An Approximation Algorithm for the Maximum Leaf Spanning Arborescence Problem

MATTHEW DRESCHER*
McGill University
and
ADRIAN VETTA†
McGill University

We present an $O(\sqrt{\text{OPT}})$-approximation algorithm for the maximum leaf spanning arborescence problem, where OPT is the number of leaves in an optimal spanning arborescence. The result is based upon an $O(1)$-approximation algorithm for a special class of directed graphs called willows. Incorporating the method for willow graphs as a subroutine in a local improvement algorithm gives the bound for general directed graphs.

Categories and Subject Descriptors: G.2.1 [**Mathematics of Computing**]: Combinatorics—*Combinatorial Algorithms*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: approximation algorithms, directed graphs, maximum leaf, arborescence

1. INTRODUCTION

The vertices of any spanning tree can be naturally partitioned into two types: leaves and internal vertices. A natural question is then to find a spanning tree T in a undirected graph G containing the maximum number of leaves. This is know as the *Maximum Leaf Spanning Tree* (MLST) problem and has applications in communication network design [Guha and Khuller 1996], circuit layouts [Storer 1981], and distributed systems [Payan et al. 1984]. This problem, however, is hard; Galbiati et al [Galbiati et al. 2004] proved it to be MAXSNP-complete.

Given this, there has been much work on designing approximation algorithms. Interestingly, it turns out that the MLST problem is one of those select problems for which many of the standard tools give good approximation guarantees. In particular, Lu and Ravi have shown that both local improvement and greedy techniques can be applied successfully. First, they gave constant-factor approximation algorithm using a local improvement algorithm [Lu and Ravi 1992]. Specifically, they analysed a *k-exchange algorithm*. This involves taking a spanning tree T and exhaustively searching for a spanning tree T' that differs from T in at most k edges

* School of Computer Science, McGill University. Email: knavely@gmail.com
† Department of Mathematics and Statistics, and School of Computer Science, McGill University. Supported in part by NSERC grant 28833-04 and FQRNT grant NC-98649. Email: vetta@math.mcgill.ca

and has more leaves; the search is then repeated until no improved tree can be found. Lu and Ravi proved that this produced approximation guarantees of 5 and 3, respectively, for the cases $k = 1$ and $k = 2$.

Later, Lu and Ravi [Lu and Ravi 1998] proved that greedy algorithms also perform well. They presented a nearly linear time algorithm with a factor 3 guarantee. The basic approach is to greedily grow a forest (according to specific *expansion rules*) with many leaves. Using the structural properties of the resulting forest, they show that the forest can then be extended to a spanning tree with a relatively small cost in terms of leaves. Solis-Oba [Solis-Oba 1998] subsequently refined this approach to produce a factor 2 approximation algorithm.

In this paper we consider directed graphs. The analog of spanning tree in a directed graph is called an *arborescence*; more precisely an arborescence is a directed graph G that contains a unique vertex r such that there is exactly one directed path in G from r to each vertex $u \in G$.

Our problem then becomes the *Maximum Leaf Spanning Arborescence* (MLSA) problem: Given a directed graph G, find a spanning arborescence T containing the maximum number of leaves. Here, we will assume that we are also given a specified root vertex r for the arborescence. Clearly, an algorithm for this case can be used to solve the MLSA problem by applying it on each possible choice of root.

As with many graph problems, the directed version seems trickier than its undirected counterpart. For example, the techniques developed for the MLST problem cannot be applied to the MLSA problem. Figure 1 gives a spanning arborescence T on which a k-exchange algorithm performs poorly. To improve T requires exchanging more than half of the arcs of the arborescence; that is, we need $k > \frac{1}{2}n$. Moreover doing so enables us to find the optimal arborescence T' which contains $\frac{1}{2}n$ leaves compared to just two in T. Consequently, the k-exchange algorithm gives a trivial $\theta(\text{OPT})$-approximation guarantee. Similarly, in the directed setting, it is easy to construct bad examples for all obvious types of greedy algorithm. For example, greedily growing a forest will fail as the arc directions may then prohibit the forest from being connected up efficiently.

1.1 Related Work.

Very recently, Alon et al. [Alon et al. 2007a], [Alon et al. 2007b] examined the parameterised complexity of MLSA. They showed that MLSA is *fixed parameter tractable* for a large class of directed graphs. Specifically, their algorithm applies to the family $\mathcal{F}$ of digraphs for which the maximum leaf arborescence has the same number of leaves as the maximum leaf spanning arborescence. For example, $\mathcal{F}$ includes all strongly connected digraphs; it does not, however, contain the digraph $D = (V, A)$, where $V = [n]$ and $A = \{(i, i+1) : 1 \leq i \leq n-1\} \cup \{(n, j) : 2 \leq j \leq n-1\}$. Their approach is to show that a digraph $D \in \mathcal{F}$ either has a spanning arborescence with at least k leaves or the underlying graph has pathwidth at most $f(k)$. For fixed k, dynamic programming can be applied in the latter case, giving a polynomial time algorithm that decides whether or not a digraph $D \in \mathcal{F}$ has a spanning arborescence with at least k leaves. We remark that they left as an open problem the parameterized complexity of MLSA for all digraphs, which was even more recently settled in [Bonsma and Dorn 2007].

Although the problem of determining the parameterized complexity and the prob-

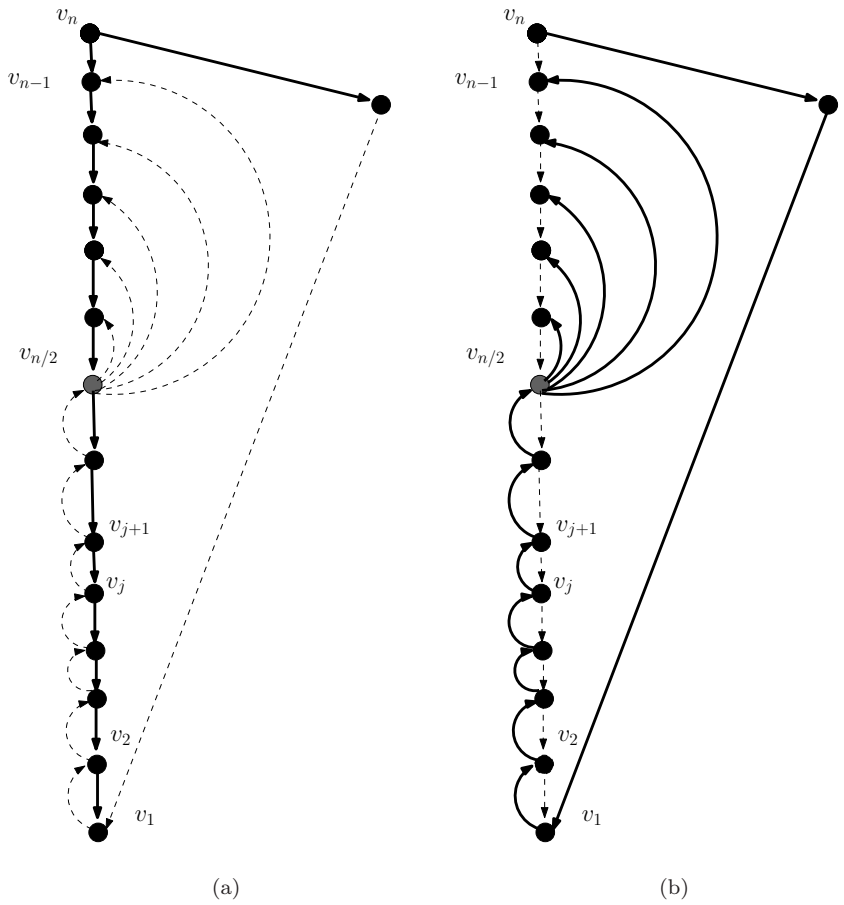

Fig. 1. (a) Initial spanning arborescence with 2 leaves. (b) After an arbitrarily high number of edge swaps we have a new spanning arborescence with an arbitrarily high number of leaves.

lem of finding an approximation algorithm for MLSA are quite different, there are common techniques that have proved to be successful for both problems. The parameterized algorithms in [Alon et al. 2007a] are based on local improvements. In this paper, we introduce a special class of directed graphs which we call "willow" graphs. Both the parameterized algorithms of [Alon et al. 2007a], [Alon et al. 2007b] implicitly make use of "willow" graphs in their proof on bounds of pathwidth.

