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

Roberto Solis-Oba*

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 previous best performance ratio of 3 achieved by algorithms of Ravi and Lu. Our algorithm can be implemented to run in linear time using simple data structures. We also study the variant of the problem in which a given subset of vertices are required to be leaves in the tree. We provide a 5/2-approximation algorithm for this version of the problem.

1 Introduction

In this paper we study the problem of finding a spanning tree with maximum number of leaves in an undirected graph. 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 approximation 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. However, the problems are very different when considering how well their solutions can be approximated. Khuller and Guha [5] presented an approximation preserving reduction from the set-cover problem to the minimum connected dominating set, thus showing that the solution for this latter problem cannot be approximated within a constant factor of the optimal. 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].

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

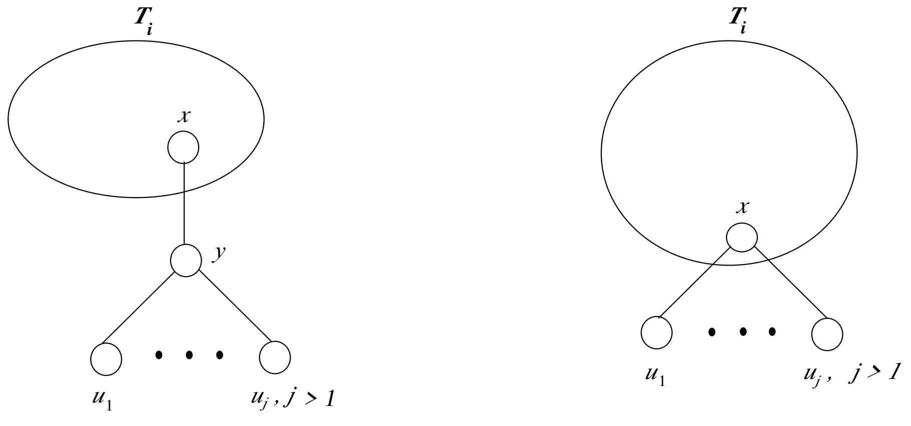
There are several papers that deal with the question of determining the largest value ℓ_k such that every connected graph with minimum degree k has a spanning tree with at least ℓ_k leaves [1, 4, 7, 11]. Kleitman and West [7], Storer [11], and Griggs et al. [4] showed that every connected graph with 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]proved that $\ell_k \geq (2n + 8)/5$, and for arbitrary k they give a lower bound of $(1 - \Omega(\ln k/k))n$ for the number of leaves. This bound was later 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. They used local-improvement heuristics that resulted in approximation algorithms with approximation ratios of 5 and 3. Later [9] they introduced the concept of *leafy forest* that allowed them to design a very efficient 3-approximation algorithm for the problem. A leafy forest has two nice properties that they exploit in this algorithm: (1) it can be completed into a spanning tree by changing 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 leaves 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 used by Kleitman and West [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 take advantage of its special structure to build a spanning tree with a number of leaves close to that in the forest.

Informally, the expansion rules of low priority used by our algorithm increase by a small amount the number of leaves in the forest, while the rules of high priority increase this number by a large amount. We are able to prove that the approximation ratio of the algorithm is 2 by showing that each rule of low priority adds to the forest at least one vertex that must be internal in any tree T^* with maximum number of leaves. Moreover, we show that this set of internal vertices is disjoint with the set of internal vertices required to interconnect the subtrees induced in T^* by the vertices spanned by F. This is enough to prove the bound of 2 for the approximation ratio of the algorithm. We give an example which shows that this bound is tight. By careful implementation, the algorithm can be made to run in linear time using simple data structures.

We also consider the variant of the problem in which a given set of vertices S 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



(a)

(b)

Figure 1: Expansion rules.

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 Section 2 we present our approximation algorithm for the problem of finding a spanning tree with maximum number of leaves. In Section 3 we prove a weaker bound of 3 for the approximation ratio of the algorithm, and in Section 4 we strengthen the analysis to show the ratio of 2. In Section 5 we describe a linear time implementation of the algorithm.In Section 6 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. When the trees in F are joined, some of the leaves of F become internal vertices in T. We say that a leaf of F is *killed* if it becomes an internal vertex of 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 killed.

