w and an arbitrary neighbor $v \in V(F)$) can be done in time $O(d_G(w))$ using its adjacency list. This operation adds all vertices in $N_G(w) \setminus V(F)$ to F, and if $i \geq 2$, adds w to F. To correctly update $d^o(x)$ for all vertices $x \in V(G)$, we do the following: for every vertex y added to F and every $x \in N_G(y)$, we subtract 1 from $d^o(x)$. In total, updating the outside degrees this way takes time $O(d_G(w) + \sum_{y \in N_G(w) \setminus V(F)} d_G(y))$.

To ensure that all vertices are stored in the correct list R_i after this operation, we inspect all vertices x for which $d^o(x)$ has been modified and all vertices have been added to F. For every such vertex x, it can be verified that we can move x to the correct list, and remove it from its previous list, in constant time: Any vertex x that has been added to F has to be removed from R_2, R_3 or R_4, and added to R_1 if $d^o(x) \geq 2$; any vertex $x \notin V(F)$ for which $d^o(x)$ has been modified is first removed from its previous list, and subsequently added to R_2 if $d^o(x) \geq 3$, or to R_3 if $d^o(x) = 2$. (Vertices

x for which $d^o(x)$ has been modified have at least one neighbor in F, so we never need to add vertices to R_4.) In conclusion, updating the lists can again be done in time $O(d_G(w) + \sum_{y \in N_G(w) \setminus V(F)} d_G(y))$. (Recall that adding to and deleting from the lists can be done in constant time.) □

Now we can prove our main result.

Proof of Theorem 1 If G has maximum degree at most 2, then an optimal tree can trivially be found in linear time. Otherwise, we use Algorithm 1. Theorem 11 shows that Algorithm 1 is a 2-approximation algorithm for MAX- LEAVES SPANNING TREE, so it suffices to show that the algorithm has a linear time implementation. Let $n = |V(G)|$ and $m = |E(G)|$, where G is the input graph. The data structures described above can be initialized in linear time $O(n + m)$. Subsequently, one iteration of the while loop in the first phase of Algorithm 1 takes time $O(d_G(w) + \sum_{y \in N_G(w) \setminus V(F)} d_G(y))$, where w is the expanded vertex (Lemma 12). So all vertices in $y \in N_G(w) \setminus V(F)$ are added to F in this iteration. Since every vertex of G is added to F at most once, and expanded at most once, we may write

$$\sum_{w} \left(d_G(w) + \sum_{y \in N_G(w) \setminus V(F)} d_G(y) \right) \leq \sum_{v \in V(G)} 2 d_G(v) = 4m,$$

where the first summation is over all vertices w that are expanded in the first phase of Algorithm 1. Therefore, the first phase of Algorithm 1 terminates in linear time.

For the first while-loop in the second phase of Algorithm 1, a linear time implementation is analog (but easier). The last while loop can be implemented in linear time by translating it to the problem of finding a spanning tree, as follows. The previous parts of the algorithm can easily be extended to assign indices $g(v)$ to the the vertices $v \in V(G)$ such that $g(v) \in \{1, \ldots, k\}$, where k is the number of components of F^*, and $g(u) = g(v)$ if and only if u and v are in the same component of F^*. In linear time, all edges $uv \in E(G)$ with $g(u) \neq g(v)$ can be identified, and the problem can be translated to that of finding a spanning tree in an auxiliary (multi-)graph G' with $V(G') = \{1, \ldots, k\}$. Since $|V(G')| \leq n$ and $|E(G')| \leq m$, this problem can be solved in time $O(n + m)$ using e.g. breadth-first search. □

5 Discussion

Our main open question is whether the approximation ratio of two for MAX- LEAVES SPANNING TREE can be improved. For Algorithm 1, this ratio is asymptotically tight. To see this, for any positive integer k, define a graph G_k on $3k + 1$ vertices as shown in Fig. 3a. The graph G_k has an optimal spanning tree T_k with $2k - 1$ leaves; see Fig. 3b. However, there is a valid execution of Algorithm 1, starting with the expansion of r, that yields a spanning tree T_k^* with only $k+2$ leaves; see Fig. 3c. Hence $|L(T_k)|/|L(T_k^*)| \to 2$ as $k \to \infty$.