1.2 Our Contribution.

Our approach is motivated by the example in Figure 1. How can we deal with the difficulty inherent in this simple directed graph? To do this, we consider a specific family of directed graphs, called willow graphs, which contains the structures causing the problems in Figure 1. We first present a constant factor approximation algorithm for the MLSA problem in willow graphs.

The key observation then is that, in a directed graph, paths in an arborescence T must induce willow-like graphs; otherwise we can easily obtain local improvements to T. Consequently, we can apply the algorithm for willow graphs as a subroutine in a $O(\sqrt{\text{OPT}})$-approximation algorithm for the MLSA problem in directed graphs, where OPT is the number of leaves in an optimal spanning arborescence.

2. AN APPROXIMATION ALGORITHM FOR WILLOW GRAPHS

We begin with some terminology. Let G be a directed graph with arc set E. We define the *out-degree* of a vertex v, denoted by $\deg^+(v)$, to be the number of arcs in E emanating from v; more formally $\deg^+(v) := |\{u \in G : (v,u) \in E\}|$. Let T be an arborescence in G and $i \in \mathbb{N}$. We denote by T_i the set of vertices with out-degree i in T; similarly, $T_{\geq i}$ denotes the set of vertices with out-degree at least i.

We now state some general observations that apply to any directed graph G. Observe that if a vertex v is an internal (non-leaf) vertex in an arborescence $T \subseteq G$, then adding any outgoing arc from v does not reduce the number of leaf vertices. It follows that we wish to select a subset R of vertices such that R is spanned by an arborescence rooted at r and every vertex $u \in V - R$ has an incoming arc whose tail is in R.

2.1 Willows.

An ordering $\{v_1, v_2, \ldots, v_n\}$ of the vertices of a directed graph partitions the arc set into two groups; *up arcs* (v_i, v_j) satisfy $i < j$ and *down arcs* (v_i, v_j) satisfy $i > j$. A directed graph $W = (V, A)$ is called a *willow* if V has a vertex ordering for which the *down arcs* are precisely a Hamiltonian path H.

For vertices $v_i, v_j \in H$, we say that v_i is *lower* than v_j if $i < j$. We denote this using the " $<$ " operator and write $v_i < v_j$. We define a *closed interval* $[v_i, v_j] := \{v \in W : v_i \leq v \leq v_j\}$; an open interval can be defined analogously.

Observe that in Figure 1, our bad example for the k-exchange algorithm, the induced subgraph on $v_1, v_2, \ldots v_n$ contains an induced willow. Consequently, being able to deal with willows is a necessary attribute of any good algorithm. Conversely, we will show in Section 3 that an approximation algorithm for the maximum leaf arborescence problem in a willow gives an approximation algorithm for the general case (albeit with a weaker approximation guarantee). Ergo, in this section we present an approximation algorithm for willows.

2.2 Pitchforks.

An important structure for our algorithm is a pitchfork. A *pitchfork* consists of

(i) A directed path $\{w_0, w_1, \ldots, w_k\}$ with *tail* w_0 and *head* w_k. The arcs of the path form a *handle*. (The handle needs not be non-empty; in this case $w_0 = w_k$).

(ii) A set of at least two arcs emanating from the head w_k, called *prongs*, that point to vertices (called *prong vertices* or just *prongs* if there is no confusion) disjoint from the handle.

Figure 2 illustrates a pitchfork. To understand why pitchforks will be useful, consider the following simple result.

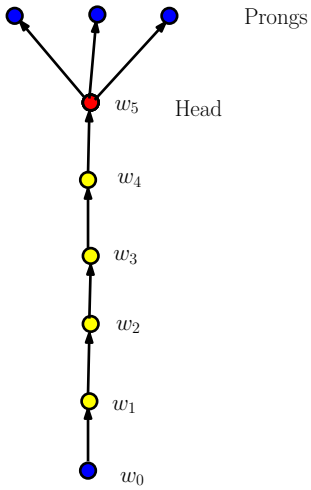

Fig. 2. A 3-prong pitchfork with a 5-arc handle.

LEMMA 2.1. *In any arborescence T, the number of leaves is greater than the number of vertices with out-degree at least two.*

Proof. The number of leaves in T is

$$|T_0| = 1 + \sum_{v:\deg^+(v)\geq 2} (\deg^+(v) - 1) \;\geq\; 1 + |T_{\geq 2}|. \quad \square$$

Hence, an arborescence with a large number of nodes of out-degree at least two has a large number of leaves. This observation motivates our algorithm. The idea is to grow an arborescence T by adding one pitchfork at a time; the tail of the pitchfork should be in the current arborescence with all of its other vertices lying outside the current arborescence. Intuitively, if the handles of the pitchforks are small then the constructed arborescence will not have too many internal nodes of degree one and will thus have a large number of leaves by Lemma 2.1.

Of course, such an approach faces two immediate difficulties. First, what happens if the algorithm finds pitchforks with long handles? Secondly, what if the algorithm cannot find any pitchfork at all? The latter problem is easy to deal with: If the algorithm "gets stuck" we will obtain a certificate showing that the optimal solution cannot do much better at that point. We have to deal with the former problem in a similar fashion; the need for the output of long handles implies an improved upper bound. However, showing this is not quite as straightforward and requires a more careful examination of the structure of the algorithm. Consequently, we will now describe the algorithm formally and then see how it provides a constant factor approximation algorithm for willow graphs.

2.3 AN ALGORITHM FOR WILLOW GRAPHS.

Take a willow W with Hamilton path $H := v_n, v_{n-1}, ..., v_1$. We remark that it is easy to determine in polynomial time if W has a spanning arborescence rooted at v_1. Therefore, we may assume that W does contain such an arborescence.

From here on, we will use the words "tree" and "arborescence" interchangeably since the latter is cumbersome. We hope there is no confusion as we will only be considering directed graphs.

Recall that our algorithm constructs T, beginning with v_1, by finding and adding pitchforks. In order to obtain an upper bound on the performance of the algorithm we will colour the vertices of the graph when we add them to the arborescence. We color the head of a pitchfork *red*, the internal vertices on the handle *yellow*, and the prong vertices *blue*.

We will attempt to colour the vertices of G in a roughly contiguous fashion from v_1 up to v_n. To do this we begin, at time $t = 0$, by colouring the root vertex v_1 blue. Subsequently, at time t, we focus on the lowest uncoloured interval, say $I_t = [v_{i+1}, v_{j-1}]$. We let $\mathcal{P}_t$ denote the set of pitchforks whose tail vertices lie in the current tree, whose handles lie in $I_t \cup \{v_j\}$, and whose prongs are uncoloured. We search for a minimal pitchfork $F_t \in \mathcal{P}_t$ whose highest handle vertex $v_k \leq v_j$ is minimised. By "minimal" we mean that the handle of F_t does not properly contain the handle of another pitchfork in $\mathcal{P}_t$. If such a pitchfork exists, we add it to the current tree. Note that the tail of the handle of such a pitchfork must be either a blue vertex in $\{v_1, \ldots, v_i\}$ or the blue vertex v_j. Then the head of the pitchfork either coincides with its tail or is in the interval $I_t \cup \{v_j\}$ if the handle is non-empty.

