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

Roberto Solis-Oba

Max Planck Institut für Informatik
Im Stadtwald, 66123 Saarbrücken, Germany
solis@mpi-sb.mpg.de

Abstract. We study the problem of finding a spanning tree with maximum number of leaves. We present a simple 2-approximation algorithm for the problem, improving on the approximation ratio of 3 achieved by the best previous algorithms. We also study the variant in which a given set of vertices must be leaves of the spanning tree, and we present a 5/2-approximation algorithm for this version of the problem.

1 Introduction

In this paper we study the problem of finding in a given graph a spanning tree with maximum number of leaves. This problem has applications in the design of communication networks [5], circuit layouts [11], and in distributed systems [10]. Galbiati et. al [3] have proven that the problem is MAX SNP-complete, and hence that there is no polynomial time approximation scheme for the problem unless P=NP. In this paper we present a 2-approximation algorithm for the problem, improving on the previous best performance ratio of 3 achieved by algorithms of Ravi and Lu [8,9].

We briefly review previous and related work to this problem. The problem of finding a spanning tree with maximum number of leaves is, from the point of view of optimization, equivalent to the problem of finding a minimum connected dominating set. But, the problems are very different when considering how well their solutions can be approximated. Khuller and Guha [5] gave an approximation preserving reduction from the set cover problem to the minimum connected dominating set. This result implies that the set cover problem cannot be approximated within ratio $(1 - o(1)) \ln n$, where n is the number of elements in the ground set, unless $NP \subseteq \mathrm{DTIME}(n^{\log \log n})$ [2]. However the solution to the problem of finding a spanning tree with maximum number of leaves is known to be approximable within a constant of the optimum value [8,9].

There are several papers that deal with the question of determining the largest value ℓ_k such that every connected graph with minimum degree k has a spanning tree with at least ℓ_k leaves [1,4,7,11]. Kleitman and West [7], Storer [11], and Griggs et al. [4] show that every connected graph with n vertices and minimum degree $k = 3$ has a spanning tree with at least $n/4 + 2$ leaves. For $k = 4$ Kleitman and West [7] give a bound of $(2n + 8)/5$ for the value of ℓ_k,

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 441–452, 1998.

and for arbitrary k they give a lower bound of $(1 - \Omega(\ln k/k))n$ for the number of leaves. This bound was improved by Duckworth et al. [1] to $\frac{k-5}{k+1}2^k + 2$ for the special case of a hypercube of dimension k. All these algorithms can be used to approximate the solution to the problem of finding a spanning tree with maximum number of leaves, but only for graphs with minimum degree k, $k \geq 3$.

Ravi and Lu [8] presented the first constant-factor approximation algorithm for the problem on arbitrary graphs. Using local-improvement techniques they designed approximation algorithms with performance ratios of 5 and 3. Later [9] they introduced the concept of *leafy forest* that allowed them to design a more efficient 3-approximation algorithm for the problem. A leafy forest has two nice properties: (1) it can be completed into a spanning tree by converting a small number of leaves of the forest into internal vertices of the tree, and (2) the number of leaves in an optimal tree can be upper bounded in terms of the number of vertices in the forest.

We improve on the algorithms by Ravi and Lu by providing a linear time algorithm that finds a spanning tree with at least half of the number of leaves in any spanning tree of a given undirected graph. Our algorithm uses *expansion rules*, to be defined later, similar to those in [7]. However we assign priorities to the rules and use them to build a forest instead of a tree as in [7]. Incidentally, the forest F that our rules build is a leafy forest, so we can take advantage of its structure to build a spanning tree with a number of leaves close to that in the forest.

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 the forest. The "low" priority rules are needed, though, to ensure that only a bounded number of leaves become internal vertices when connecting the trees in F to form a spanning tree.

The key idea that allows us to prove the approximation ratio of 2 for the algorithm is an upper bound for the number of leaves in any spanning tree that takes into account the number of times that "low" priority rules need to be used to build the forest F.

We also consider the variant of the problem in which a given set S of vertices must be leaves and a spanning tree T_S with maximum number of leaves subject to this constraint is sought. By using the above algorithm we reduce this problem to a variant of the set covering problem in which instead of minimizing the size of a cover, we want to maximize the number of sets which do not belong to the cover. We present a simple heuristic for this latter problem which yields a (5/2)-approximation algorithm for finding the spanning tree T_S.