Every tree T_i of the forest F is built by first choosing a vertex of degree at least 3 as its root. Then the expansion rules described in Figure 1 are used to grow the tree. These rules are applied to the leaves of the tree. If a leaf x has at least two neighbors not in T_i then the rule shown in Figure 1(b) is used and all the neighbors of x not belonging to T_i are placed as its

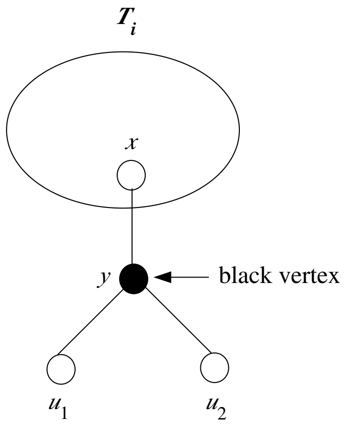


Figure 2: Rule 1.

children. 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 Figure 1(a) can be used. This rule puts y as the only child of x and all the neighbors of y not in T_i are placed as children of y. When a rule is applied to a vertex x we say that x is *expanded* by the rule.

We assign priorities to the expansion rules as follows. The rule shown in Figure 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. When building a tree, if two different leaves 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. A tree T_i is grown until no expansion rule can be applied to its leaves.

In the rest of the paper we will refer to the rule of priority 1 simply as rule 1. Rule 1 adds two new leaves to some tree T_i and at the same time it kills one leaf from T_i. The vertex y, parent of the two new leaves (see Figure 2), is called a *black vertex.*

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 the 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

Let $F = \{T_0, T_1, \ldots, T_k\}$ be the forest built by our algorithm, and let X be the set of vertices not spanned by F. We call X the set of *exterior* vertices. It is easy to see that an exterior vertex cannot be adjacent to an internal vertex of some tree T_i. Our expansion rules allow us to prove the following properties for the exterior vertices.

Lemma 3.1 *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. Assume that there is a vertex $v \in X$ 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 generated 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 Figure 1(a). □

Lemma 3.2 *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 vertex 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. Let 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. Then vertices u, p, and w are not yet in F when T_j is built. This means that algorithm *tree* would expand v and place u as its child in T_j. Hence v and w must be leaves of T_j. □

Corollary 3.1 *Any spanning tree of G has at most $|V(F)| - 2k$ leaves, where $V(F)$ is the set of vertices spanned by the forest F.*

Proof. Let T' be a spanning tree of G and let F' be the forest induced by T' on $V(F)$. By Lemmas 3.1 and 3.2, any way of interconnecting the trees in 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 show first that the approximation ratio of algorithm *tree* is at most 3, and in the next section we show how to tighten the analysis to prove the approximation ratio of 2. Given a graph G, let T^* be a spanning tree of G with maximum number of leaves. 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. Given a tree T let $\ell(T)$ be the set of its leaves. The set of black vertices in the forest F is denoted as $B(F)$.

Lemma 3.3 *For any tree $T_i \in F$, $|\ell(T_i)| \geq 3 + |B(T_i)| + \frac{|V(T_i)|-3|B(T_i)|-4}{2}$.*

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

Lemma 3.4 *The approximation ratio of algorithm tree is smaller than* 3.

Proof. Let T be the spanning tree built by our algorithm and T^* be a tree with maximum number of leaves. By Corollary 3.1 and Lemma 3.3,

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

Note 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 three vertices to the tree. So $|V(F)| > 4k + 3|B(F)|$, and therefore,

$$\frac{\ell(T^*)}{\ell(T)} < 2 + \frac{2|B(F)|}{2|B(F)|+2k} \leq 3.$$

□

Note that if $|B(F)| = 0$ then the proof of Lemma 3.3 would give a bound of 2 for the approximation 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

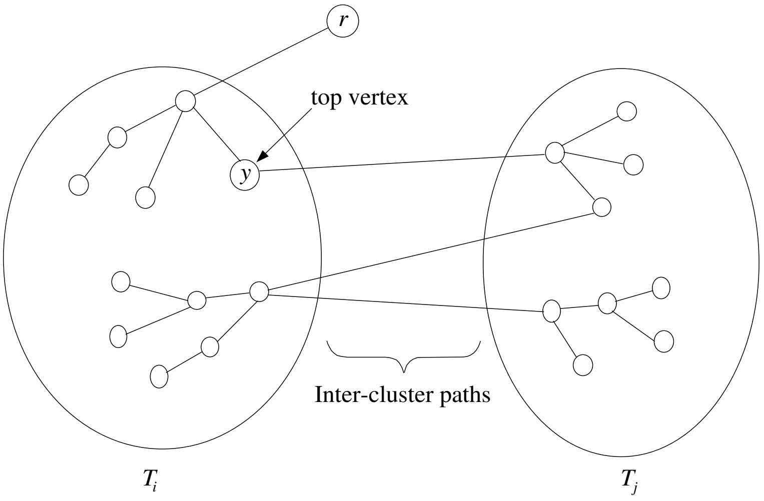


Figure 3: Vertex y is a top vertex.