If such a pitchfork does not exist then we have two possibilities. If $j \leq n$ then we add the downpath from v_j to v_{i+1}; the endpoints v_{i+1} and v_j are coloured blue and the vertices in the interior of the downpath are coloured yellow. If v_{i+1} is not a leaf at the end of the algorithm, then this downpath must, at some point in the algorithm, have been followed by a pitchfork with tail at v_{i+1}; that is, two downpaths cannot be joined consecutively. If $j = n+1$ then we add a path through all the remaining uncoloured vertices; again the endpoints of this path are coloured blue and the interior vertices are coloured yellow. We then repeat and consider the next uncoloured interval.

Our algorithm also partitions the vertices into intervals by marking certain vertices. The set of marked vertices is denoted by K, and K is initially empty at time $t = 0$. When we attach a pitchfork to the current tree, we add its highest handle vertex to K if it is higher than all vertices in K. Similarly, when we attach a downpath starting at v_j, we add v_j to K if v_j is higher than all vertices in K. If the algorithm is forced to terminate by attaching a final path through all the uncoloured vertices then we add the vertex v_n to K. The following table formalizes the algorithm and Figure 3 illustrates an example.

Observe that requiring our pitchforks to be "minimal" gives the following characterization of yellow vertices.

Observation 2.2. *The yellow vertices are those vertices that are not blue and have exactly one uncoloured out-neighbor when they are added to the tree.*

Proof. By construction we see that a yellow vertex must have at least one uncolored out-neighbor when it is added to the tree as an internal vertex in some path. A vertex y is coloured yellow by the algorithm either at stage 3(ii) or 4(ii). In the latter case, y can have have no more than one uncoloured out-neighbor or else it is easy to see that this would violate the condition of step (4) that $\mathcal{P}_t = \emptyset$. In the former case, y was coloured yellow and added to the tree as part of a minimal

pitchfork F_t. However it is easy to see that taking y's uncolored neighbors as the prongs, y as the head, and the handle of F_t up to y gives a pitchfork in $\mathcal{P}_t$ whose handle is properly contained in the handle of F_t, contradicting the minimality of F_t. □

WILLOW(G)

(1) INITIALIZE $t := 0$, colour v_1 blue, $T := v_1, K := \emptyset$; $t := 1$.
(2) GIVEN T and t, let $I_t := [v_{i+1}, v_{j-1}]$ be the lowest uncoloured interval.
—Let $\mathcal{P}_t$ denote the set of pitchforks whose tail vertices lie in the current tree T, whose handles lie in $I_t \cup \{v_j\}$, and whose prongs are uncoloured.
—If no such pitchforks exist then set $\mathcal{P}_t := \emptyset$.
(3) If $\mathcal{P}_t \neq \emptyset$ then
(i) Let $F_t \in \mathcal{P}_t$ be a minimal pitchfork whose highest handle vertex is minimized. [By "minimal" we mean that the handle of F_t does not properly contain the handle of another pitchfork in $\mathcal{P}_t$.]
(ii) Set $T := T \cup F_t$.
—Colour F_t. Colour the head h_t *red*, the internal handle vertices *yellow* and the prongs *blue*.
(iii) Let k_t be the highest handle vertex of F_t. If $k_t > \max\{k \in K\}$ then set $K := K \cup k_t$.
(4) If $\mathcal{P}_t = \emptyset$ then
(i) If $j \leq n$ then
—Add the downpath $F_t := \{v_j, v_{j-1}, \ldots, v_{i+1}\}$ to T.
—Colour the vertices $[v_{i+2}, v_{j-1}]$ yellow, and colour v_{i+1} blue.
—If $v_j > \max\{k \in K\}$ then set $K := K \cup v_j$.
(ii) If $j - 1 = n$ then extend T to a spanning arborescence as follows:
—Find a path P that starts at somevertex of T and goes through every remaining uncoloured vertex.
—Colour the last vertex of P *blue*, and colour all internal vertices of P *yellow*.
—Set $K := K \cup v_n$.
(5) TERMINATE if all vertices are coloured; otherwise set $t := t + 1$, and goto (2).

We remark that both the colouring and marking are not required by the algorithm; they will be used solely to guide the analysis of the algorithm's performance. Some comments are in order here. First observe that when we add a pitchfork the tail vertex will have already been coloured blue. Consequently, a vertex may receive more than one colour; for example, a vertex could first be added to the tree as a prong vertex, and later could be the head of a pitchfork with an empty handle (and, is thus also the tail of the pitchfork); such a vertex will be coloured both blue and red. The set of yellow vertices, however, intersects neither the set of red vertices nor the set of blue vertices. Secondly, the set K of markers naturally partition the vertices into intervals. The markers also possess the following useful property.

