An Approximation Algorithm for the Maximum Leaf Spanning Arborescence Problem

Matthew Drescher

Master of Science

School of Computer Science

McGill University

Montreal, Quebec

October 2, 2007

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements of the degree of Master of Science

DEDICATION

I dedicate this thesis to Ameera Chowdhury, whose unwavering belief in me provided me with the courage to move to Montreal and pursue discrete mathematics.

ACKNOWLEDGEMENTS

I would like to thank Adrian Vetta for being a better supervisor than I could have ever imagined possible: He took me on as a student and put together funding and a plan when I came to his office last summer out of the blue without any of these things. None of this would have been possible without Bruce Reed's generous funding. I would also like to thank Ameera Chowdhury for her helpful discussions and advice. Thanks to the McGill Discrete Math Group at large of which I felt very privileged to be a part. Thanks to everyone in the MCFC team for all the soccer games! I would like to thank my parents who were always encouraging and supportive. Finally I would like to thank Tony and the Shash for keeping me awake!

ABSTRACT

We present an $O(\sqrt{\textsc{opt}})$-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.

ABRÉGÉ

Nous présentons une approximation algorithmique $O(\sqrt{\text{OPT}})$ pour le probléme d'envergure maximum d'arborescence oú OPT est le nombre de feuilles dans une étendue arborescente optimal. Le résultat est basé sur une approximation algorithmique $O(1)$ pour une classe spéciale de graphiques dirigés appelés saules. En incorporant la méthode des graphiques de saules comme une sous-routine dans une amélioration algorithmique locale nous donne le lien des graphiques dirigée généraux.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1
Introduction

The vertices of any spanning tree can be naturally partitioned into two types: leaves and internal vertices. In many applications, the leaves of a tree hold special significance. For example, in communication networks the leaves of a tree represent the receivers. A natural question then is to find a spanning tree T in a connected undirected graph G containing the maximum number of leaves. This is known as the *Maximum Leaf Spanning Tree* (MLST) problem and has applications in communication network design [8], circuit layouts [14], and distributed systems [12]. This problem, however, is hard; Galbiati et al [7] proved it to be MAXSNP-complete.

Given this, there has been much work on designing approximation algorithms. Interestingly, the MLST problem is one of those select problems for which many of the standard tools give good approximation guarantees.

The primary focus of this thesis is the *Maximum Leaf Spanning Arborescence* (MLSA) problem: Given a directed graph G, whose underlying undirected graph is connected, find a spanning arborescence T containing the maximum number of leaves. Our new contributions include a $\sqrt{\text{OPT}}$ factor approximation algorithm where OPT is the number of leaves in an optimal spanning arborescence. To the best of our knowledge this is the first nontrivial approximation algorithm for the problem.

We begin by reviewing some of the more successful approximation algorithms for the undirected MLST problem in Chapter 2. Our treatment of the undirected algorithms is by no means an in-depth study, but serves to highlight concepts and ideas from the undirected case that influenced our approach in the directed case. The material in Chapter 2 is all previous work and contains no original results.

In Chapter 3 we turn to the MLSA problem. Chapter 3 contains work that is joint with Adrian Vetta and presents the following new material:

1. We introduce *willows*, a special family of directed graphs for which MLSA has a constant factor approximation algorithm.
2. Using the constant factor algorithm for willows as a subroutine, we develop a $O\sqrt{\textsc{opt}}$ factor algorithm for the MLSA problem in general.

CHAPTER 2
Undirected Background

In this chapter, we look at some of the techniques that proved successful in the undirected version of the problem. We do so through a "directed lens" and point out notions that have had an impact on our approach for the directed problem.

We first review some basic graph theoretic terminology. A *tree* is a graph in which any two vertices are connected by exactly one path. An *arborescence* is an oriented tree in which all vertices are reachable via a directed path from a distinguished vertex called the root. A *spanning tree* of a connected undirected graph G is a subgraph of G that is a tree and that contains all vertices of G [6]. Similarly, a *spanning arborescence* of a directed graph D is a subgraph of D that is an arborescence and that contains all the vertices of D [9].

There are many polynomial time algorithms, such as Kruskal's algorithm and Prim's algorithm, to find a spanning tree in a graph [5]. As we mentioned in Chapter 1, however, the problem of finding a spanning tree in a graph with the maximum number of leaves is MAXSNP-complete [7], which means that it is unlikely that a polynomial time algorithm for this problem exists. We can cope with the seeming intractability of the MLST problem by designing an *approximation algorithm* [15], a polynomial time algorithm that finds a spanning tree of G whose number of leaves is a guaranteed fraction of the number of leaves in the optimal solution. We now turn to some of the techniques that were useful in developing good approximation algorithms for the MLST problem.