leaves. To prove the bound of 2 for the approximation 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, with the subtrees 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. In the following discussion we will assume that T^* is a rooted tree.

We modify the optimal tree $T^*$and the tree T built by our algorithm by contracting into a single edge every path formed by exterior vertices of degree 2 in T^*. 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.

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 $y \in V(T_i)$ be the closest, non-exterior vertex, to x in p_{rx}. We say that y is a *top vertex* (see Figure 3).

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 closest ancestor of v in T^*_x which is not an exterior vertex. Let $A(T^*_x)$ be the set formed by the ancestors $a(v)$ for all exterior vertices v which are leaves in T^*_x. Note that every vertex in $P(T^*_x)$ and every vertex in $A(T^*_x)$ is a leaf of some tree in F.

We need the following property of the sets $P(T^*_x)$, $B_r(T^*_x)$, and $A(T^*_x)$ to prove the bound of 2 for the approximation ratio of our algorithm.

Lemma 4.1 *In every subtree T^*_x of T^*, the sets $P(T^*_x)$, $A(T^*_x)$, and $B_r(T^*_x)$ are disjoint.*

Proof. 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 $v \notin A(T)$. Let us assume that $v \in P(T^*_x)$. Since v cannot be adjacent to an exterior vertex, then it must be adjacent to some vertex $u \in T_j \neq T_i$. Note that if our algorithm builds tree T_i before T_j, then vertex u would have been placed as child of v in T_i. Similarly, if tree T_j is built first, then vertex u would have been expanded with a rule of priority 2 since v has at least three neighbors in T_i. This expansion would place v as a child of u in T_j. Therefore, sets $B_r(T^*_x)$ and $P(T^*_x)$ are also disjoint.

To show that $P(T^*_x)$ and $A(T^*_x)$ are disjoint, assume the opposite, i.e., there is a vertex v in some tree T_i such that $v \in P(T^*_x)$ and $v \in A(T^*_x)$. Since no vertex can be adjacent to two exterior vertices, then v must be adjacent to some vertex $u \in T_j \neq T_i$. Note that tree T_i cannot be built before T_j because then v would be expanded with a rule of priority 2. This cannot happen since v is a leaf of T_i. Also, T_j cannot be formed before T_i because then u would have been expanded with a rule of priority at least 1 that would have placed v as its child. Hence sets $P(T^*_x)$ and $A(T^*_x)$ must be disjoint. □

The *deficit* of a subtree T^*_x of T^*, denoted as *deficit*(T^*_x), is defined 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 4.1 will show that

$$\begin{aligned}
\ell(T^*) &\leq |V(T^*)| - |A(T^*)| - |P(T^*)| - |B_r(T^*)| + 1 \\
&= |V(F)| - |P(T^*)| - B_r(T^*)| + 1 \\
&< |V(F)| - |B(F)| - k - k \text{ because there are at least } k \text{ top vertices in } T^* \\
&\qquad\qquad\qquad\qquad \text{and } |B_r(T^*)| = |B(F)| + k + 1.
\end{aligned}$$

This will immediately prove the following theorem.

Theorem 4.1 *Algorithm tree finds a spanning tree with at least half of the number of leaves in T^*.*

Proof. By the proof of Lemma 3.4, the tree T built by our algorithm has $|\ell(T)| \geq (|V(F)| - |B(F)| - 2k)/2$. Hence by the above discussion,

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

□

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 4.1. We say that a subtree T_x^* has *maximum deficit one* if *deficit*$(T_x^*) = 1$ and for every vertex v of T_x^* the deficit of the subtree T_v^* is at most 1. We now show some properties of trees of maximum deficit one which we need to prove the bound for the deficit of T^*.