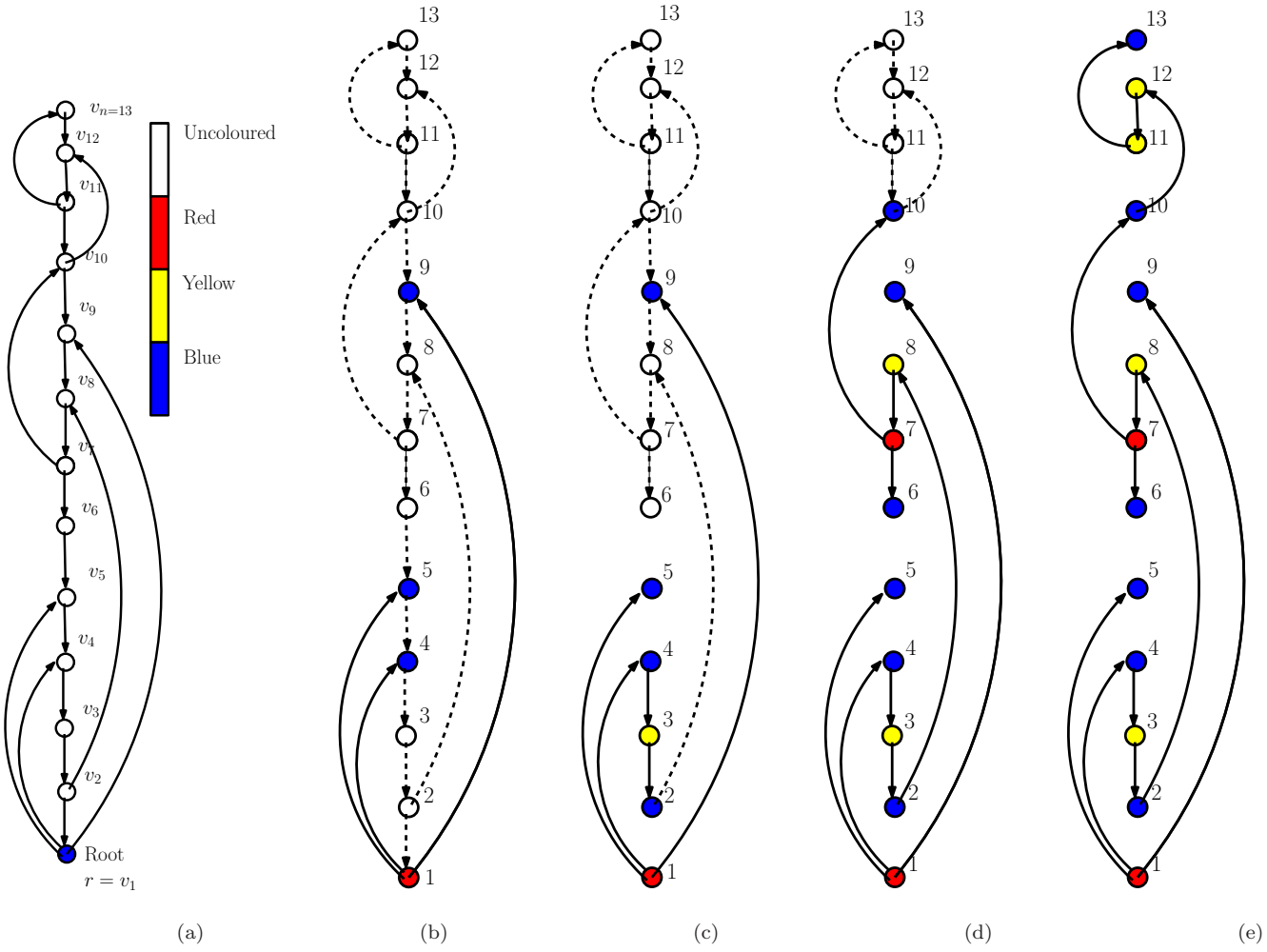

Fig. 3. (a) Time $t = 0$ and the willow W. (b) At time $t = 1$, we have $I_2 = [v_2, v_{13}]$ and add the pitchfork with head v_1 and prongs v_4, v_5, v_9; $K = \{v_1\}$. (c) At time $t = 2$, we have $I_2 = [v_2, v_3]$ and $\mathcal{P}_2 = \emptyset$ so the algorithm adds the downpath; $K = \{v_1, v_3\}$. (d) At time $t = 3$, we have $I_3 = [v_6, v_7, v_8]$ and the algorithm adds a pitchfork with tail v_2, handle v_8, v_7, head v_7, and prongs v_6, v_{10}; $K = \{v_1, v_3, v_8\}$. (e) At time $t = 4$, we have $I_4 = [v_{11}, v_{12}, v_{13}]$ and $\mathcal{P}_4 = \emptyset$ so T is extended to a spanning tree using path $P = \{v_{10}, v_{12}, v_{11}, v_{13}\}$; $K = \{v_1, v_3, v_8, v_{13}\}$.

Observation 2.3. *At the time vertex k_i is added to K, there are no yellow or red vertices higher than k_{i-1}.* □

Now that we have described our algorithm, we will prove that it is indeed a constant factor approximation algorithm. We will show that the algorithm outputs a spanning arborescence, runs in polynomial time, and has a constant factor worst-case guarantee.

LEMMA 2.4. *The algorithm produces a spanning arborescence.*

Proof. Clearly, step (3) adds a pitchfork and step 4(i) adds a directed path to T, and by construction these objects do exist when they are added. It remains to prove that the path P added in step 4(ii) does exist. Suppose that at time t, the lowest uncoloured interval is $I_t = [v_{i+1}, v_n]$ and that $\mathcal{P}_t = \emptyset$. As there is a spanning arborescence of W rooted at v_1, there is a path $\hat{P}$ in W from v_1 to v_n. Let y be the last vertex of $\hat{P}$ in $[v_1, v_i]$. Let P_1 be the segment of $\hat{P}$ from y to v_n.

If $P_1 = (y, v_n)$ then adding the downpath from v_n to v_{i+1} gives the desired path through all the uncoloured vertices. Otherwise, consider the vertices $V(P_1) \setminus \{y, v_n\}$ that form the interior of the path P_1. We claim these interior vertices form an interval; specifically, $V(P_1) \setminus \{y, v_n\} = [v_{i+1}, v_j]$ for some j such that $i+1 < j < n$. Suppose, for a contradiction, that the interior of P_1 is not an interval. This implies that, for some $s \neq 1$, there exists a vertex v_{i+s} in the interior of P_1 such that v_{i+s-1} is not in the interior of P_1. Let w be the vertex following v_{i+s} in P_1. Observe that the vertices v_{i+s-1} and w are uncoloured at time t because they lie in $I_t = [v_{i+1}, v_n]$. Moreover, v_{i+s-1} and w do not lie on the subpath of P_1 from y to v_{i+s}, denoted $y \xrightarrow{P_1} v_{i+s}$, because v_{i+s-1} is not in the interior of P_1 and w follows v_{i+s}. We then have a pitchfork with tail y, head v_{i+s}, handle $y \xrightarrow{P_1} v_{i+s}$, and prongs v_{i+s-1} and w. This contradicts the assertion that $\mathcal{P}_t = \emptyset$.

Now define P_2 to be the downpath from v_n to v_{j+1}. The path P is the concatenation of P_1 and P_2, and is thus a path through all the uncoloured vertices. □

LEMMA 2.5. *The algorithm runs in polynomial time.*

Proof. First let's see that step (2) can be carried out in polynomial time. It is easy to find $I_t = [v_{i+1}, v_{j-1}]$, so we just need to find a minimal pitchfork F_t with highest handle vertex $k_t \leq v_j$ minimised. This we can do by exhaustively searching over all possible choices for the tail vertex, head vertex, sets of two prongs, and k_t. Given such choices it is easy to check if the desired pitchfork exists in polynomial time. Similarly, it easy to find the path P of step 4(ii) in polynomial time as outlined in Lemma 2.4 □

We now obtain an approximation guarantee for the algorithm. To do this, we utilise the markings and colouring made by the algorithm. Let R, B and Y be the set of red, blue and yellow vertices, respectively. (Recall that the sets R and B may intersect.)

LEMMA 2.6. *The algorithm outputs an arborescence with more than* $\frac{1}{5}(|R|+|B|)$ *leaves.*

Proof. Observe that the set of red vertices is *exactly* the set of vertices in T with out-degree at least 2. Consequently, $|R| < |T_0|$ by Lemma 2.1.

We now prove that $|B| \leq 4|T_0|$. First consider $B \setminus T_0$, the blue vertices that are not leaves in T. We show that $|B \setminus T_0| \leq 3|T_0|$. Let $b \neq v_1$ be a blue vertex that is not a leaf in T. We consider two cases.

(i) *The vertex b has a red descendant.*
Let r be b's closest red descendant in T. We show that at most one other

blue vertex can have r as a closest red descendant. In a pitchfork the red head is immediately followed by a blue prong. Therefore the consecutive blue nodes in the arborescence are separated by a red vertex unless they were added in a downpath. Recall there can not be two consecutive downpaths in the arborescence; moreover a downpath containing b cannot be followed by the final path of Step 4(ii) because b has a red descendent. Thus, no more than two vertices in $B \setminus T_0$ can share a closest red descendant. Therefore, the number of vertices in $B \setminus T_0$ with a red descendant is at most $2|R|$.

(ii) *The vertex b does not have a red descendant.*
We show that the number of vertices in $B \setminus T_0$ without a red descendant is at most $|T_0| + 1$. Suppose that b is the prong of a pitchfork F. Since $b \in B \setminus T_0$ does not have a red descendant, b must be the first vertex of a downpath P. Now the last vertex of P is either a leaf or the first vertex of the final downpath. Since the last vertex of the final downpath is a leaf, the number of vertices in $B \setminus T_0$ without a red descendant is at most $T_0 + 1$.

