Volume 76 Issue 7 November 2010 ISSN 0022-0000
ELSEVIER
JOURNAL OF COMPUTER AND SYSTEM SCIENCES
Managing Editor
E. K. Blum
Editors
Alfred V. Aho
Ronald Fagin
J. Hartmanis
John Hopcroft
R. M. Karp
D. E. Knuth
Raymond E. Miller
Christos Papadimitriou
Michael Rabin
Hartley Rogers, Jr.
Dana Scott
J. D. Ullman
L. A. Zadeh
Available online at
ScienceDirect
www.sciencedirect.com

Journal of Computer and System Sciences 76 (2010) 650–662

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

Algorithm for finding k-vertex out-trees and its application to k-internal out-branching problem [☆]

Nathann Cohen [a], Fedor V. Fomin [b], Gregory Gutin [c,*], Eun Jung Kim [c], Saket Saurabh [b], Anders Yeo [c]

[a] INRIA – Projet MASCOTTE, 2004 route des Lucioles, BP 93 F-06902, Sophia Antipolis Cedex, France
[b] Department of Informatics, University of Bergen, POB 7803, 5020 Bergen, Norway
[c] Department of Computer Science, Royal Holloway, University of London, Egham, Surrey TW20 0EX, UK

A R T I C L E   I N F O

Article history:
Received 22 May 2009
Received in revised form 20 December 2009
Available online 25 January 2010

Keywords:
Out-tree
Out-branching
Internal vertices
Divide-and-conquer

A B S T R A C T

An out-tree T is an oriented tree with only one vertex of in-degree zero. A vertex x of T is internal if its out-degree is positive. We design randomized and deterministic algorithms for deciding whether an input digraph contains a given out-tree with k vertices. The algorithms are of running time $O^*(5.704^k)$ and $O^*(6.14^k)$, respectively. We apply the deterministic algorithm to obtain a deterministic algorithm of runtime $O^*(c^k)$, where c is a constant, for deciding whether an input digraph contains a spanning out-tree with at least k internal vertices. This answers in affirmative a question of Gutin, Razgon and Kim (Proc. AAIM'08) [9].

© 2010 Elsevier Inc. All rights reserved.

1. Introduction

An *out-tree* is an oriented tree with only one vertex of in-degree zero called the *root*. The k-Out-Tree problem is the problem of deciding for a given parameter k, whether an input digraph contains a given out-tree with $k \geqslant 2$ vertices. In their seminal work on Color Coding Alon, Yuster, and Zwick [1] provided fixed-parameter tractable (FPT) randomized and deterministic algorithms for k-Out-Tree. While Alon, Yuster, and Zwick [1] only stated that their algorithms are of runtime $O(2^{O(k)}n)$, however, it is easy to see (see Subsection 2.1), that their randomized and deterministic algorithms are of complexity[1] $O^*((4e)^k)$ and $O^*(c^k)$, where $c \geqslant 4e$.

The main results of [1], however, were a new algorithmic approach called Color Coding and a randomized $O^*((2e)^k)$ algorithm for deciding whether a digraph contains a path with k vertices (the k-Path problem). Chen et al. [4] proposed another approach, a randomized divide-and-conquer technique; the new approach allowed them to design a randomized $O^*(4^k)$-time algorithm for k-Path. To divide the technique of Chen et al. [4] uses two colors. The colors are 'symmetric', i.e., both colors play similar role and the probability of coloring each vertex in one of the colors is 0.5. In this paper, we further develop the technique of [4] by making it asymmetric, i.e., the two colors play different roles and the probability of coloring each vertex in one of the colors depends on the color. As a result, we refine the result of Alon, Yuster, and Zwick by obtaining randomized and deterministic algorithms for k-Out-Tree, of runtime $O^*(5.704^k)$ and $O^*(6.14^k)$, respectively.