The rest of the paper is organized in the following way. In Sect. 2 we present our approximation algorithm. In Sect. 3 we prove a weaker bound of 3 for the performance ratio of the algorithm, and in Sect. 4 we strengthen the analysis to show the ratio of 2. Finally, in Sect. 5 we present a (5/2)-approximation algorithm for the version of the problem in which a given set of vertices must be leaves of the tree.

2 The Algorithm

Let $G = (V, E)$ be an undirected connected graph. We denote by m the number of edges and by n the number of vertices in G. Let T^* be a spanning tree of G with maximum number of leaves. In this section we present an algorithm that finds a spanning tree T of G with at least half of the number of leaves in T^*.

The algorithm first builds a forest F by using a sequence of *expansion rules*, to be defined shortly. Then the trees in F are linked together to form a spanning tree T. The expansion rules used to build F are designed so that a "large" fraction of the vertices in F are leaves and when T is formed, the smallest possible number of leaves from F are transformed into internal vertices. Hence the resulting spanning tree has "many" leaves. When forming the spanning tree T, we say that a leaf of F is *killed* when it becomes an internal vertex of T.

We now elaborate on how the forest F is constructed. Every tree T_i of F is built by first choosing a vertex of degree at least 3 as its root. Then the expansion rules described in Fig. 1 are used to grow the tree. These rules are applied to the leaves of the tree as follows. If a leaf x has at least two neighbors not in T_i then the rule shown in Fig. 1(b) is used which places all neighbors of x not belonging to T_i as its children. On the other hand, if x has only one neighbor y that does not belong to T_i and at least two neighbors of y are not in T_i, then the rule shown in Fig. 1(a) is used. This rule puts y as the only child of x and all the neighbors of y not in T_i are made children of y.

When a rule is applied to a vertex x we say that x is *expanded* by the rule. A tree T_i is grown until none of its leaves can be expanded.

We assign priorities to the expansion rules as follows. The rule shown in Fig. 2, namely a leaf x has a single neighbor y not in F and y has exactly two neighbors outside F, has priority 1. All other expansion rules have priority 2. In the rest of the paper we will refer to the rule of priority 1 simply as rule 1. The internal vertex y added with rule 1 (see Fig. 2) is called a *black vertex*.

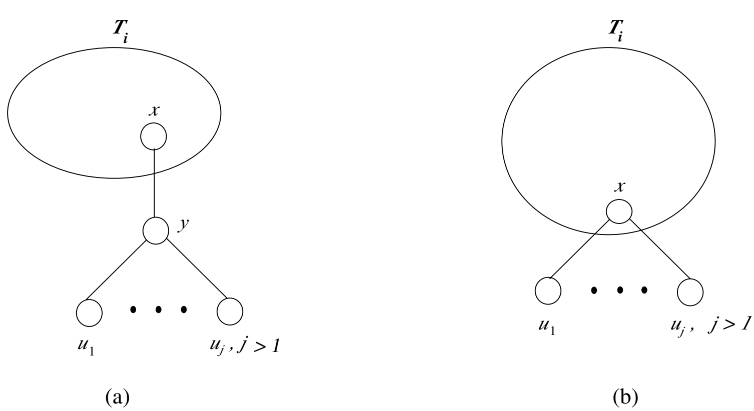

Fig. 1. Expansion rules.

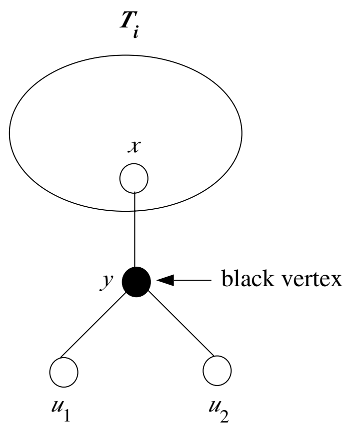

Fig. 2. Rule 1.

When building a tree $T_i \in F$, if two different leaves of T_i can be expanded, the leaf that can be expanded with the highest priority rule is expanded first. If two leaves can be expanded with rules of the same priority, then one is arbitrarily chosen for expansion. Our algorithm for finding a spanning tree T with many leaves is the following.

Algorithm *tree*(G)
$F \leftarrow \emptyset$
while there is a vertex v of degree at least 3 **do**
  Build a tree T_i with root v and leaves the neighbors of v.
  while at least one leaf of T_i can be expanded **do**
    Find a leaf of T_i that can be expanded with
    a rule of largest priority, and expand it.
  end while
  $F \leftarrow F \cup T_i$
  Remove from G all vertices in T_i and all edges incident to them.
end while
Connect the trees in F and all vertices not in F to form a spanning tree T.