Lemma 4.2 *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 4.3 *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 4.4 *If T_x^* is a subtree of maximum deficit one, then its root x is either a leaf or it has at lest 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 vertex u, and by Lemma 3.1 the subtree T_u^* has deficit at most zero. Thus the only way in which T_x^* can have deficit 1 is if x has another child v and *deficit*$(T_v^*) = 1$. (2) If $x \in V(T_x^*) - B_r(T_x^*) - P(T_x^*)$, then it is not difficult to see that 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 $x \in B_r(T_x^*)$ and x is not a leaf of T^* then the deficit of subtree T_x^* is smaller than 1, and

(b) if $x \in P(T_x^*)$ then x has at least two children u and v such that u belongs to the same cluster as x and *deficit*$(T_u^*) = 1$, and v does not belong to the same cluster as x and *deficit*$(T_v^*) < 1$.

We prove (a) and (b) by induction on the number of vertices in T_x^*. For the basis of the induction, we assume that T_x^* consists of a single vertex x, and that x is a leaf of T^*. Thus claims (a) and (b) trivially hold.

For the induction step we need to consider two different cases, $x \in B_r(T_x^*)$ and $x \in P(T_x^*)$. If $x \in B_r(T_x^*)$, we show that T_x^* cannot have maximum deficit 1. Let u be a child of x and *deficit*$(T_u^*) = 1$. Note that u cannot be a leaf of T^* because then the only way in which *deficit*(T_u^*) can be 1 is if $u \in B_r(T^*)$. By Lemma 4.3 this cannot happen. Hence, u is an internal vertex of T^*. By induction hypothesis all internal vertices v of T_u^* for which *deficit*$(T_v^*) = 1$ have at least two children. To simplify the argument we modify the tree T_x^* as follows. For each internal vertex v of T_u^* for which *deficit*$(T_v^*) = 1$ remove all its children except two of them chosen as follows.

1. If $v \in V(T_x^*) - A(T_x^*) - P(T_x^*)$, then keep two children of v that are roots of subtrees of deficit one.

2. If $v \in A(T_x^*)$, then keep the exterior vertex adjacent to v and one of its children that is the root of a subtree of deficit one.

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 every internal vertex of T_x^*, with the possible exception of x, have degree 3. Also, by Lemma 4.2, at least one leaf of T_x^* belongs to $B_r(T_x^*)$ and so there are at least two vertices in set $B_r(T_x^*)$. 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 our algorithm 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 after vertex w_1 is added to F all its neighbors must belong to F. This means that if w_1 is a leaf then its parent must be in S, and if $w_1 = x$ then u must belong to S. By induction hypothesis w_1 cannot be an internal vertex of T_u^*.

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 u would also belong to S. Since u is an internal vertex, then x would not be the last vertex in L. 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 would be expanded by our algorithm using a rule of priority 2 before w_2 is added to F. By definition of S, this 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 the deficit of tree $T_{y_1}^*$ is 1 and so by Lemma 4.2 y_1 must belong to $B_r(T_x^*)$. Since we assumed that w_1 was the only vertex from $B_r(T_x^*)$ in S then it must be the case that $y_1 = w_1$. But then by definition of y, vertex y_2 must also be a leaf and $\mathit{deficit}(T_{y_2}^*) = 1$. By Lemma 4.2, y_2 must also 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. $y \in P(T_x^*)$. Then either *(i)* y_1 is exterior, *(ii)* y and y_1 belong to the same cluster and $y_1 \in B_r(T_x^*)$, or *(iii)* y_1 and y belong to different clusters. We argued above against the first possibility. If $y_1 \in B_r(T_x^*)$ then we argued above that $y_1 = w_1$. This means that y_2 must be a leaf in a cluster different from y. In this case simply rename y_2 as y_1 and reduce case *(ii)* to case *(iii)*. For *(iii)* let us assume that y belongs to cluster T_i and y_1 to cluster T_j. We know that y has three neighbors. As above let y_2 be the neighbor of y that belongs to S and let y_3 be the third neighbor. Note that if y_3 is an internal vertex, then it cannot belong to S because then L would not be the longest path as assumed above. Also, if y_3 is a leaf then it must belong to $B_r(T_x^*)$ since by induction hypothesis *deficit*$(T_{y_3}^*) = 1$. Therefore vertex y_3 does not belong to S.