It is worth to mention here two recent related results on k-Path due to Koutis [10] and Williams [16] based on an algebraic approach. Koutis [10] obtained a randomized $O^*(2^{3k/2})$-time algorithm for k-Path and Williams [16] extended his

☆ An extended abstract of this paper has appeared in the proceedings of COCOON'09 Cohen et al. (2009) [6].
* Corresponding author.
E-mail addresses: nathann.cohen@sophia.inria.fr (N. Cohen), fedor.fomin@ii.uib.no (F.V. Fomin), gutin@cs.rhul.ac.uk (G. Gutin), eunjung@cs.rhul.ac.uk (E.J. Kim), saket.saurabh@ii.uib.no (S. Saurabh), anders@cs.rhul.ac.uk (A. Yeo).
[1] In this paper we often use the notation $O^*(f(k))$ instead of $f(k)(kn)^{O(1)}$, i.e., O^* hides not only constants, but also polynomial coefficients.

0022-0000/$ – see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jcss.2010.01.001

Algorithm 1 $\mathcal{L}(T, r)$.

Require: A digraph D with a given coloring $c : V(D) \to \{1, \ldots, k\}$, an out-tree T on k vertices, a specified vertex r of D.
Ensure: $\mathcal{C}_T(u)$ for each vertex u of D, which is a family of all color sets that appear on colorful copies of T in D, where u plays the role of r.

```
1: if $|V(T)| = 1$ then
2:    for all $u \in V(D)$ do
3:       Insert $\{c(u)\}$ into $\mathcal{C}_T(u)$.
4:    end for
5:    Return $\mathcal{C}_T(u)$ for each vertex $u$ of $D$.
6: else
7:    Choose an arc $(r', r'') \in A(T)$.
8:    Let $T'$ and $T''$ be the subtrees of $T$ obtained by deleting $(r', r'')$, where $T'$ and $T''$ contains $r'$ and $r''$, respectively.
9:    Call $\mathcal{L}(T', r')$.
10:   Call $\mathcal{L}(T'', r'')$.
11:   for all $u \in V(D)$ do
12:      Compose the family of color sets $\mathcal{C}_T(u)$ as follows:
13:      for all $(u, v) \in A(D)$ do
14:         for all $C' \in \mathcal{C}_{T'}(u)$ and $C'' \in \mathcal{C}_{T''}(v)$ do
15:            $C := C' \cup C''$ if $C' \cap C'' = \emptyset$
16:            Insert $C$ into $\mathcal{C}_T(u)$.
17:         end for
18:      end for
19:   end for
20:   Return $\mathcal{C}_T(u)$ for each vertex $u$ of $D$.
21: end if
```

ideas resulting in a randomized $O^*(2^k)$-time algorithm for k-PATH. While the randomized algorithms based on Color Coding and Divide-and-Color are not difficult to derandomize, it is not the case for the algorithms of Koutis [10] and Williams [16]. Thus, it is unknown whether there are deterministic algorithms for k-PATH of runtime $O^*(2^{3k/2})$. Moreover, it is not clear whether the randomized algorithms of Koutis [10] and Williams [16] can be extended to solve k-OUT-TREE.

While we believe that the study of fast algorithms for k-OUT-TREE is a problem interesting on its own, we provide an application of our deterministic algorithm. The vertices of an out-tree T of out-degree zero (nonzero) are *leaves* (*internal vertices*) of T. An *out-branching* of a digraph D is a spanning subgraph of D which is an out-tree. The MINIMUM LEAF problem is to find an out-branching with the minimum number of leaves in a given digraph D. This problem is of interest in database systems [7] and the Hamilton path problem is its special case. Thus, in particular, MINIMUM LEAF is NP-hard. In this paper we will study the following parameterized version of MINIMUM LEAF: given a digraph D and a parameter k, decide whether D has an out-branching with at least k internal vertices. This problem denoted k-INT-OUT-BRANCHING was studied for symmetric digraphs (i.e., undirected graphs) by Prieto and Sloper [14,15] and for all digraphs by Gutin et al. [9]. Gutin et al. [9] obtained an algorithm of runtime $O^*(2^{O(k \log k)})$ for k-INT-OUT-BRANCHING and asked whether the problem admits an algorithm of runtime $O^*(2^{O(k)})$. Note that no such algorithm has been known even for the case of symmetric digraphs [14,15]. In this paper, we obtain an $O^*(2^{O(k)})$-time algorithm for k-INT-OUT-BRANCHING using our deterministic algorithm for k-OUT-TREE and an out-tree generation algorithm.