3 Analysis of the Algorithm

In this section we show that the performance ratio of algorithm *tree* is at most 3, and in the next section we tighten the analysis to prove the performance ratio of 2. Given a graph $G = (V, E)$, let $F = \{T_0, T_1, \ldots, T_k\}$ be the forest built by our algorithm. For a tree $T_i \in F$, let $V(T_i)$ be the set of vertices spanned by it, and let $B(T_i)$ be the set of black vertices in T_i. The set of vertices spanned by F is $V(F)$ and the set of black vertices in F is denoted as $B(F)$. The set of leaves in a tree T is denoted as $\ell(T)$.

Let $X = V - V(F)$, be the set of vertices not spanned by F. We call X the set of *exterior* vertices. An exterior vertex cannot be adjacent to an internal vertex of some tree T_i because when a vertex x is expanded all its neighbors not in F are placed as its children.

We show that forest F has the property that any way of interconnecting the trees T_i to form a spanning tree T of G kills exactly $2k$ leaves. Moreover, exactly one leaf of F must be killed to attach every exterior vertex to T. This two facts allow us to bound the number of leaves in any spanning tree of G in terms of the number of vertices in F.

Lemma 1. *Let $G' = (V', E')$ be the graph formed by contracting every tree $T_i \in F$ to a single vertex and then removing multiple edges between pairs of vertices. Every exterior vertex has degree at most 2 in G'.*

Proof. The proof is by contradiction. Assume that there is an exterior vertex v that has degree at least 3 in G'. Consider 3 of the neighbors of v. Note that these 3 vertices cannot be exterior vertices because then algorithm *tree* would have chosen v as the root of a new tree. Hence, at least one neighbor of v is in F. Let T_i be the first tree built by our algorithm that contains one of the neighbors u of v. Since v is adjacent to two vertices not in T_i then algorithm *tree* would have expanded u using a rule of the form shown in Fig. 1(a), placing v as its child. □

Lemma 2. *Let u be a leaf of some tree $T_i \in F$. If u is adjacent to two vertices $v, w \notin T_i$, then v and w are leaves of the same tree $T_j \in F$.*

Proof. Clearly, v and w cannot be both exterior vertices. Also neither v nor w can be internal vertices of some tree T_j because if, say, v is an internal vertex of T_j then

1. if the algorithm builds tree T_j before T_i, then u would be placed as child of v in T_j, and
2. if T_i is built first, then v would have been placed as child of u. To see this observe that every internal vertex of a tree $T_j \in F$ has at least 3 neighbors in T_j. Thus, vertex u would have been expanded with a rule of the form shown in Figure 1(a) while building tree T_i.

Hence at least one of v, w must be a leaf in F. Let v be a leaf of some tree T_j and p be the parent of u in T_i. If w is not a leaf of T_j, then we can assume without loss of generality that when algorithm *tree* adds vertex v to T_j vertex w still does not belong to F. Note that then tree T_i cannot be built before T_j because our algorithm would have placed v and w as children of u. So let T_j be built before T_i.

Since vertices u, p, and w do not belong to F when T_j is built, then algorithm *tree* would expand v and place u as its child in T_j, which is a contradiction. □

Corollary 1. *Any spanning tree of G has at most $|V(F)| - 2k$ leaves.*

Proof. Let T' be a spanning tree of G and let F' be the forest induced by T' on $V(F)$. By Lemmas 1 and 2, any way of interconnecting the trees of F' to form a spanning tree must kill at least $2k$ leaves from F'. Also, to form a spanning tree of G, every exterior vertex not used to interconnect the trees in F' must be attached to a different leaf of F'. □

We now give a bound for the number of leaves in forest F and prove a performance ratio of 3 for the algorithm.

Lemma 3. *For any tree* $T_i \in F$, $|\ell(T_i)| \geq 3+|B(T_i)|+(|V(T_i)|-3|B(T_i)|-4)/2$.

Proof. The root of T_i is a vertex of degree at least 3, so it has at least 3 children. Also, every application of rule 1 adds two new leaves to T_i while killing one. Hence, by using $|B(T_i)|$ times rule 1 the number of leaves in T_i is increased by $|B(T_i)|$. All other vertices in T_i are added by rules of priority 2. It is not difficult to check that a rule of priority 2 increases the number of leaves in T_i by at least half of the number of vertices added to T_i by that rule. □

Lemma 4. *The performance ratio of algorithm tree is smaller than* 3.