2.1 Local Search

The very natural technique of Local Search can be described generically as follows:

```
1. Start with any feasible solution $F$.
2. While possible:
    (a) Perform some *local* modification of $F$ which gives a new *improved* solution $F'$.
    (b) Set $F := F'$.
```

This technique does not work for every problem; for example, it fails on the *graph coloring* problem. Lu and Ravi, however, show in [10] that, in the particular case of the MLST problem, application of local search gives a constant factor approximation algorithm.

To apply the above generic algorithm we need only specify how to find an initial feasible solution and how to improve locally. A variety of efficient algorithms exist for finding a spanning tree so we turn our attention to the problem of local improvement. Lu and Ravi observed the following: If T is a spanning tree of $G = (V, E)$ and $e \in E(G) - E(T)$ then $e \cup T$ contains a unique cycle C_e^T. Removing any edge $f \in C_e^T$ from $T \cup e$, where $e \neq f$, gives a new spanning tree T'. For a spanning tree T, let $L(T)$ denote the number of leaves of T. We define the following operation:

```
**Swap(T)**
  1. Find any $e \in E(G) - E(T)$ such that there exists $f \in C_e^T$ with the property that $L((T - f) \cup e) > L(T)$.
  2. If found return $(T - f) \cup e$; otherwise return $T$.
```

This operation leads directly to the local search algorithm.

```
**Local Search Algorithm**
  1. Find a spanning tree $T$ of $G$.
  2. While ($T \neq$ Swap($T$))
      (a) Set $T :=$ Swap($T$).
```

In [10], Lu and Ravi illustrate an example for which no single edge swap leads to an improvement, but a sequence of edge swaps does. We then have the following generalization of the local search algorithm.

```
$k$-**Swap(T)**
  1. Find a subset $H \subseteq E(G) - E(T)$ and a subset $F \subseteq E(T)$ that satisfy $(T - F) \cup H$ is a spanning tree of $G$, $L((T - F) \cup H) > L(T)$, and $|H| = |F| \leq k$.
  2. If found return $(T - F) \cup H$; otherwise return $T$
```

For each increasing value of k, this defines an increasingly complex local search algorithm.

```
$k$-Local Search Algorithm [LSA($k$)]
    1. Find a spanning tree $T$ of $G$.
    2. While ($T \neq k$-Swap($T$))
        (a) Set $T := k$-Swap($T$).
```

Lu and Ravi prove that the local search and k-local search algorithms give constant factor approximation guarantees in [10].

Theorem 2.1.1. *LSA*(1) *is a factor* 5 *approximation algorithm.* $\square$

Theorem 2.1.2. *LSA*(2) *is a factor* 3 *approximation algorithm.* $\square$

We will not give the proofs here; rather we shall illustrate some of the ideas by proving the weaker but easier result, Theorem 2.1.5, which shows that LSA(1) is a factor 10 approximation algorithm. Theorem 2.1.5 is due to Lu and Ravi [10]. We will need the following notation: For a tree T and $i \in \{0, 1, 2, \ldots\}$, let T_i denote the set of vertices of T with degree i; similarly let $T_{\geq i}$ denote the set of vertices of T with degree at least i. We let T^* denote the optimal solution. Before commencing with the proof, we develop some lemmas.

Lemma 2.1.3. *If T is a spanning tree then $T_{\geq 3} \leq |T_1| - 2$.* $\square$

Let T be a spanning tree of G. Note that the vertices of T_2 naturally decompose into a set of maximal paths which we denote by $\mathcal{P}_2$.

Lemma 2.1.4. $|\mathcal{P}_2| \leq 2|T_1|$.

Proof. Each path in $\mathcal{P}$ is terminated by either a vertex from $T_{\geq 3}$ or from T_1. $|\mathcal{P}_2| \leq |T_{\geq 3}| + |T_1|$ and the result follows by Lemma 2.1.3. □

Theorem 2.1.5. *Suppose LSA(1) returns the tree T. Then $|T_1| \geq (1/10)|T_1^*|$.*

Proof. We have

$$\begin{aligned} |T_1^*| &= |T_1^* \cap T_1| \cup |T_1^* \cap T_2| \cup |T_1^* \cap T_{\geq 3}| \\ &\leq |T_1| + |(T_1^* \cap T_2)| + |T_{\geq 3}| \\ &\leq 2|T_1| + |(T_1^* \cap T_2)|. && \text{(by Lemma 2.1.3)} \end{aligned}$$

Thus we need only concern ourselves with showing $|(T_1^* \cap T_2)| \leq 8|T_1|$. Since $T_2 = \cup_{P \in \mathcal{P}_2} P$, we have by Lemma 2.1.4 that

$$|(T_1^* \cap T_2)| \leq \max_{P \in \mathcal{P}_2}\{|P \cap T_1^*|\} \cdot |\mathcal{P}_2| \leq \max_{P \in \mathcal{P}_2}\{|P \cap T_1^*|\} \cdot 2|T_1|.$$

Therefore, we need only show that each $P \in \mathcal{P}_2$ can have at most four vertices which are leaves in T_1^*.