Therefore, by Lemma 2.1 and noting that v_1 is also blue, we obtain

$$|B \setminus T_0| \leq 2|R| + (T_0 + 1) + 1 \leq 2(T_0 - 1) + (T_0 + 1) + 1 = 3|T_0|.$$

Consequently we obtain an upper bound on the number of blue vertices:

$$|B| = |B \setminus T_0| + |B \cap T_0| \leq 3|T_0| + |T_0| = 4|T_0|.$$

Hence,

$$|R| + |B| < |T_0| + 4|T_0| = 5|T_0|$$

so the algorithm outputs a tree with more than $(1/5)(|R| + |B|)$ leaves. □

Next consider the set of yellow vertices. If we can show that only a small number of yellow vertices can be leaves in the optimal arborescence T^* then we would be done. To do this, we consider the interval partition produced by the set K of markers. Let $K := \{k_1, ..., k_l\}$ be ordered according to the time the vertices were marked by the algorithm. Note that, by construction, for $i < j$, we have $k_i < k_j$.

Notice that an optimal solution T^* contains at most one leaf in $(k_i, k_{i+1}]$ on a path in T^* that contains k_{i+1}, by the definition of "willow". Hence, we can bound the number of these types of leaves by the number of such intervals. Other leaves of T^* in $(k_i, k_{i+1}]$ must be on a path in T^* whose vertices are all lower than k_{i+1}. To bound these types of leaves, we notice two of them cannot share a closest blue ancestor at the time when they were colored because this would give a pitchfork whose highest handle vertex is lower than k_{i+1}, contradicting the definition of k_{i+1}.

THEOREM 2.7. *The number of yellow vertices that are leaves in the optimal solution is at most $2|B| + |R|$.*

Proof. Let T^* denote the optimal arborescence. For each $1 \leq i \leq l$, define a subset A_i of $T_0^* \cap Y$ as:

$$A_i := \{y \in T_0^* \cap Y \cap (k_{i-1}, k_i] : \text{the path from } v_1 \text{ to } y \text{ in } T^* \text{ uses } k_i.\}.$$

Next let $A = \bigcup_{i=1}^{l} A_i$. We will show that $|A| \leq |K| \leq |B| + |R|$ and that $|(T_0^* \cap Y) - A| \leq |B|$; from this, the statement of the theorem follows.

We first prove that $|A_i| \leq 1$. Suppose, for a contradiction, that $y_1, y_2 \in A_i$. Without loss of generality, we can assume $y_1 < y_2$. Recall that by the definition of "willow" there is only one directed path from a vertex to any vertex lower than it. Since the path from v_1 to y_1 in T^* goes through $k_i \geq y_2 > y_1$, the vertex y_2 is on the path from v_1 to y_1 in T^*, which contradicts the fact that y_2 is a leaf in T^*. Hence, there can be at most one vertex of A in each interval $(k_{i-1}, k_i]$ so $|A| \leq |K| = l$. Now $|K| \leq |B| + |R|$ because each marked vertex either belongs to the handle of a distinct pitchfork found in Step 3 or a path found in Step 4 of the algorithm.

We now show that $|(T_0^* \cap Y) - A| \leq |B|$. Let $y_1, y_2 \in (T_0^* \cap Y) - A$. Without loss of generality, assume that $y_1 < y_2$ and that $y_1 \in (k_{j-1}, k_j]$. We will prove that y_1 and y_2 cannot share a common closest blue ancestor in T^*. Suppose, for a contradiction, that b is the closest blue ancestor in T^* of y_1 and y_2. Let P_1 and P_2 be the paths in T^* from v_1 to y_1 and y_2, respectively. Let $t(j)$ be the time at which vertex k_j is coloured, and let w be the last vertex on P_1 that was coloured by the start of time $t(j)$. Since v_1 is coloured blue at time $t = 0$, the vertex w exists. Also, $w \neq y_1$ by Observation 2.3 since y_1 is yellow and higher than k_{j-1}.

We will show that the vertex w must be blue. Let x be the vertex after w on P_1. Suppose first, for a contradiction, that w is red. Since x is an out-neighbor of w, either x was coloured before w or x was coloured blue when w was coloured red; both possibilities contradict the definition of w. Now suppose that w is yellow. By Observation 2.2, this implies that when w was coloured, it had exactly one uncoloured out-neighbor. This out-neighbor must be x so x must have been coloured at the same time as w, a contradiction. Thus, w must be blue and, because b is the closest blue ancestor of y_1, either $w = b$ or w occurs before b on P_1. A similar argument shows that w is the last vertex on P_2 that is coloured blue at the start of time $t(j)$, since b is the closest blue ancestor of y_2 and $y_2 > y_1 > k_{j-1}$.

Consider the tree $P_1 \cup P_2$. Since y_1 and y_2 are leaves in T^*, there exists a unique vertex h whose out-degree is two in $P_1 \cup P_2$. Clearly, h occurs after b on P_1; otherwise b would not be an ancestor of y_1 and y_2. Since $y_1 \in (T_0^* \cap Y) - A$, all vertices on P_1 must be lower than k_j. Now k_j lies in $I_{t(j)}$, the lowest uncoloured interval at time $t(j)$. By the definition of w, the vertices on P_1 after w and up to h are uncoloured at time $t(j)$ and are lower than k_j, so they must lie in $I_{t(j)}$ as well. Observe that the vertex k_{j-1} is not in $I_{t(j)}$ so the vertices on P_1 after w are higher than k_{j-1}.

Now let p_1 and p_2 denote the children of h on the paths P_1 and P_2, respectively. Since p_1 and p_2 occur after w on these paths, they are uncoloured at time $t(j)$. Therefore, at the start of time $t(j)$, we have a pitchfork $F \in \mathcal{P}_{t(j)}$ with tail w, handle $w \xrightarrow{P_1} h$, head h, and prongs p_1, p_2. Since the highest handle vertex of F is lower than k_j and higher than k_{j-1}, this contradicts the definition of k_j.

We have shown that that two vertices in $T_0^* \cap Y - A$ cannot share a common closest blue ancestor in T^*. Therefore, $|T_0^* \cap Y - A| \leq |B|$ so

$$|T_0^* \cap Y| = |A| + |T_0^* \cap Y - A| \leq (|B| + |R|) + |B| = 2|B| + |R|. \quad \square$$

Putting Lemma 2.6 and Theorem 2.7 together gives a constant factor approximation algorithm for the MLSA problem in willow graphs.

THEOREM 2.8. *The algorithm gives a factor* 14 *approximation guarantee for willow graphs.*

Proof. We have that

$$\begin{aligned} |T_0^*| &\leq |(B \cap T_0^*)| + |R \cap T_0^*| + |Y \cap T_0^*| \\ &\leq |B| + |R| + |Y \cap T_0^*| \\ &\leq |B| + |R| + (2|B| + |R|) && \text{(by Theorem 2.7)} \\ &= 3|B| + 2|R| \\ &< 14|T_0| && \text{(by the proof of Lemma 2.6.)} \end{aligned}$$

Therefore, we have a factor 14 approximation algorithm for willow graphs. □

3. AN APPROXIMATION ALGORITHM FOR GENERAL GRAPHS

We are now ready to present an approximation algorithm for general graphs. The algorithm is a glorified local improvement algorithm. It is an iterative algorithm with up to two phases in each iteration. The first phase is a simple 1-exchange algorithm with a *clean-up* step. The clean-up step allows us to partition the graph into willow-like pieces; the second phase then looks for improvements by applying our willow algorithm on each piece.

To guide our algorithm, we will use a colouring scheme similar to that utilized by our willow algorithm. At any point, we will colour our current arborescence T as follows.

—If v has out-degree at least two in T, then colour v *red*.

—If v is the child of a red vertex or is a leaf then colour v *blue*.