For a set X of vertices of a subgraph H of a digraph D, $N_H^+(X)$ and $N_H^-(X)$ denote the sets of out-neighbors and in-neighbors of vertices of X in H, respectively. Sometimes, when a set has a single element, we will not distinguish between the set and its element. In particular, when H is an out-tree and x is a vertex of H which is not its root, the unique in-neighbor of x is denoted by $N_H^-(x)$. For an out-tree T, $\mathrm{Leaf}(T)$ denotes the set of leaves in T and $\mathrm{Int}(T) = V(T) - \mathrm{Leaf}(T)$ stands for the set of internal vertices of T.

2. Algorithms for k-OUT-TREE

In Subsection 2.1, we introduce and analyze the randomized algorithm for k-OUT-TREE by Alon, Yuster and Zwick [1]. In Subsection 2.2, we introduce and analyze a new randomized algorithm for k-OUT-TREE. We derandomize our algorithm in Subsection 2.3.

2.1. Algorithm of Alon, Yuster and Zwick

Let $c : V(D) \to \{1, \ldots, k\}$ be a vertex k-coloring of a digraph D and let T be a k-vertex out-tree contained in D (as a subgraph). Then $V(T)$ and T are *colorful* if no pair of vertices of T are of the same color.

Algorithm 1 of [1] verifies whether D contains a colorful out-tree H such that H is isomorphic to T, when a coloring $c : V(D) \to \{1, \ldots, k\}$ is given. Note that a k-vertex subgraph H will be colorful with a probability of at least $k!/k^k > e^{-k}$. Thus, we can find a copy of T in D in e^k expected iterations of Algorithm 1.

Theorem 2.1. *Let T be an out-tree on k vertices and let $D = (V, A)$ be a digraph. A subgraph of D isomorphic to T, if one exists, can be found in $O(k(4e)^k \cdot |A|)$ expected time by running the algorithm $\mathcal{L}(T, r)$ for a random coloring c iteratively.*

Proof. Let $c : V(D) \to \{1,\dots,k\}$ be a given coloring of D and suppose T' and T'' are the subtrees of T obtained in line 8. Let $|V(T')| = k'$ and $|V(T'')| = k''$, where $k' + k'' = k$. Then $|\mathcal{C}_{T'}(u)| = \binom{k-1}{k'-1}$ and $|\mathcal{C}_{T'}(u)| = \binom{k-1}{k''-1}$. Checking $C' \cap C'' = \emptyset$ takes $O(k)$ time. Hence, lines 11–19 require at most $\binom{k}{k/2}^2 \cdot k|A| \leqslant k2^{2k}|A|$ operations.

Let $T(k)$ be the number of operations for $\mathcal{L}(T,r)$. We have the following recursion

$$T(k) \leqslant T(k') + T(k'') + k2^{2k-2}|A|. \tag{1}$$

By induction, it is not difficult to check that $T(k) \leqslant k4^k|A|$. Since the expected number of iterations of the algorithm $\mathcal{L}(T,r)$ is at most e^k, we achieve the claimed running time. □