Suppose for a contradiction that a path $P \in \mathcal{P}_2$ contains more than four leaves of T_1^*. In particular, suppose P contains five leaves $v_1, v_2, v, v_1', v_2' \in P \cap T_1^*$ in this order. Let x' be a vertex not in P. Let x be the first vertex in $V - P$ on the path $_{T^*}(v, x')$, and let w be the last vertex from P on path $_{T^*}(v, x)$. Observe that path$_T(v, w)$ =path$_{T^*}(v, w)$ by the choice of w. Since v_2 and v_1' are leaves in T^* they cannot be part of path$_{T^*}(v, x')$. Thus path$_{T^*}(v, x')$ has to leave P before either v_2 or v_1'. Hence w lies between v_2 and v_1' on path P. Now, the edge $\{w, x\}$ in T^* does not lie in T so adding it to T creates a cycle $C^T_{\{w,x\}}$. Since

w lies between v_2 and v_1' on P, either v_1 and v_2 or v_1' and v_2' are in the cycle $C^T_{\{w,x\}}$. Without loss of generality, suppose v_1 and v_2 lie in $C^T_{\{w,x\}}$: This implies $T \cup (x, w) - e$ is a local improvement where e is an edge on P between v_1 and v_2, which contradicts the definition of T. $\square$

Before we conclude, we ask the reader to pause and consider the MLSA problem. Certainly the k-Local Search Algorithm can be modified for this problem. What structure would the output of such an algorithm have? For what value of k could such an algorithm have a good if any approximation guarantee?

Figure 2.1 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.

2.2 The Greedy Approach

In [11] Lu and Ravi employ a new greedy strategy obtaining a factor 3, near linear time approximation algorithm for the MLST problem. Solis-Oba then improves on this result and gives a factor 2 approximation algorithm. This algorithm is based on a greedy strategy which follows *expansion rules* to grow a spanning forest with many leaves that can be extended to a spanning tree at the cost of a few leaves.

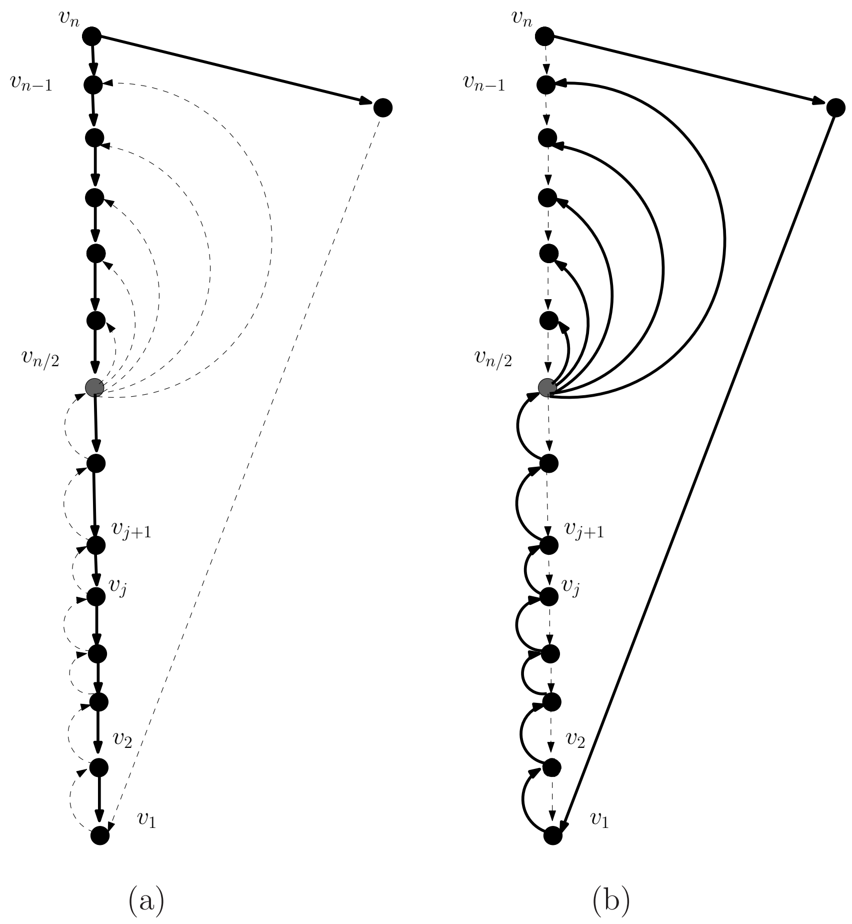

Figure 2–1: (a) Initial spanning arborescence with 2 leaves. (b) After an arbitrarily high number of edge swaps we have a new spanning arborescence with an arbitrarily high number of leaves.