Proof. Let T be the spanning tree built by our algorithm and let T^* be a spanning tree with maximum number of leaves. By Corollary 1 and Lemma 3,

$$\frac{\ell(T^*)}{\ell(T)} \leq \frac{|V(F)|-2k}{\sum_{i=0}^{k}(3+|B(T_i)|+(|V(T_i)|-3|B(T_i)|-4)/2)-2k}$$
$$\leq \frac{2(|V(F)|-2k)}{|V(F)|-|B(F)|-2k+2} \leq 2+\frac{2|B(F)|}{|V(F)|-|B(F)|-2k} .$$

Observe that $|V(F)| = \sum_{i=0}^{k}|V(T_i)| \geq \sum_{i=0}^{k}(4+3|B(T_i)|)$ because the root of a tree T_i has degree at least 3 and each application of rule 1 adds 3 vertices to the tree. So $|V(F)| > 4k+3|B(F)|$, and therefore, $\ell(T^*)/\ell(T) < 2+2|B(F)|/(2|B(F)|+2k) \leq 3$. □

Note that if $|B(F)| = 0$ then the proof of Lemma 3 would give a bound of 2 for the performance ratio of the algorithm. However, if $|B(F)| > 0$ then our analysis yields a bound of only 3. Intuitively this is because rule 1 adds three vertices to a tree, but it increases the number of leaves of the tree by only 1. So, only one third of the vertices added by rule 1 are leaves. To prove the bound of 2 for the performance ratio of our algorithm we will show that there must be at least one internal vertex in T^* for every black vertex in F.

4 Top, Exterior, and Black Vertices

Let $T_0, T_1, \ldots, T_k$ be the trees in the forest F, indexed in the order in which they are built by algorithm *tree*. The set of vertices $V(T_i)$ spanned by tree T_i is called a *cluster*. Fix a spanning tree T^* of G with maximum number of leaves.

We choose one of the internal vertices r of T^* as its root. For the rest of this section we will assume that T^* is a rooted tree.

To simplify the analysis we modify the optimal tree $T^*$and the tree T built by our algorithm as follows. Let $v_0, v_1, \ldots, v_i$ be a path in T^* such that $v_1, \ldots, v_i$ are exterior vertices, v_i is a leaf of T^*, and $v_1, \ldots, v_{i-1}$ have degree 2 in T^*. We remove vertices $v_1, \ldots, v_{i-1}$ and add edge (v_0, v_i). By doing this every exterior vertex which is a leaf in T^* is directly connected to a non-exterior vertex. Note that this change does not modify the number of leaves of T^* or T. Moreover, the forest F is not affected by this change. We call the resulting trees T^* and T.

We prove below a tighter upper bound than that given in Corollary 1 for the maximum number of leaves in a spanning tree of G, by showing that some vertices in $V(F)$ must be internal vertices in every spanning tree.

Consider a vertex x of some cluster $V(T_j)$ which does not contain the root r. Let p_{rx} be the path in T^* from r to x. Let x be the only vertex from $V(T_j)$ in p_{rx} and let y be the closest, non-exterior vertex, to x in p_{rx}. We say that y is a *top vertex.* The set of top vertices is a subset of the leaves of F which are killed when the trees T_i are interconnected to form T.

Given a vertex x of T^*, let T^*_x be the subtree of T^* rooted at x. We denote the set of top vertices in T^*_x as $P(T^*_x)$. Let $B_r(T^*_x)$ be the set formed by the black vertices in T^*_x and the vertices in T^*_x which are roots of trees in F. For every exterior vertex v which is a leaf in T^*_x, let $a(v)$ be the parent of v in T^*_x. Let $A(T^*_x)$ be the set formed by the parents $a(v)$ of all exterior vertices v which are leaves in T^*_x. Note that every vertex in $A(T^*_x)$ is a leaf of some tree in F.