Let $\mathcal{C}$ be a family of vertex k-colorings of a digraph D. We call $\mathcal{C}$ an (n,k)-*family of perfect hashing functions* if for each $X \subseteq V(D)$, $|X| = k$, there is a coloring $c \in \mathcal{C}$ such that X is colorful with respect to c. One can derandomize Algorithm 1 of Alon et al. by using any (n,k)-family of perfect hashing functions in the obvious way. The time complexity of the derandomized algorithm depends of the size of the (n,k)-family of perfect hashing functions. Let $\tau(n,k)$ denote the minimum size of an (n,k)-family of perfect hashing functions. Nilli [12] proved that $\tau(n,k) \geqslant \Omega(e^k \log n/\sqrt{k})$. It is unclear whether there is an (n,k)-family of perfect hashing functions of size $O^*(e^k)$ [4], but even if it does exist, the running time of the derandomized algorithm would be $O^*((4e)^k)$.

2.2. *New algorithm for k-*OUT-TREE

Before we introduce our new randomized algorithm for k-OUT-TREE, we will give a brief account of the basic idea behind it. Let T be an out-tree on k vertices and let D be a digraph in which we want to find a copy of T. As in the randomized algorithm by Alon, Yuster and Zwick in [1], we break T into two subtrees T_w and T_b. However, unlike the former which deletes an arc of T, we break it by choosing a 'splitting vertex' denoted as v^* and furthermore the resulting two subtrees overlap exactly on this splitting vertex v^*. Next we randomly partition the digraph D into two vertex-disjoint parts D_w and D_b, and then find a copy of T_w in D_w and a copy of T_b in D_b, if one exists. If we try sufficiently many partitions of D, it is possible to find a copy of T whenever D contains one as a subgraph (with some good probability in a randomized version of the algorithm, which can be derandomized consequently).

The trouble is that the fact that D_w and D_b contain copies of T_w and T_b, respectively does not necessarily means that D contains a copy of T as a whole. We need to ensure that there exist copies of T_w and T_b that actually overlap (and overlap only) on a vertex of D corresponding to the splitting vertex v^*. To this end, we allow some vertices of D_w, say S, to be shared by D_b by considering $D_b + S$ instead of D_b. Here S is the set of vertices in D_w that could correspond to the splitting vertex v^* of T_w. When we search for a copy of T_b in $D_b + S$, only those trees isomorphic to T_b in $D_b + S$ are considered legitimate where the vertex corresponding to v^* lies in S. In other words, we convey the information S obtained in the phase for T_w-D_w to the next phase for T_b-D_b so that we do not only ensure the global connectivity of $T_w + T_b = T$ in D but also reduce the search space for finding a copy of T_b in D_b.

Moreover, by conveying the information for v^* we can save the extra effort for 'merging' the solutions (i.e. copies of T_w and T_b). Rather, once we obtain a copy of T_b in $D_b + S$, it follows immediately that we have a copy of T in D. Since the number of partitions of D we need to try is a function of k, the time complexity of finding a copy of T in D can be written as $T(k,n) = f(k)(T(k',n) + T(k-k',n) + p_1(n)) + p_2(n)$, $T(1,n) = p_3(n)$, where $p_i(n)$ is polynomial in n for $i = 1,2,3$. This is why the running time of our algorithm remains polynomial in n. Making this approach efficient depends crucially on two aspects:

1. to obtain k' in the above formula as close to half of k as possible; and
2. to replace $f(k)$ with as small growing function as possible.

For the latter, we use a simple *unbalanced-partition-strategy* which will be explained later. We achieve the former goal by choosing an appropriate splitting vertex v^* and then using it to obtain a T_w-T_b split. The splitting procedure is one of the key part of our algorithm and next we describe this procedure in details.

The following lemma is well known and will be used as a basic scheme of choosing v^*.