We shall not go into full detail here. However, in order to provide the reader with some intuition and to illustrate this more successful approach, we shall state the algorithm and then give the much shorter proof that it is a factor 3 algorithm.

2.2.1 Expansion Rules

Given a sub-tree T, we apply a set of rules to the leaves of T in order to obtain an expanded tree T'. Let x be a leaf in T.

1. If x has at least 2 neighbors outside of T, then add the neighbors of x as x's children in the extended tree T'.
2. If x has only one neighbor y outside of T but at least 2 neighbors of y are outside T then add y as x's child along with y's neighbors as y's children in the extended tree T'

The above rules are assigned priorities as follows:

- Rule 1 has priority over rule 2: If x and x' are leaves and x, x' can be expanded using rules 1 and 2 respectively, then expand x.
- If x and x' can both be expanded by rule 1, choose the vertex with more neighbors outside T.
- If x and x' with single neighbors y, y' in $G - T$ respectively can be expanded by rule 2, choose the vertex whose single neighbor (y or y') has more neighbors outside of T.

We are now in a position to describe the algorithm.

```
**Greedy Expansion(G)**
  1. Set $F = \emptyset$.
  2. While there remains a vertex $v \in G$ of degree at least 3:
      (a) Let $T^i$ be the tree with root $v$ and the neighbors of $v$ in $G$ as leaves.
      (b) While there is some leaf in $T^i$ that can be expanded:
            i. Expand a leaf with highest priority rule.
  3. Set $F = F \cup T^i$ and $G = G - T^i$.
  4. Connect the trees in $F$ and the vertices not in $F$ to obtain the spanning tree $T$.
```

2.2.2 Analysis

Suppose $F = \{T^0, T^1, \dots, T^k\}$ is the forest returned by the algorithm. Let $X = G - F$ be the vertices not spanned by F, which we will refer to as *exterior vertices*. We will need the following lemmas.

Lemma 2.2.1. *Let G' denote the graph obtained by contracting each tree in F to a single vertex. Each exterior vertex has degree at most two in G'.*

Proof. Suppose, for a contradiction, that there is an exterior vertex x with three neighbors in G'. At least one of these neighbors must be in F; otherwise the algorithm would have continued with x as the root of a new tree. Let $T^j \in F$ be the *first* tree added to F that contains a neighbor y of x. Clearly, y cannot be an internal vertex; otherwise x would have been added as a child of y. However, x is adjacent to two vertices outside T^j so y would have been expanded by rule 2. $\square$

A similar argument proves the following lemma.

Lemma 2.2.2. *Let u be a leaf of $T^i \in F$. If u is adjacent to two vertices $v, w \notin T^i$, then v and w are leaves of the same tree $T^j \in F$.* $\square$

The following corollary applies to any spanning tree of G and, in particular, to an optimal spanning tree.

Corollary 2.2.3. *If $\overline{T}$ is any spanning tree of G, then $\overline{T}$ has at most $|V(F)| - 2k$ leaves, where $F = \{T^0, T^1, \ldots, T^k\}$ is the forest returned by the algorithm.*

Proof. Suppose $\overline{T}$ is a spanning tree of G. Let $\overline{F} = \overline{T}[F]$ be the forest induced on $\overline{T}$ by F. In order to extend $\overline{F}$ to a spanning tree we must connect the trees in $\overline{F}$. Either we must connect two leaves of F by an edge or we must connect two leaves of F by a path through X, the set of exterior vertices. Lemma 2.2.2 and Lemma 2.2.1 therefore imply that we will lose at least $2k$ leaves of F in the process. Clearly, each leaf in X must have a unique parent p that is a leaf in F'; otherwise p would have been expanded using rule 1. $\square$

We call a vertex y in F *black* if it has 2 children and is the only child of its parent x. In other words x was expanded with the lowest priority instance of rule 2. Let $B(V')$ be the set of black vertices in $V' \subseteq V$.

Lemma 2.2.4. *For $T^i \in F$, $|T^i_1| \geq 3 + |B(T^i)| + (1/2)(|T^i| - 3|B(T^i)| - 4)$.*

Proof. By construction T^i's root has at least 3 children. Consider the application of the expansion rules when growing the tree. Notice that, when we use the least priority rule, we kill one leaf and add two leaves for a net increase of one leaf. Since we use the least priority rule $|B(T^i)|$ times, this corresponds to $|B(T^i)|$ leaves that were contributed by the least priority rule. Also note that the total number of vertices added by this rule is $3|B(T^i)|$. Every other expansion type clearly adds at least as many leaves as internal vertices. Excluding the root, its children, and the vertices added by the lowest priority rule, we see there are $|T^i| - 3|B(T^i)| - 4$ vertices added by these other expansion types. Hence, we add at least $(1/2)(|T^i| - 3|B(T^i)| - 4)$ leaves. □

We now show that the greedy expansion algorithm is a factor 3 algorithm:

Theorem 2.2.5. *If T is the spanning tree returned by Greedy Expansion(G) then $|T_1^*| < 3|T_1|$.*

Proof. Applying Corollary 2.2.3 and Lemma 2.2.4 we have

$$\begin{aligned}\frac{|T_1^*|}{|T_1|} &\leq \frac{|V(F)| - 2k}{\sum_{i=0}^{k}(3 + |B(T^i)| + \frac{|V(T^i)|-3|B(T^i)|-4}{2}) - 2k} \\ &\leq \frac{2(|V(F)| - 2k)}{|V(F)| - |B(F)| - 2k + 2} \\ &\leq 2 + \frac{2|B(F)|}{|V(F)| - |B(F)| - 2k} \qquad (2.1)\end{aligned}$$

To finish up the proof we observe that $|V(F)| > 4k + 3|B(F)|$. To see this, note that each root in F and its at least three children contribute a total of at least four vertices. Since each application of the lowest priority rule adds three vertices, we have

$$|V(F)| = \sum_{i=0}^{k} |V(T^i)| \geq \sum_{i=0}^{k}(4 + 3|B(T^i)|) = 4(k+1) + 3|B(F)|.$$

Substituting this into 2.1 finishes the proof:

$$\frac{|T_1^*|}{|T_1|} \leq 2 + \frac{2|B(F)|}{|V(F)| - |B(F)| - 2k} \leq 3. \quad \square$$

Solis-Oba does a more intricate case analysis of the algorithm and shows:

Theorem 2.2.6. *If T is the spanning tree returned by Greedy Expansion(G) then* $|T_1^*| \leq 2|T_1|$. □

Again, we ask the reader to pause and consider modifying this algorithm for the directed MLSA problem. What if anything about this approach can work when we have directed edges? Are there certain subclasses of directed for which it will be more successful?

It is easy to construct bad examples for all obvious types of greedy algorithms. For example, greedily growing a forest will fail as the arc directions may then prohibit the forest from being connected up efficiently.

Our approach to MLSA is motivated by the example in Figure 2.1. 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 2.1. We first present a constant factor approximation algorithm for the MLSA problem in willow graphs.

The key observation then is that paths in an arborescence T in a general directed graph must induce willow-like graphs, otherwise local improvements to T can be obtained easily. Consequently, we are able to apply the algorithm for willow graphs as a subroutine in a $O(\sqrt{\text{OPT}})$-approximation algorithm for the

MLSA problem in general directed graphs, where OPT is the number of leaves in an optimal spanning arborescence.

CHAPTER 3
The Directed Case

3.1 An Approximation Algorithm for Willow Graphs

We can not proceed without some terminology. Let T be an arborescence and let $i \in \mathbb{N}$. 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 begin with 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.

We say a directed graph G is *strongly connected* if there is a directed path between every ordered pair of vertices. A *strong component* of G is a maximal strongly connected subgraph of G. A *source strong component* is a strong component C of G such that there is no arc from $G - C$ to C.

It is possible to test in polynomial time for the existence of a spanning arborescence in a directed graph G. Decomposing a directed graph $G = (V, E)$ into its strongly connected components can be accomplished in time $O(|V| + |E|)$, an application of the classic *depth first search* algorithm see for example [5]. Therefore the following theorem gives one such method [2].

Theorem 3.1.1 ([3]). *A directed graph G has a spanning arborescence if and only if G has a unique source strong component.*

Hence we assume from now on that any directed graph G which we consider contains a spanning arborescence.

3.1.1 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 our bad example for the k-exchange algorithm contained an induced willow. Consequently, being able to deal with willows is a necessary attribute of any good algorithm. Conversely, we will show in Section 3.2 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.

3.1.2 Pitchforks.

A key 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 need 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 disjoint from the handle.

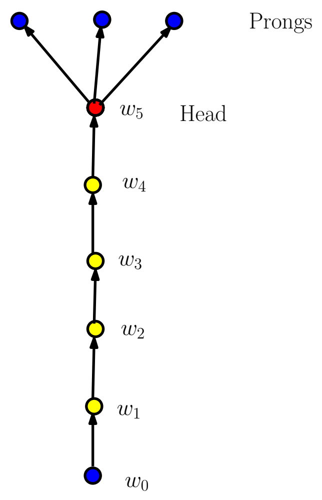

Figure 3–1: A 3-prong pitchfork with a 5-arc handle.

Figure 3–1 illustrates a pitchfork. To understand why pitchforks will be useful, consider the following simple result.

Lemma 3.1.2. *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 will have a large number of leaves. This observation is the motivation behind 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 its other vertices lying outside the current arborescence. Intuitively, if the handles of the pitchforks are small then the constructed arborescence will have few internal nodes of degree one and so will have a large number of leaves by Lemma 3.1.2.

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.