A related open question is whether the approximation ratio for MAX- LEAVES SPANNING TREE on directed graphs can be improved significantly. Currently, the best known ratio is 92 [12]. See also [31].

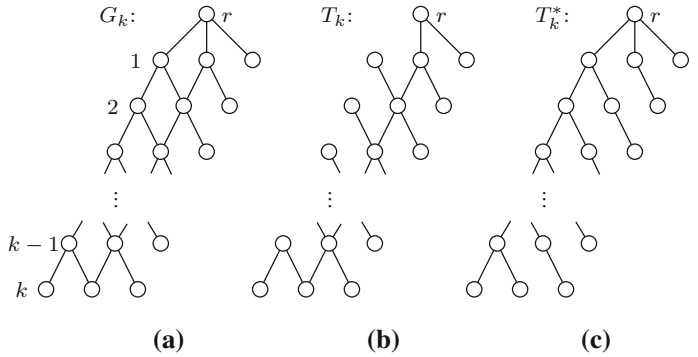

Fig. 3 An example showing that the approximation ratio 2 is tight for Algorithm 1. **a** The graph G_k, **b** an optimal spanning tree T_k, **c** the tree T^* returned by the algorithm

References

1. Alon, N., Fomin, F.V., Gutin, G., Krivelevich, M., Saurabh, S.: Spanning directed trees with many leaves. SIAM J. Discret. Math. **23**(1), 466–476 (2009)
2. Binkele-Raible, D., Fernau, H.: An exact exponential-time algorithm for the directed maximum leaf spanning tree problem. J. Discret. Algorithms **15**, 43–55 (2012)
3. Binkele-Raible, D., Fernau, H., Fomin, F.V., Lokshtanov, D., Saurabh, S., Villanger, Y.: Kernel(s) for problems with no kernel: on out-trees with many leaves. ACM Trans. Algorithms **8**(4) (2012) **(Article 38)**
4. Bodlaender, H.L.: On linear time minor tests with depth-first search. J. Algorithms **14**(1), 1–23 (1993)
5. Bonsma, P.: Max-leaves spanning tree is APX-hard for cubic graphs. J. Discret. Algorithms **12**, 14–23 (2012)
6. Bonsma, P., Brueggemann, T., Woeginger, G.J.: A faster FPT algorithm for finding spanning trees with many leaves. In: Mathematical Foundations of Computer Science (MFCS 2003). Volume 2747 of Lecture Notes in Computer Science, pp. 259–268. Springer, Berlin (2003)
7. Bonsma, P., Dorn, F.: Tight bounds and a fast FPT algorithm for directed max-leaf spanning tree. ACM Trans. Algorithms **7**(4) (2011) **(Article 44)**
8. Bonsma, P., Zickfeld, F.: A 3/2-approximation algorithm for finding spanning trees with many leaves in cubic graphs. SIAM J. Discret. Math. **25**(4), 1652–1666 (2011)
9. Bonsma, P., Zickfeld, F.: Improved bounds for spanning trees with many leaves. Discret. Math. **312**(6), 1178–1194 (2012)
10. Correa, J.R., Fernandes, C.G., Matamala, M., Wakabayashi, Y.: A 5/3-approximation for finding spanning trees with many leaves in cubic graphs. In: Approximation and Online Algorithms (WAOA 2007). Volume 4927 of Lecture Notes in Computer Science, pp. 184–192. Springer, Berlin (2008)
11. Daligault, J., Gutin, G., Kim, E.J., Yeo, A.: FPT algorithms and kernels for the directed k-leaf problem. J. Comput. Syst. Sci. **76**(2), 144–152 (2010)
12. Daligault, J., Thomassé, S.: On finding directed trees with many leaves. In: Parameterized and Exact Computation (IWPEC 2009). Volume 5917 of Lecture Notes in Computer Science, pp. 86–97. Springer, Berlin (2009)
13. Drescher, M., Vetta, A.: An approximation algorithm for the max leaf spanning arborescence problem. ACM Trans. Algorithms **6**(3) (2010) **(Article 46)**
14. Estivill-Castro, V., Fellows, M.R., Langston, M.A., Rosamond, F.A.: FPT is P-time extremal structure I. In: Algorithms and Complexity in Durham (ACiD 2005). Volume 4 of Texts in Algorithmics, pp. 1–41. King's College, London (2005)
15. Fellows, M., Lokshtanov, D., Misra, N., Mnich, M., Rosamond, F., Saurabh, S.: The complexity ecology of parameters: an illustration using bounded max leaf number. Theory Comput. Syst. **45**(4), 822–848 (2009)