   By Lemma 3.2 vertex y_3 must belong to cluster T_i or to cluster T_j. If y_3 is in cluster T_j then since $y_3 \notin S$ algorithm *tree* must build tree T_i before T_j. But this means that the algorithm would have expanded y by placing y_1 and y_3 as its children in T_i. Hence y_3 must belong to cluster T_i and T_j must be built before T_i.

   Since y is a top vertex then, by definition, its parent in T_x^* must belong to cluster T_i and, moreover, by induction hypothesis one of its children also belongs to T_i. This leads also to a contradiction since then our algorithm would have expanded vertex y_1 with a rule of priority 1 making y a black vertex in T_j.

The above argument proves that an internal vertex from $B_r(T^*)$ cannot be the root of a subtree of T^* of maximum deficit one.

We now prove claim (b). Let $x \in P(T_x^*)$ and let u be a child of x that belongs to a different cluster from x. Let us assume that *deficit*$(T_u^*) = 1$ and derive a contradiction from such assumption. This will prove (b). Note that by definition u cannot be a top vertex, and u cannot belong to $B_r(T_x^*)$ because otherwise algorithm *tree* would have put u and x in the same cluster. Observe also that u cannot belong to the set $A(T_x^*)$ because then vertex u would be adjacent to two vertices (x and an exterior vertex) not in T_j, and hence u would be expanded by our algorithm and x would be placed as its child in cluster T_j.

Let x belong to cluster T_i and u to cluster T_j. Since by induction hypothesis u has two children, algorithm *tree* cannot form cluster T_i before cluster T_j because then x would expanded with a rule of priority 1 making u a black vertex in T_i. So we assume that cluster T_j is built before cluster T_i.

Consider the tree T_u^*. Let us trim the tree as described above so that each internal vertex has exactly two children and call the resulting tree T_u^*. As above, let w_1 and w_2 be the first and second vertices from $B_r(T_u^*)$ that are added to F by our algorithm. Let S be the set of vertices from T_u^* which have been added to F just before w_2 is included in F. Let L be a longest path starting at w_1 and going only through internal vertices of T_u^* that belong to S. Let y be the last vertex in L. Using the same arguments as above we can show that y cannot belong to $V(T_u^*)$, which is a contradiction. □

We are ready to prove that the deficit of tree T^* is at most 1, and hence that the approximation ratio of algorithm *tree* is 2 as claimed in Theorem 4.1.

Lemma 4.5 *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 4.4 we know that x cannot belong to $B_r(T^*_x)$. 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 4.4 with the only exception that for vertex$x$ we keep three children u, v, and w as follows.

1. If $x \in A(T^*_x)$, then choose u to be an exterior vertex and v and w to be roots of trees of maximum deficit 1.

2. If $x \in P(T^*_x)$, then choose u to be a child that belongs to a cluster different from x, and v and w to be children in the same cluster as x and which are roots of trees of maximum deficit 1.

3. If $x \in V(T^*_x) - A(T^*_x) - P(T^*_x)$, then choose u, v, and w to be roots of trees of maximum deficit 1.

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 and w_2 be the first and second vertices from $B_r(T^*_x)$ which are included in some tree of F by algorithm *tree*. Let S be the set of vertices from T^*_x which have been added to forest F just before w_2 is included in some tree of F. Let L be a longest path starting at w_1 and passing through internal vertices of T^*_x which belong to S. Since by Lemma 4.4 every vertex in T^*_x has degree at least three, we can use the same arguments as in the proof of Lemma 4.4 to derive a contradiction. This proves that *deficit*$(T^*_x) \leq 1$ for all vertices x. □

The approximation ratio stated in Theorem 4.1 is tight as the example in Figure 4 shows. The spanning tree T^* with maximum number of leaves is shown in Figure 4(b). This tree has $2k-1$ leaves, where the number of vertices in the graph is $3k+1$. The tree found by algorithm *tree* is shown in Figure 4(c) assuming that the first vertex selected by the algorithm is vertex r. This tree has only $k+2$ leaves.