3.1.3 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 the case of 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. Our colouring is

- The head of a pitchfork is coloured *red*.
- The internal vertices on the handle are coloured *yellow*.
- The prong vertices are coloured *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 still a leaf at the end of the algorithm, then this downpath must, at some point in the algorithm, have been be 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 3-2 illustrates an example.

```
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 mini-
          mized. [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 $T := T \cup F_t$ and colour $F_t$. Colour the head $h_t$ *red*, the internal han-
          dle 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 $P$ from $T$ through every uncoloured vertex.
            • On $P$, colour the last vertex *blue* and internal vertices *yellow*.
            • Set $K := K \cup v_n$.
 5. TERMINATE if all vertices are coloured; otherwise set $t := t + 1$, and goto (2).
```

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

Observation 3.1.3. *The yellow vertices are those vertices that are not blue and have exactly one uncoloured out-neighbor when they are added to the tree.*

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.

Observation 3.1.4. *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 need to show the algorithm outputs a spanning arborescence, runs in polynomial time, and has a constant factor worst-case guarantee.

Lemma 3.1.5. *The algorithm produces a spanning arborescence.*

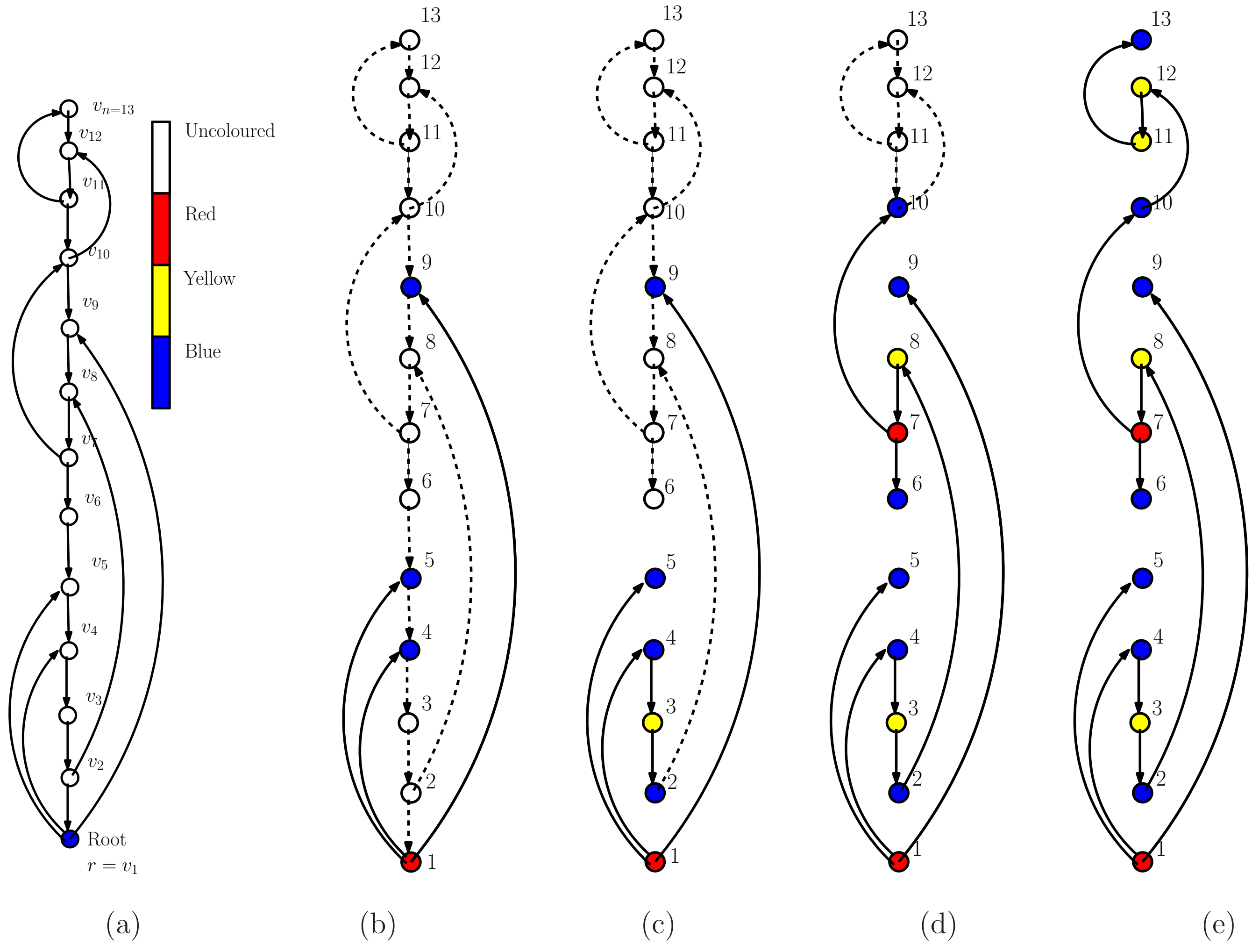

Figure 3–2: (a) Time $t = 0$ and the willow W. (b) At time $t = 1$, we have $I_2 = [v_2, v_{13}]$ and add the pitchfork with head v_1 and prongs v_4, v_5, v_9; $K = \{v_1\}$. (c) At time $t = 2$, we have $I_2 = [v_2, v_3]$ and $\mathcal{P}_2 = \emptyset$ so the algorithm adds the downpath; $K = \{v_1, v_3\}$. (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 $P = \{v_{10}, v_{12}, v_{11}, v_{13}\}$; $K = \{v_1, v_3, v_8, v_{13}\}$.