Lemma 5. *In every subtree T^*_x of T^*, the sets $P(T^*_x), A(T^*_x)$, and $B_r(T^*_x)$ are disjoint.*

Proof. Here we only prove that $B_r(T^*_x) \cap A(T^*_x) = \emptyset$. The other proofs are similar. Let v be a vertex in $B_r(T^*_x)$. Either v is a black vertex or the root of some tree $T_i \in F$. Observe that when algorithm *tree* adds vertex v to T_i, all the neighbors of v which do not belong already to F are placed as children of v in T_i. Thus, v cannot be adjacent to an exterior vertex and so $v \notin A(T)$. □

We define the *deficit* of a subtree T^*_x, and denote it as *deficit*(T^*_x), as $|P(T^*_x) \cup A(T^*_x) \cup B_r(T^*_x)| - |I(T^*_x)|$, where $I(T^*_x)$ is the set of internal vertices in T^*_x. We prove below that the deficit of T^* is at most 1. This together with Lemma 5 shows that the number of leaves in T^* is at most $|V| - |A(T^*)| - |P(T^*)| - |B_r(T^*)| + 1 = |V(F)| - |P(T^*)| - |B_r(T^*)| + 1 < |V(F)| - |B(F)| - k - k$ because there are at least k top vertices in T^* and $|B_r(T^*)| = |B(F)| + k + 1$. This will immediately prove the following theorem.

Theorem 1. *Algorithm tree finds a spanning tree with at least half of the number of leaves in an optimal tree.*

Proof. By the proof of Lemma 4, the tree T built by our algorithm has $|\ell(T)| \geq (|V(F)| - |B(F)| - 2k)/2$. Hence by the above discussion,

$$\frac{\ell(T^*)}{\ell(T)} < \frac{|V(F)| - |B(F)| - 2k}{(|V(F)| - |B(F)| - 2k)/2} \leq 2 \ . \qquad \square$$

4.1 Bounding the Deficit of T^*

In this section we prove that the deficit of T^* is at most 1, this will complete the proof of Theorem 1. We say that a subtree T_x^* has *maximum deficit one* if $\mathit{deficit}(T_x^*) = 1$ and for every vertex v of T_x^* the deficit of the subtree T_v^* is at most one.

Lemma 6. *Let T_x^* be a subtree of T^* of maximum deficit one. Then at least one leaf of T_x^* belongs to $B_r(T^*)$.*

Proof. Since all vertices in $P(T_x^*)$ and $A(T_x^*)$ are internal vertices of T_x^*, then the only way in which T_x^* can have deficit 1 is if one of its leaves belongs to $B_r(T_x^*)$. □

Lemma 7. *No edge in G connects two vertices from $B_r(T^*)$.*

Proof. When algorithm *tree* adds a black vertex $u \in B_r(T^*)$ to some tree of F, all neighbors of u not in F are placed as its children. Since by definition the children of a vertex $u \in B_r(T^*)$ cannot belong to $B_r(T^*)$ the claim follows. □

Lemma 8. *If T_x^* is a subtree of maximum deficit one, then its root x is either a leaf or it has at least 2 children.*

Proof. The claim follows trivially if $x \in B_r(T^*)$ is a leaf of T^*. So we assume that x is an internal vertex of T^*. We consider 3 cases. (1) If $x \in A(T_x^*)$, then x is the parent of an exterior leaf u, and so the subtree T_u^* has deficit zero. Thus the only way in which T_x^* can have deficit 1 is if x has another child v and $\mathit{deficit}(T_v^*) = 1$. (2) If $x \notin B_r(T_x^*) \cup P(T_x^*) \cup A(T_x^*)$, then x must be the parent of at least 2 subtrees of deficit 1. (3) For the case when $x \in B_r(T_x^*)$ or $x \in P(T_x^*)$, we prove the lemma by showing that

(a) if $x \in B_r(T_x^*)$, then $\mathit{deficit}(T_x^*) < 1$, and
(b) if $x \in P(T_x^*)$, then x has at least two children u and v such that u is in the same cluster as x and $\mathit{deficit}(T_u^*) = 1$, and v is not in the same cluster as x and $\mathit{deficit}(T_v^*) < 1$.

We can prove claims (a) and (b) by induction on the number of vertices in T_x^*. Here we sketch the proof for claim (a) only. The basis of the induction is trivial. For the induction step, let us assume that $\mathit{deficit}(T_x^*) \geq 1$ and derive a contradiction from such assumption. Let u be a child of x and $\mathit{deficit}(T_u^*) = 1$. Note that u cannot be a leaf of T^* because then the only way in which $\mathit{deficit}(T_u^*)$ can be 1 is if $u \in B_r(T^*)$, but by Lemma 7 this cannot happen. Hence u is an internal vertex of T^* and by induction hypothesis all internal vertices v of T_u^* for which $\mathit{deficit}(T_v^*) = 1$ have at least two children.