5 Implementation Issues

In this section we briefly describe how algorithm *tree* can be implemented to run in linear time. We represent the input graph $G = (V, E)$ as an adjacency list N. There is an entry $N(v)$ for each vertex $v \in V$. Entry $N(v)$ stores a linked list $L(v)$ containing the neighbors of v, and a variable $t(v)$ (with initial value null) that will indicate the tree T_i to which vertex v belongs in

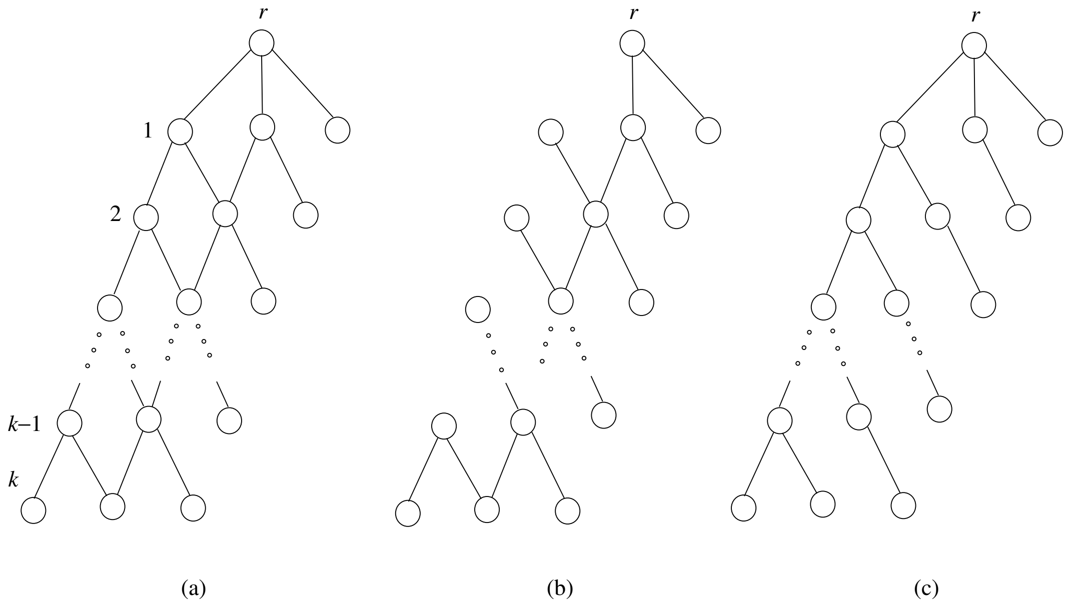


Figure 4: Example showing that the approximation ratio of 2 is tight for algorithm *tree*. (a) Input graph, (b) optimal tree, (c) tree selected by the algorithm.

F. For each neighbor u of v there is a node in $L(v)$, this node stores two pointers, a pointer to entry $N(u)$ and a pointer to the cell that represents vertex v in the list $L(u)$. We use a list T_L to store the leaves of the current tree. We also use the following three pointers: p_1 and p_2 which are used to scan list T_L looking for a leaf that can be expanded with a rule of priority 1 or a rule of priority 2 respectively, and pointer $p_3$is used to scan array N looking for a vertex of degree at least 3.

To build forest F, we scan N with p_3, computing for each entry $N(v)$ the number of neighbors of vertex v. When a vertex v of degree at least 3 is found, such a vertex is placed as the root of a new tree T_v. Then vertex v is expanded by placing all its neighbors as its children in T_v. Next, for each neighbor u of v, vertex v is removed from the list of neighbors of u in $L(u)$, and vertex u is marked as belonging to tree T_v. All the neighbors of v are placed in T_L.

Then list T_L is scanned with pointer p_2 to find a leaf that can be expanded with a rule of priority 2. For each entry $T_L(v)$, the list of neighbors of v is traversed and the neighbors that already belong to T_v are removed from the list. If a vertex v with at least 2 neighbors not in T_v is found, then it is expanded as described above. If no such vertex is found then the list is scanned with pointer p_1 to find a vertex that can be expanded with rule 1. This process is repeated until there are no more vertices in T_L that can be expanded. When that happens, tree T_v is placed in F. Then list N is scanned again to find a vertex of degree at least 3 as described above.

Note that the list of neighbors of a vertex is scanned at most three times, one with pointer p_3, one with p_2, and one with p_1, so the total time needed is $O(m)$. Also the expansion of a vertex takes time proportional to its degree, and therefore the total time needed to form forest F is $O(m)$.

Once the forest F is built, every tree $T_i \in F$ is contracted to a single vertex and depth first search is used to find a spanning tree of the contracted graph. These edges are added to F to form the spanning tree T. The total time needed by the algorithm is $O(m)$.