—Colour all other vertices *yellow*.

We denote the sets of red, blue and yellow vertices by R, B and Y, respectively. Again, the sets R and B may intersect whereas $Y \cap R = Y \cap B = \emptyset$. Observe also that the set of yellow vertices consists of all the vertices of out-degree exactly one in T except for those that are coloured blue.

3.1 PHASE I

Let T be a spanning arborescence in a directed graph $G = (V, A)$. In order to describe our local improvement algorithm the following fact will be useful.

OBSERVATION 3.1. *Given $a = (u, v) \in A$, let $\hat{a} = (x, v) \in T$ be the unique arborescence arc entering v. Then $(T - \hat{a}) \cup a$ is a spanning arborescence if and only if v is not an ancestor of u.*

Proof. First suppose v is an ancestor of u in T and let P be the path from v to u in T. The graph $(T - \hat{a}) \cup a$ has a directed cycle $P \cup \{u, v\}$, so $(T - \hat{a}) \cup a$ is not a spanning arborescence. On the other hand, if v is not an ancestor of u then clearly $(T - \hat{a}) \cup a$ is a spanning arborescence. □

This observation will be of use in describing our local improvement algorithm where we will attempt to improve our current tree by substituting a non-tree arc a for its *exchange partner* $\hat{a}$. In general, let Q be a set of arcs that form a forest. Then let $\hat{Q}$ consist of all the arcs in T that are exchange partners for arcs in Q.

The process of setting $T := (T - \hat{Q}) \cup Q$ will be called a k-exchange, where k is the number of arcs in Q, provided that it produces a valid arborescence. In particular, Phase I of the algorithm is a 1-exchange algorithm (this is similar to the method applied by Lu and Ravi on undirected graphs). Our input is any spanning arborescence T.

PHASE I: LOCAL IMPROVEMENT.

(1) [1-EXCHANGE]
While there is an arc $a \in G-T$ such that $(T-\hat{a})\cup a$ is an arborescence with more leaves than T:
(a) Set $T := (T - \hat{a}) \cup a$.

(2) [TREE-SHORTENING]
While there is a vertex u and arc $a = (u, v)$ with v a descendant of u:
(a) Set $T := (T - \hat{a}) \cup a$, where $\hat{a} = (x, v) \in T$.
(b) Recolour T.

Phase I is repeated until the tree T is unchanged throughout a whole iteration; namely, no 1-exchange or tree-shortening modifications are possible. We now show that Phase I runs in polynomial time.

LEMMA 3.2. *Phase I runs in polynomial time.*

Proof. The conditions for Steps (1) and (2) can easily be tested in polynomial time. The 1-exchange step adds an extra leaf each time it is applied, so clearly this happens at most n times. After a tree-shortening step the depth of vertex v and its descendants decrease. Therefore this step can only be carried out at most n^2 times between successive 1-exchange applications . □

We call a tree on which no Step (1) or Step (2) modifications can be applied a *short-tree*. Note that Step (2) does not change the number of leaves; otherwise it would have been carried out in Step (1). This implies the parent of v must be red since if v is the only child of its parent then the parent would become a new leaf after Step (2).

In order to see why short-trees are useful, take a maximal yellow path $P = \{p_1, \ldots, p_j\}$ in a short-tree T. By maximality, p_1 must be the descendant of a blue vertex, say p_0. We call the path $W = \{p_0, p_1, \ldots, p_j\}$ a *closed maximal yellow* path and denote the set of all closed maximal yellow paths in a short-tree T by $\mathcal{P}$. Observe that the child of p_j in T is either a red vertex or a leaf; we will refer to this vertex as r_W.

OBSERVATION 3.3. *Any closed maximal yellow path in a short-tree T induces a willow graph in G.* □

This observation will be of value as we would like to isolate these yellow paths and attack them with the willow algorithm from the previous section. This we will do in the second phase of the algorithm.

3.2 Phase II

We wish to apply our willow algorithm on the paths in $\mathcal{P}$. As we have seen, these paths induce willows in G. However, we cannot just use the willow algorithm on some $W \in \mathcal{P}$ as there may be other arcs, for example with heads in W and tails in $V - W$, that may be of use to us. Therefore, we transform our tree into a form on which the willow algorithm can be applied successfully. In particular, we will use the algorithm to try to obtain a spanning arborescence that has a "large" number of leaves from W. If we succeed on some W then we will have an improved arborescence; if we fail on every $W \in \mathcal{P}$ then we will obtain a certificate that our current tree is approximately optimal.

We now formally describe the procedure for transforming our tree. Our input is a closed maximal yellow path $W \in \mathcal{P}$ in a short-tree T such that $W \cup r_W$ contains at least one vertex which is reachable from r by a path whose interior does not intersect W. If no vertex on W satisfies this property, then any path from r to r_W must contain W and we cannot hope to find a spanning arborescence of G rooted at r with a large number of leaves in W.

To see this, let $L = \{l_0, ..., l_s\}$ be a path from r to some $p_i = l_s \in W$. Let $\hat{p} \in L \cap W$ be the last vertex in the interior of L that is in W before the interior of L leaves W for the last time. Let T_{p_0} be the subtree of T rooted at p_0. Finally let l_k be the last vertex in L that is not in W. By the tree-shortening property of T and the definition of $\hat{p}$, the vertex l following $\hat{p}$ on L must be in $T \setminus T_{p_0}$. The concatenation of the path in T from r to l with the subpath from l to l_{k+1} in L gives a path from r to l_{k+1}, a vertex in W, whose interior does not intersect W.

Therefore, we restrict our attention to closed maximal yellow paths with this property; we then define z_W to be the lowest vertex in $W \cup r_W$ that can be reached from r by a path whose interior does not intersect W. Assume $z_W = p_i$ and let $P = \{p_0, p_1, \ldots, p_{i-1}\}$ be the segment of W above p_i.

Swap(W): Willowfication

—Let z_W be the lowest vertex in $W \cup r_W$ reachable from r by a path Q whose interior does not intersect W.

—Set $T := (T - \hat{Q}) \cup Q$ provided that $(T - \hat{Q}) \cup Q$ is an arborescence. *[Note that z_W is no longer a child of p_{i-1} after this $|Q|$-exchange.]*

—Contract $T - P$ into z_W and call it $\tilde{z}$; add an arc from $p_{i-1} \in P$ to $\tilde{z}$.

—We have now built a willow $\tilde{W} = \{p_0, p_1, \ldots, p_{i-1}, \tilde{z}\}$ with root $\tilde{z} = p_i$.

Observe that, since there is a path in $T - W$ from r to p_0, there is an arc from $\tilde{z}$ to p_0 in $\tilde{W}$. Consequently, the willow $\tilde{W}$ always contains a spanning arborescence. Figure 4 illustrates the Swap procedure used by the algorithm.