To simplify the proof we modify the tree T_x^* as follows. For each internal vertex v of T_u^* such that $\mathit{deficit}(T_v^*) = 1$, remove all its children except two of them chosen as follows. Note that by induction hypothesis $v \notin B_r(T_x^*)$.

1. If $v \in V(T_x^*) - A(T_x^*) - P(T_x^*)$, keep 2 children that are roots of subtrees of deficit 1.
2. If $v \in A(T_x^*)$, keep the exterior vertex adjacent to v and one of its children that is the root of a subtree of deficit 1.
3. If $v \in P(T_x^*)$, then keep a vertex v_1 from the same cluster as v such that $\mathit{deficit}(T_{v_1}^*) = 1$ and a vertex v_2 not in the same cluster as v such that $\mathit{deficit}(T_{v_2}^*) < 1$. By induction hypothesis these vertices must exist. Also, remove all children of vertex v_2.

We denote the resulting tree as T_x^*. Note that these changes do not affect the value of $\mathit{deficit}(T_x^*)$.Every internal vertex of T_x^*, with the possible exception of x, has degree 3. Also, by Lemma 6, at least one leaf of T_x^* belongs to $B_r(T_x^*)$ and since $x \in B_r(T_x^*)$ then $|B_r(T_x^*)| \geq 2$. Consider the forest F built by our algorithm. Let w_1 and w_2 be, respectively, the first and second vertices from $B_r(T_x^*)$ that are added to forest F by algorithm *tree*. Note that w_1 and w_2 are not added at the same time to F.

Let S be the set of vertices from T_x^* which have been added to F by algorithm *tree* just before w_2 is included in some tree of F. Since w_1 is either a black vertex or the root of some tree $T_i \in F$, then all its neighbors must belong to S. Consider a longest path L in T_x^* starting at w_1 and going only through internal vertices of T_x^* that belong to S. Let y be the last vertex in L. Note that $y \neq x$ because otherwise $w_1 = x$ and so y would not be the last vertex in L since then (the internal vertex) u would also belong to S. Similarly, if w_1 is a leaf then its parent in T_x^* belongs to S and so $y \neq w_1$.

At least two of the neighbors of y in T_x^* must belong to S because otherwise y could be expanded by algorithm *tree* using a rule of priority 2 before w_2 is added to F, which by definition of S cannot happen. Let y_1 and y_2 be two neighbors of y in S. Clearly, both y_1 and y_2 cannot be internal vertices because otherwise L would not be a longest path as described above. Thus, let y_1 be a leaf of T_x^*. There are four cases that need to be considered.

1. $y \in V(T_x^*) - A(T_x^*) - P(T_x^*)$. Then $\mathit{deficit}(T_{y_1}^*) = 1$ and so $y_1 \in B_r(T_x^*)$ by Lemma 6. Since w_1 is the only vertex from $B_r(T_x^*)$ in S then $y_1 = w_1$. But by definition of y, vertex y_2 must also be a leaf and $\mathit{deficit}(T_{y_2}^*) = 1$. By Lemma 6, y_2 must belong to $B_r(T_x^*)$ which contradicts our assumption that w_1 was the only vertex from $B_r(T_x^*)$ in S.
2. $y \in B_r(T_x^*)$. This cannot happen since $y \neq w_1$ and we assumed that there is only one vertex form $B_r(T_x^*)$ in S.
3. $y \in A(T_x^*)$. Then either y_1 is an exterior vertex or $y_1 \in B_r(T_x^*)$. But y_1 cannot be an exterior vertex since $y_1 \in S$. Also y_1 cannot belong to $B_r(T_x^*)$ because if it does then $y_1 = w_1$ and y_2 would be an internal vertex of T^* contradicting our assumption for L.
4. $y \in P(T_x^*)$. We can derive a contradiction in this case also, but we omit the proof here since it is more complex than for the other cases.

The above arguments show that y cannot exist, and therefore if $x \in B_r(T^*)$ is an internal vertex of T^* then $\mathit{deficit}(T_x^*) < 1$. □

Lemma 9. *For all vertices x in T^*, deficit$(T^*_x) \leq 1$.*

Proof. The proof is by contradiction. Let x be an internal vertex of T^* such that T^*_x is a minimal tree of deficit larger than one, i.e., *deficit*$(T^*_x) > 1$ and for all vertices $v \neq x$ in T^*_x, *deficit*$(T^*_v) \leq 1$. By the proof of Lemma 8 we know that $x \notin B_r(T^*_x)$. Also by the same proof, if $x \in P(T^*_x)$ then x must have at least three children: two in the same cluster as x and one in a different cluster. We trim the tree T^*_x as described in the proof of Lemma 8 with the only exception that for vertex x we keep three children u, v, and w as follows.