6 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 now 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 of edges S are leaves. There are two conditions that need to hold 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 $V - S$. Thus 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. Clearly 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 have not yet been added 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 the vertices in 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 $v, w \in C$, and let $v \in C_1$. Then, $C - \{v\}$ would also cover S, which cannot happen since C is a minimal set that covers S. 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 6.1 $\ell(T) \geq \max\{|S'|, \ell(T'), \frac{2}{3}(|\ell(T')| + |S'|)\}$.

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 $\ell(T) = \ell(T') - |C| + |S| \geq \ell(T')$.

We now show that $\ell(T) \geq \frac{2}{3}(|\ell(T') + |S'|)$. Let C_1 be the set formed by all vertices in C that have only one child, and let S_1' be the set of children of C_1. Observe 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 it follows that $|\ell(T') - C| \geq |C_1|$. Therefore,

$$\begin{aligned} 3(|S' - S_1'| + |C_1| + |\ell(T') - C|) &\geq 2(|S' - S_1'| + |C_1| + |\ell(T') - C|) + 2|C - C_1| + |C_1| + |C_1| \\ &= 2(|S'| - |S_1'| + |C_1| + |\ell(T')|) \\ &= 2(|S'| + |\ell(T')|), \text{ because } |S_1'| = |C_1|. \end{aligned}$$

Since the $\ell(T) = |S' - S_1'| + |C_1| + |\ell(T') - C|$ 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 6.1 $\ell(T^*)/\ell(T) \leq 5/2$.

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 in it. Let T' be as defined above. Then by Theorem 4.1, $\ell(T^+)/\ell(T') \leq 2$. Also, note that $\ell(T^*) \leq \ell(T^+) + |S'|$, hence

1. if $|S'| \leq \ell(T')/2$, then by Lemma 6.1,

$$\frac{\ell(T^*)}{\ell(T)} \leq \frac{\ell(T^+) + |S'|}{\ell(T')} \leq 2 + \frac{1}{2} = \frac{5}{2},$$

2. if $|S'| \geq 2\ell(T')$, then by Lemma 6.1,

$$\frac{\ell(T^*)}{\ell(T)} \leq \frac{\ell(T^+) + |S'|}{|S'|} \leq \frac{\ell(T^+)}{2\ell(T')} + 1 = 2,$$

3. if $\ell(T')/2 < |S'| < 2\ell(T')$, then by Lemma 6.1,

$$\frac{\ell(T^*)}{\ell(T)} \leq \frac{\ell(T^+) + |S'|}{\frac{2}{3}(\ell(T') + |S'|)} \leq \frac{3}{2} + \frac{\ell(T')}{\frac{2}{3}(\ell(T') + |S'|)} \leq \frac{3}{2} + 1 = \frac{5}{2}.$$

□

References

[1] W. Duckworth, P.E. Dunne, A.M. Gibbons, and M. Zito, *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*, 28th ACM Symposium on Theory of Computing (1996), pp. 314–318.

[3] G. Galbiati, F. Maffioli, and A. Morzenti, *A short note on the approximability of the maximum leaves spanning tree problem*, Information Processing Letters, 52 (1994), pp. 45–49.

[4] J.R. Griggs, D.J. Kleitman, and A. Shastri, *Spanning trees with many leaves in cubic graphs*, Journal of Graph Theory, 13 (1989), pp. 669–695.

[5] S. Guha and S. Khuller, *Approximation algorithms for connected dominating sets*, Proceedings of the Fourth Annual European Symposium on Algorithms (1996), pp. 179–193.

[6] S. Guha, and S. Khuller, *Improved methods for approximating node weight Steiner trees and connected dominating sets*, Technical Report CS-TR-3849, 1997, University of Maryland.

[7] D.J. Kleitman and D.B. West, *Spanning trees with many leaves*, SIAM Journal on Discrete Mathematics, 4 (1991), pp. 99–106.

[8] H. Lu and R. Ravi, *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, pp. 533–542.

[9] H. Lu and R. Ravi, *A near-linear time approximation algorithm for maximum-leaf spanning tree*, Technical report CS-96-06, Brown University, 1996.

[10] C. Payan, M. Tchuente, and N.H. Xuong, *Arbres avec un nombre de maximum de sommets pendants*, Discrete Mathematics, 49 (1984), pp. 267–273.

[11] J.A. Storer, *Constructing full spanning trees for cubic graphs*, Information Processing Letters, 13 (1981), pp. 8–11.