The key property of the willow $\tilde{W}$ is that no spanning arborescence $\overline{T}$ of G can have more leaves in W than the optimal arborescence in $\tilde{W}$. As a first step towards proving this, we show that all the leaves of any spanning arborescence $\overline{T}$ that are

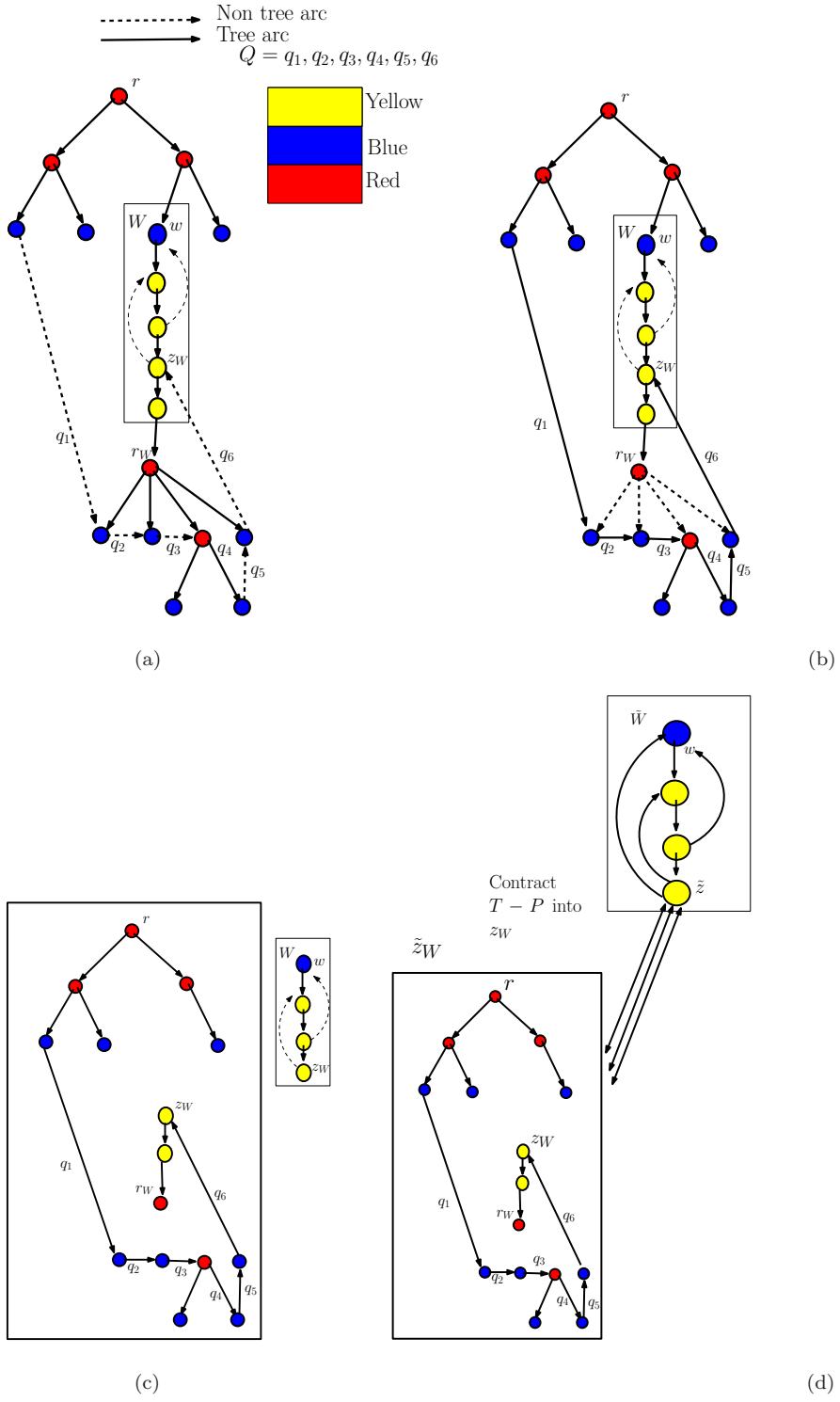

Fig. 4. (a) Find path Q to z_W. (b) Add edges of the path Q and drop the exchange partners from the tree. (c) Contract the rest of the tree $V - P$ into z_W and call it $\tilde{z}$. (d) Add an arc from $p_{i-1} \in P$ to $\tilde{z}$ creating a willow $\tilde{W}$.

in W must lie above the vertex z_W. We will need the following notation: For any closed maximal yellow path W in a short-tree T, let T_W denote the subtree of T rooted at the first vertex p_0 of W.

Notice that in the Willowification procedure $\tilde{W}$ does not contain any vertices lower than z_W in W. The next lemma shows that we need not worry about these vertices as they could not contribute to the number of leaves in any spanning tree of G which includes any optimal solution.

LEMMA 3.4. *Take a closed maximal yellow path W in a short-tree T, and let $\overline{T}$ be a spanning arborescence of G. If x is a leaf of $\overline{T}$ that lies in W then $x > z_W$.*

Proof. Let W be a closed maximal yellow path in a short-tree T. Now suppose, for a contradiction, that $x \leq z_W$ is a leaf in $\overline{T}$. Consider the path $\overline{P}$ from r to r_W in $\overline{T}$. If $r_W = z_W$ then the statement holds and there is nothing to prove. So we assume $r_W < z_W$, which implies by the definition of z_W that the interior of path $\overline{P}$ must intersect W. Let u be the last vertex of $\overline{P}$ that is not on W and let (u, v) be an arc in $\overline{P}$. Then $v \in W$ and $\overline{P}$ must end with the downpath $[v, r_W]$. Therefore, we must have $v < x \leq z_W$ because x is a leaf in $\overline{T}$.

We claim that u is a descendant of r_W in T. If not, there exists a path P' from r to u in T that does not intersect W. Thus, adding (u, v) to the path P' gives a path from r to v in G whose interior does not intersect W. Since $v < z_W$, this contradicts the definition of z_W.

Let y be the last vertex preceding u on $\overline{P}$ that is not a descendant of r_W. As T is a short-tree, we know that y is not in W because there are no arcs in G from W to the subtree of T rooted at r_W. Hence, y is not in T_W and so there is a path P'' in T from r to y whose interior does not intersect W. Then the path P'' from r to y followed by the subpath of $\overline{P}$ from y to v is a path in G from r to v whose interior doesn't intersect W. This contradicts the definition of z_W since $v < z_W$. Thus, x is not a leaf in $\overline{T}$. □

We now show no spanning arborescence $\overline{T}$ of G can have more leaves in W than the optimal arborescence in $\tilde{W}$. Therefore applying our algorithm for willow graphs on $\tilde{W}$ produces a spanning arborescence of $\tilde{W}$ which contains to within a constant factor the number of leaves from $\tilde{W}$ in an optimal solution.

LEMMA 3.5. *Let $W = \{p_0, \ldots, p_j\}$ be a closed maximal yellow path in a short-tree T. Then no spanning arborescence $\overline{T}$ of G can have more leaves in W than the optimal arborescence in $\tilde{W}$.*

Proof. We prove the result by showing that, given any spanning arborescence $\overline{T}$ of G with l leaves in W, we can construct a spanning tree $\tilde{T}$ of $\tilde{W}$ with l leaves. Let Q be the path from r to z_W in G whose interior does not intersect W. Take the short-tree T and exchange Q with its exchange partners; call this new tree $\hat{T}$. Let $z_W = p_i$ and observe that $P := \{p_0, \ldots, p_{i-1}\}$ is a path in $\hat{T}$ all of whose vertices have out-degree one except for the vertex p_{i-1}, which is a leaf.

By Lemma 3.4, all the leaves of $\overline{T}$ in W must lie on P. Note that $\overline{T}[P]$, the restriction of $\overline{T}$ to P, is a forest. Thus, since $\overline{T}$ is a spanning arborescence, there must be a set of arcs $X \subseteq \overline{T}$ from $V - P$ to the root of each component of P. In addition, $\hat{T}[V - P]$, the restriction of $\hat{T}$ to $V - P$, is a spanning arborescence of

$V - P$ because all vertices of P have degree at most one.