1. If $x \in A(T^*_x)$, u is an exterior vertex, v and w are roots of trees of maximum deficit 1.
2. If $x \in P(T^*_x)$, u is a child in a cluster different from x, and v and w are children in the same cluster as x and which are roots of trees of maximum deficit one.
3. If $x \in V(T^*_x) - A(T^*_x) - P(T^*_x)$, u, v, and w are roots of trees of maximum deficit one.

Since x has at least two children which are roots of trees of deficit 1, then at least two leaves of T^*_x belong to $B_r(T^*_x)$. Let w_1, w_2, S, and L be as in the proof of Lemma 8. By using arguments similar to those used to prove Lemma 8 we can derive a contradiction, thus showing that *deficit*$(T^*_x) \leq 1$ for all vertices x. □

5 Fixing a Set of Leaves

Consider now that the vertices in some set $S \subset V$ are required to be leaves in a spanning tree of G, and the problem is to find a tree with maximum number of leaves subject to this constraint. In this section we present a 5/2-approximation algorithm for the problem.

It is easy to check if a graph G has a spanning tree in which a given set S of vertices are leaves. Two conditions are needed for such a spanning tree to exist. First, the graph obtained by removing from G all vertices in S and all edges incident to them must be connected. And second, every vertex in S must have at least one neighbor in $V - S$. For the rest of this section we will assume that there is at least one spanning tree having the vertices in S as leaves.

Without loss of generality we can assume that S forms an independent set of G, i.e. there are no edges having both endpoints in S. We can make this assumption since we are interested only in spanning trees in which the vertices of S are leaves and none of these trees includes an edge connecting two vertices from S.

Let $S_1 \subseteq S$ be the set formed by the vertices of degree 1 in S. Let G' be the graph obtained by removing from G the vertices in $S - S_1$ and all edges incident to them. Run algorithm *tree* on graph G' and let T' be the tree that it finds. Note that all vertices of S_1 are leaves in T'. If any vertex $v \in S - S_1$ is adjacent to an internal vertex u of T' then v is placed as child of u in T'. Let T' be the resulting tree. Let $S' \subseteq S$ be the set formed by the vertices in S which do not belong to T'. Note that the neighbors of vertices in S' are all leaves of T'.

We say that a subset C of leaves of T' *covers* the vertices in S' if every vertex in S' is adjacent to at least one vertex in C. Let C be a minimal subset of leaves of T' that covers S', i.e., for every vertex $u \in C$ there is at least one vertex $v \in S'$ such that u is the only neighbor of S' in C. To build a spanning tree for G, we place arbitrarily the vertices of S' as children of C.

Let $C_1 \subseteq C$ be the set of vertices in C with only one child and let S'_1 be the children of C_1. Since C is a minimal cover for S' then every vertex in S'_1 is adjacent to only one vertex in C. To see this assume that a vertex $u \in S'_1$ is adjacent to at least 2 vertices $v, w \in C$, where $v \in C_1$. But then, $C - \{v\}$ would also cover S, which cannot happen since C is a minimal cover. By the same argument, every vertex in S'_1 is adjacent to at least one vertex in $\ell(T') - C$.

Find a minimal set $C'_1 \subseteq \ell(T') - C$ that covers S'_1. Note that $C - C_1 \cup C'_1$ is a minimal cover for S'. If $|C'_1| < |C_1|$ then place the vertices in S'_1 as children of C' instead of as children of C_1. We let T be the spanning tree formed by this algorithm.

Lemma 10. $\ell(T) \geq \max\{|S'|, \ell(T'), \frac{2}{3}(|\ell(T')| + |S'|)\}$.

Proof. Trivially $\ell(T) \geq |S'|$ since all vertices in S' are leaves of T. Let C be the minimal cover for S' selected by our algorithm to attach the vertices of S' to the tree. Then $|C| \leq |S'|$, and so $\ell(T) = \ell(T') - |C| + |S| \geq \ell(T')$.