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}. Then $P = P_1 \cup P_2$ is a path through all the uncoloured vertices, as required. □

Lemma 3.1.6. *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 3.1.5 □

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 3.1.7. *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 3.1.2.

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. To see this, note that in a pitchfork the red head is immediatly followed by a blue prong. It follows that consecutive blue nodes in the arborescence are separated by a red vertex unless they were added in a downpath. Recall, however, that there can be no 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 3.1.2 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 $k_i < k_j$.

Theorem 3.1.8. *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 $y_1 < y_2$. 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 3.1.4 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 3.1.3, 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 $t(j)$. Since the vertices on P_1 after w and up to h are uncoloured at time $t(j)$ and are lower than k_j, 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 these results together gives a worst case guarantee for willow graphs.

Theorem 3.1.9. *The algorithm gives a factor* 14 *approximation guarantee for willow graphs.*

Proof. We have that

$$\begin{aligned}
|T_0^*| &\leq |(B \cap T_0^*)| + |R \cap T_0^*| + |Y \cap T_0^*| \\
&\leq |B| + |R| + |Y \cap T_0^*| \\
&\leq |B| + |R| + (2|B| + |R|) && \text{(by Theorem 3.1.8)} \\
&= 3|B| + 2|R| \\
&< 14|T_0|. && \text{(by the proof of Lemma 3.1.7)}
\end{aligned}$$

Therefore, we have a factor 14 approximation algorithm for willow graphs. $\square$

3.2 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 upto 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.

3.2.1 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 3.2.1. *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 3.2.2. *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 . $\square$

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 $\mathcal{P}$.

Observation 3.2.3. *Any closed maximal yellow path in a short-tree T induces a willow graph in G.* $\square$

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.

3.2.2 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 that 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 we cannot hope to find a spanning arborescence of G with a large number of leaves in 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 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 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 3–3 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 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.

Lemma 3.2.4. *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 $W = \{p_0, \ldots p_j\}$ be a closed maximal yellow path in a short-tree T. Observe that the child of p_j in T is either a red vertex or a leaf; we call this vertex r_W. Now suppose, for a contradiction, that $x \leq z_W$ is a leaf in $\overline{T}$. Consider the