Add the forest $\overline{T}[P]$ and the arcs X to $\hat{T}[V - P]$ to form a new tree $T' := \hat{T}[V - P] \cup X \cup \overline{T}[P]$. Observe that leaves of $\overline{T}$ in W are leaves of T'. Form a new graph $\tilde{T}$ by contracting $\hat{T}[V - P]$ in T' to a root r' and adding an arc from the vertex $p_{i-1} \in P$ to r'. Then $\tilde{T}$ is a spanning arborescence of $\tilde{W}$ with l leaves. □

We are finally ready to describe the second phase of the algorithm. It applies the algorithm Willow(G) to the maximal yellow paths after they have been "willowficated" by the Swap procedure.

Phase II: Greedy.

(1) Given a short-tree T and closed maximal yellow path set $\mathcal{P}$.
(2) For each willow $W \in \mathcal{P}$ consider $\tilde{W}$ = Swap(W): If $\tilde{T}$ =Willow($\tilde{W}$) has more leaves than $|T_0|$ then improve T as follows:
—Un-contract $V - P$ in $\tilde{T}$, where $P = \{p_0, p_1, \ldots, p_{i-1}\}$ is the segment of W above $z_W = p_i$.

If an improved tree is found then the algorithm returns to Phase I to turn this new tree into a short-tree. If Phase II finds no improvements then the algorithm terminates.

3.3 Analysis.

We now analyse the performance guarantee of our algorithm. We begin with some simple observations. Let T be the tree returned by our algorithm. The vertices of T_1 are either blue or yellow. Any vertex $v \in T_1$ lies in a maximal path P_v whose vertices are all contained in T_1. If $P_v \neq v$, then P_v is a closed maximal yellow path; otherwise v is a blue vertex followed by a leaf or a red vertex. Let $\mathcal{M}$ be the set of all distinct maximal paths $P_v \subseteq T_1$, namely $\mathcal{M} = \{P_v \subseteq T_1 : v \in T_1\}$. Since every path P_v must end with a leaf or a red vertex, we obtain the following simple bound on the size of $\mathcal{M}$.

Observation 3.6. *We have* $|\mathcal{M}| \leq |R| + |T_0| \leq 2|T_0| - 1$. □

Lemma 3.7. *The optimal tree* T^* *has at most* $14|T_0|$ *leaves in any path* $W \in \mathcal{P}$.

Proof. Let T be the tree returned by our algorithm. Suppose, for a contradiction, that T^* contains at least $14|T_0| + 1$ leaves in $W \in \mathcal{P}$. By Lemma 3.5, the optimal spanning arborescence in $\tilde{W}$ contains at least $14|T_0| + 1$ leaves. Since Willow(G) is a 14-approximation algorithm, it returns a spanning arborescence $\tilde{T}$ of $\tilde{W}$ with at least $\lceil \frac{1}{14}(14|T_0| + 1) \rceil = |T_0| + 1$ leaves. Thus $\tilde{T}$ has more leaves than T and we would have used W to improve T in Phase II, a contradiction. □

We now put Observation 3.6 and Lemma 3.7 together to obtain our main result, an $O(\sqrt{\text{OPT}})$-approximation algorithm for the MLSA problem in general directed graphs.

THEOREM 3.8. *The algorithm is an $O(\sqrt{\text{OPT}})$-approximation algorithm.*

Proof. Partition T_0^* as

$$T_0^* = (T_0^* \cap T_0) \cup (T_0^* \cap T_1) \cup (T_0^* \cap T_{\geq 2}).$$

By Lemma 2.1, we have

$$\begin{aligned} |T_0^*| &\leq |T_0| + |T_0^* \cap T_1| + (|T_0| - 1) \\ &= 2|T_0| + |T_0^* \cap T_1| - 1. \end{aligned}$$

Hence, it suffices to bound $|T_0^* \cap T_1|$. We have

$$\begin{aligned} |T_0^* \cap T_1| &= \textstyle\sum_{P_v \in \mathcal{M}} |T_0^* \cap P_v| \\ &\leq \textstyle\sum_{P_v \in \mathcal{M}} 14|T_0| && \text{(By Lemma 3.7)} \\ &\leq (2|T_0| - 1) \cdot 14|T_0| && \text{(By Observation 3.6)} \\ &= 28|T_0|^2 - 14|T_0| \end{aligned}$$

This gives

$$\text{OPT} = |T_0^*| \leq 2|T_0| + 28|T_0|^2 - 14|T_0| \leq 28|T_0|^2,$$

which proves that our algorithm is an $O(\sqrt{\text{OPT}})$-approximation algorithm. □

4. HARDNESS

The MLST problem, and thus the MLSA problem, is MAXSNP-complete [Galbiati et al. 2004] and is NP-hard to approximate within $1 + (1/244)$ [Chlebik and Chlebikova 2004]. Clearly, there is a huge gap between this lower bound and our upper bound and closing this gap is the key open problem here.

For the weighted versions of MLST and MLSA, good approximation algorithms are unlikely to exist. These problems are at least as hard to approximate as STABLE SET. To see this, take a graph $G = (V, E)$ and construct a graph $G' = (V', E')$ as follows:

—$V' = \{r\} \cup V \cup E$.

—The node r is connected to each node corresponding to a vertex $v \in V$.

—The node $e = \{u, v\}$ is connected to the nodes corresponding to $u, v \in V$.

—A node corresponding to an edge $e \in E$ has weight 0.

—A node corresponding to a vertex $v \in V$ has weight 1.

In a spanning tree of G' rooted at r, there must be a path from r to the node corresponding to $e = \{u, v\}$; hence, it is not possible for both u and v to be leaves. The set of leaves of weight 1 in a spanning tree of G' rooted at r therefore forms a stable set in G.

ACKNOWLEDGMENTS

The work in this paper benefited from a number of discussions with Ameera Chowdhury.

REFERENCES

Alon, N., Fomin, F., Gutin, G., Krivelevich, M., and Saurabh, S. 2007a. Better algorithms and bounds for directed maximum-leaf problems. preprint.

Alon, N., Fomin, F., Gutin, G., Krivelevich, M., and Saurabh, S. 2007b. Parameterized algorithms for directed maximum leaf problems. In *Proceedings of 34th International Colloquium on Automata, Languages and Programming (ICALP 2007)*. 352–362.

Bonsma, P. and Dorn, F. 2007. An fpt algorithm for directed spanning k-leaf. preprint.

Chlebik, M. and Chlebikova, J. 2004. Approximation hardness for dominating set problems. In *Proceedings of the 12th Annual European Symposium on Algorithms, ESA 2004*. 192–203.

Galbiati, G., Maffioli, F., and Morzenti, A. 2004. A short note on the approximability of the maximum leaves spanning tree problem. *Information Processing Letters 52*, 45–49.

Guha, S. and Khuller, S. 1996. Approximation algorithms for connected dominating sets. In *European Symposium on Algorithms*. 179–193.

Lu, H. and Ravi, R. 1992. The power of local optimzation: Approximation algorithms for maximum-leaf spanning tree. In *Proceedings of the Thirtieth Annual Allerton Conference on Communication, Control and Computing*. 533–542.

Lu, H. and Ravi, R. 1998. Approximating maximum leaf spanning trees in almost linear time. *Journal of Algorithms 29,* 1, 132–141.

Payan, C., Tchuente, M., and Xuong, N. 1984. Arbes avec un nombre de maximum de sommets pendants. *Discrete Mathematics 49*, 267–273.

Solis-Oba, R. 1998. 2-approximation algorithm for finding a spanning tree with maximum number of leaves. In *European Symposium on Algorithms*. 441–452.

Storer, J. 1981. Constructing full spanning trees for cubic graphs. *Information Processing Letters 13*, 8–11.

Received August 2007; revised May 2008; accepted April 2008.