Let C_1 be the set formed by all vertices in C that have only one child, and S'_1 be the set of children of C_1. Note that $|S' - S'_1| \geq 2|C - C_1|$ since every vertex in $C - C_1$ has at least two children from S'. Also, since every vertex in S'_1 is adjacent to one vertex in C_1 and to at least one vertex in $\ell(T') - C$, then by of the way in which C was chosen, $|\ell(T') - C| \geq |C_1|$. Hence,

$$\begin{aligned} 3(|S' - S'_1| + |C_1| + |\ell(T') - C|) &\geq 2(|S' - S'_1| + |C_1| + |\ell(T') - C|) + \\ &\quad\ 2|C - C_1| + |C_1| + |C_1| \\ &= 2(|S'| - |S'_1| + |C_1| + |\ell(T')|) \\ &= 2(|S'| + |\ell(T')|), \text{ because } |S'_1| = |C_1| \ . \end{aligned}$$

Since $\ell(T) = |S' - S'_1| + |C_1| + |\ell(T') - C|$, then $\ell(T) \geq \frac{2}{3}(|S| + |\ell(T')|)$. □

Given a graph G and a subset of vertices S let T^* be a spanning tree of G with maximum number of leaves and in which all vertices from S are leaves. Our algorithm finds a tree T with at least (2/5)-times the number of leaves in T^*.

Theorem 2. $\ell(T^*)/\ell(T) \leq 5/2$.

Proof. Let $S' \subseteq S$ be as defined above and let G' be the graph obtained by removing from G all vertices in S' and all edges incident to them. Let T^+ be a spanning tree of G' with maximum number of leaves and such that all vertices in $S - S'$ are leaves. Let T' be as defined above, by Theorem 1, $\ell(T^+)/\ell(T') \leq 2$. Note that $\ell(T^*) \leq \ell(T^+) + |S'|$, hence by Lemma 10:

1. if $|S'| \le \ell(T')/2$, then $\ell(T^*)/\ell(T) \le (\ell(T^+) + |S'|)/\ell(T') \le 2 + \frac{1}{2} = \frac{5}{2}$,
2. if $|S'| \ge 2\ell(T')$, then $\ell(T^*)/\ell(T) \le (\ell(T^+) + |S'|)/|S'| \le \ell(T^+)/(2\ell(T')) + 1 = 2$,
3. if $\ell(T')/2 < |S'| < 2\ell(T')$, then $\ell(T^*)/\ell(T) \le \frac{3}{2}(\ell(T^+)+|S'|)/(\ell(T')+|S'|) \le \frac{3}{2} + \frac{3}{2}\ell(T')/(\ell(T') + |S'|) \le \frac{3}{2} + 1 = \frac{5}{2}$. □

References

1. Duckworth W., Dunne P.E., Gibbons A.M., and Zito M.: Leafy spanning trees in hypercubes. Technical Report CTAG-97008 (1997) University of Liverpool.
2. Feige U.: A threshold of $\ln n$ for approximating set-cover. Twenty Eighth ACM Symposium on Theory of Computing (1996) 314–318.
3. Galbiati G., Maffiol F., and Morzenti A.: A short note on the approximability of the maximum leaves spanning tree problem. Information Processing Letters **52** (1994) 45–49.
4. Griggs J.R., Kleitman D.J., and Shastri A.: Spanning trees with many leaves in cubic graphs. Journal of Graph Theory **13** (1989) 669–695.
5. Guha S. and Khuller S.: Approximation algorithms for connected dominating sets. Proceedings of the Fourth Annual European Symposium on Algorithms (1996) 179–193.
6. Guha S. and Khuller S.: Improved methods for approximating node weight Steiner trees and connected dominating sets. Technical Report CS-TR-3849 (1997) University of Maryland.
7. Kleitman D.J. and West D.B.: Spanning trees with many leaves. SIAM Journal on Discrete Mathematics **4** (1991) 99–106.
8. Lu H. and Ravi R.: The power of local optimization: approximation algorithms for maximum-leaf spanning tree. Proceedings of the Thirtieth Annual Allerton Conference on Communication, Control, and Computing (1992) 533–542.
9. Lu H. and Ravi R.: A near-linear time approximation algorithm for maximum-leaf spanning tree. Technical report CS-96-06 (1996) Brown University.
10. Payan C., Tchuente M., and Xuong N.H.: Arbres avec un nombre de maximum de sommets pendants. Discrete Mathematics **49** (1984) 267–273.
11. Storer J.A.: Constructing full spanning trees for cubic graphs. Information Processing Letters **13** (1981) 8–11.