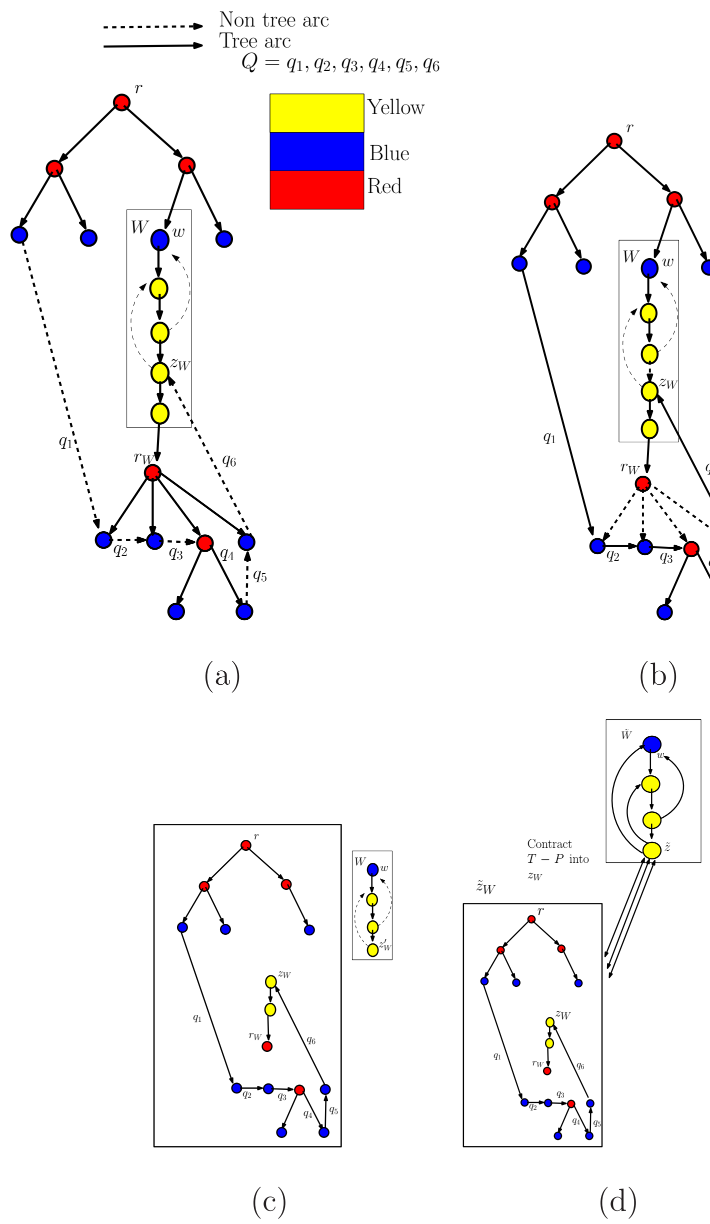

Figure 3–3: (a) Find path Q to z_W. (b) Add edges of the path Q and drop the exchange partners from the tree. (c) Contract the rest of the tree $V - P$ into z_W and call it $\tilde{z}$. (d) The arc from $p_{i-1} \in P$ to $\tilde{z}$ creates willow $\tilde{W}$.

path $\overline{P}$ from r to r_W in $\overline{T}$. Since $r_W < z_W$, 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 $\tilde{W}$.

Lemma 3.2.5. *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 3.2.4, 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 $W \in \mathcal{P}$, 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.

3.2.3 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 3.2.6. *The cardinality of $\mathcal{M}$ satisfies $|\mathcal{M}| \leq |R| + |T_0| \leq 2|T_0| - 1$.* $\square$

Lemma 3.2.7. *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 3.2.5, 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. $\square$

We now obtain our approximation guarantee.

Theorem 3.2.8. *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 3.1.2, 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

$$\begin{aligned} |T_0^* \cap T_1| &= \sum_{P_v \in \mathcal{M}} |T_0^* \cap P_v| \\ &\leq \sum_{P_v \in \mathcal{M}} 14|T_0| && \text{(By Lemma 3.2.7)} \\ &\leq (2|T_0| - 1) \cdot 14|T_0| && \text{(By Observation 3.2.6)} \\ &= 28|T_0|^2 - 14|T_0| \end{aligned}$$

This gives

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

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

REFERENCES

[1] N. Alon, F. Fomin, G. Gutin, M. Krivelevich and S. Saurabh, "Better Algorithms and Bounds for Directed Maximum-Leaf Problems," preprint, 2007.

[2] N. Alon, F. Fomin, G. Gutin, M. Krivelevich and S. Saurabh, "Parametrized algorithms for directed maximum leaf problems", preprint, 2007.

[3] J. Bang-Jensen and G.Gutin, "Digraphs: Theory, Algorithms and Applications.", *Springer-Verlag*, 2000.

[4] M. Chlebik and J. Chlebikova, "Approximation Hardness for Dominating Set Problems," *Algorithms - ESA 2004*, pp192-203, 2004.

[5] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, "Introduction to Algorithms", *The MIT Press*, 2001.

[6] R.Diestel, "Graph Theory",*Springer* , 2006.

[7] G. Galbiati, F. Maffioli, and A. Morzenti, "A Short Note on the Approximability of The Maximum Leaves Spanning Tree Problem", *Information Processing Letters*, **52**, pp45-49, 1994.

[8] S. Guha and S. Khuller, "Approximation Algorithms for Connected Dominating Sets," *Proceedings of 4th Annual European Symposium on Algorithms*, pp179-193, 1996.

[9] B.Korte and J.Vygen, "Combinatorial Optimization, Theory and Algorithms" *Springer*, 2006.

[10] H. Lu and R. Ravi, "The Power of Local Optimzation: Approximation Algorithms for Maximum-Leaf Spanning Tree", *Proceedings of the Thirtieth Annual Allerton Conference on Communication, Control and Computing*, pp533-542, 1992

[11] H. Lu and R. Ravi, “Approximating Maximum Leaf Spanning Trees in Almost Linear Time”, *Journal of Algorithms*, **29(1)** pp132-141, 1998.

[12] C. Payan, M. Tchuente, and N.H. Xuong, “Arbes avec un nombre de maximum de sommets pendants,” *Discrete Mathematics*, **49** pp267-273, 1984

[13] R. Solis-Oba, “2-Approximation algorithm for finding a spanning tree with maximum number of leaves”, *Proceedings of the 6th Annual European Symposium on Algorithms (ESA)*, LNCS **1461**, pp. 441-452, 1998.

[14] J.A. Storer, “Constructing full spanning trees for cubic graphs,” *Information Processing Letters* **13** pp8-11, 1981.

[15] V.Vazirani, “Approximation Algorithms” *Springer*, 2003.