16. Fernau, H., Kneis, J., Kratsch, D., Langer, A., Liedloff, M., Raible, D., Rossmanith, P.: An exact algorithm for the maximum leaf spanning tree problem. Theor. Comput. Sci. **412**(45), 6290–6302 (2011)
17. Fujie, T.: The maximum-leaf spanning tree problem: formulations and facets. Networks **43**(4), 212–223 (2004)
18. Galbiati, G., Maffioli, F., Morzenti, A.: A short note on the approximability of the maximum leaves spanning tree problem. Inf. Process. Lett. **52**(1), 45–49 (1994)
19. Griggs, J.R., Kleitman, D.J., Shastri, A.: Spanning trees with many leaves in cubic graphs. J. Graph Theory **13**(6), 669–695 (1989)
20. Griggs, J.R., Wu, M.: Spanning trees in graphs of minimum degree 4 or 5. Discret. Math. **104**(2), 167–183 (1992)
21. Guha, S., Khuller, S.: Approximation algorithms for connected dominating sets. Algorithmica **20**(4), 374–387 (1998)
22. Jansen, B.M.P.: Kernelization for maximum leaf spanning tree with positive vertex weights. J. Graph Algorithms Appl. **16**(4), 811–846 (2012)
23. Kamei, S., Kakugawa, H., Devismes, S., Tixeuil, S.: A self-stabilizing 3-approximation for the maximum leaf spanning tree problem in arbitrary networks. J. Comb. Optim. **25**(3), 430–459 (2013)
24. Kleitman, D.J., West, D.B.: Spanning trees with many leaves. SIAM J. Discret. Math. **4**(1), 99–106 (1991)
25. Loryś, K., Zwoźniak, G.: Approximation algorithm for the maximum leaf spanning tree problem for cubic graphs. In: Algorithms-ESA 2002. Volume 2461 of Lecture Notes in Computer Science, pp. 686–697. Springer, Berlin (2002)
26. Lu, H., Ravi, R.: The power of local optimization: Approximation algorithms for maximum-leaf spanning tree. In: Proceedings of the Thirtieth Annual Allerton Conference on Communication, Control and Computing, pp. 533–542 (1992)
27. Lu, H., Ravi, R.: Approximating maximum leaf spanning trees in almost linear time. J. Algorithms **29**(1), 132–141 (1998)
28. Payan, C., Tchuente, M., Xuong, N.H.: Arbres avec un nombre maximum de sommets pendants. Discret. Math. **49**(3), 267–273 (1984)
29. Raible, D., Fernau, H.: An amortized search tree analysis for k-leaf spanning tree. In: SOFSEM 2010: Theory and Practice of Computer Science. Volume 5901 of Lecture Notes in Computer Science, pp. 672–684. Springer, Berlin (2010)
30. Ruan, L., Du, H., Jia, X., Wu, W., Li, Y., Ko, K.: A greedy approximation for minimum connected dominating sets. Theor. Comput. Sci. **329**(1–3), 325–330 (2004)
31. Schwartges, N., Spoerhase, J., Wolff, A.: Approximation algorithms for the maximum leaf spanning tree problem on acyclic digraphs. In: Approximation and Online Algorithms (WAOA 2011). Volume 7164 of Lecture Notes in Computer Science, pp. 77–88. Springer, Berlin (2012)
32. Solis-Oba, R.: 2-approximation algorithm for finding a spanning tree with maximum number of leaves. In: Algorithms-ESA 1998. Volume 1461 of Lecture Notes in Computer Science, pp. 441–452. Springer, Berlin (1998)
33. Solis-Oba, R.: 2-Approximation Algorithm for Finding a Spanning Tree with Maximum Number of Leaves. Technical report TR 98-1-010. Max Planck Institute for Computer Science, Saarbruecken (1998). http://domino.mpi-inf.mpg.de/internet/reports.nsf/NumberView/1998-1-010
34. Storer, J.A.: Constructing full spanning trees for cubic graphs. Inf. Process. Lett. **13**(1), 8–11 (1981)