Lemma 2.2. *(See [5].) Let T be an undirected tree and let $w : V \to \mathbb{R}^+ \cup \{0\}$ be a weight function on its vertices. There exists a vertex $v \in V(T)$ such that the weight of every subtree T' of $T - v$ is at most $w(T)/2$, where $w(T) = \sum_{v \in V(T)} w(v)$.*

Consider a partition $n = n_1 + \cdots + n_q$, where n and all n_i are nonnegative integers and a bipartition (A,B) of the set $\{1,\dots,q\}$. Let $d(A,B) := |\sum_{i\in A} n_i - \sum_{i\in B} n_i|$. Given a set $Q = \{1,\dots,q\}$ with a nonnegative integer weight n_i for each element $i \in Q$, we say that a bipartition (A,B) of Q is *greedily optimal* if $d(A,B)$ does not decrease by moving an element of one partite set into another. Algorithm 2 describes how to obtain a greedily optimal bipartition in time $O(q \log q)$. For simplicity we write $\sum_{i\in A} n_i$ as $n(A)$.

Algorithm 2 Bipartition$(Q, \{n_i\colon\ i \in Q\})$.

```
Require: A set $Q = \{1, \ldots, q\}$ with a nonnegative integer weight $n_i$, $\forall i \in Q$.
Ensure: A greedily optimal bipartition $(A, B)$ of $Q$.

1: Let $A := \emptyset$, $B := Q$.
2: while $n(A) < n(B)$ and there is an element $i \in B$ with $0 < n_i < d(A, B)$ do
3:     Choose such an element $i \in B$ with a largest $n_i$.
4:     $A := A \cup \{i\}$ and $B := B - \{i\}$.
5: end while
6: Return $(A, B)$.
```

Lemma 2.3. *Let Q be a set of size q with a nonnegative integer weight n_i for each $i \in Q$. The algorithm* **Bipartition**$(Q, \{n_i\colon\ i \in Q\})$ *finds a greedily optimal bipartition $A \cup B = Q$ in time $O(q \log q)$.*

Proof. First we want to show that the values n_i chosen in line 3 of the algorithm do not increase during the performance of the algorithm. The values of n_i do not increase because the values of the difference $d(A, B)$ do not increase during the performance of the algorithm. In fact, $d(A, B)$ strictly decreases. To see this, suppose that the element i is selected in the present step. If $n(A \cup \{i\}) < n(B - \{i\})$, then obviously the difference $d(A, B)$ strictly decreases. Else if $n(A \cup \{i\}) > n(B - \{i\})$, we have $d(A \cup \{i\}, B - \{i\}) < n_i < d(A, B)$.

To see that the algorithm returns a greedily optimal bipartition (A, B), it is enough to observe that for the final bipartition (A, B), moving any element of A or B does not decrease $d(A, B)$. Suppose that the last movement of the element i_0 makes $n(A) > n(B)$. Then a simple computation implies that $d(A, B) < n_{i_0}$. Since the values of n_i in line 3 of the algorithm do not increase during the performance of the algorithm, $n_j \geqslant n_{i_0} > d(A, B)$ for every $j \in A$, the movement of any element in A would not decrease $d(A, B)$. On the other hand suppose that $n(A) < n(B)$. By the definition of the algorithm, for every $j \in B$ with a positive weight we have $n_j \geqslant d(A, B)$ and thus the movement of any element in B would not decrease $d(A, B)$. Hence the current bipartition (A, B) is greedily optimal.

Now let us consider the running time of the algorithm. Sorting the elements in nondecreasing order of their weights will take $O(q \log q)$ time. Moreover, once an element is moved from one partite set to another, it will not be moved again and we move at most q elements without duplication during the algorithm. This gives us the running time of $O(q \log q)$. □

Now we describe a new randomized algorithm for k-Out-Tree. Let D be a digraph and let T be an out-tree on k vertices. Let us specify a vertex $t \in V(T)$ and a vertex $w \in V(D)$. We call a copy of T in D a *T-isomorphic* tree. We say that a T-isomorphic tree T_D in D is a (t, w)-tree if $w \in V(T_D)$ plays the role of t.

We first give an intuitive explanation of our algorithm before giving a formal description. To find the desired tree in the given input digraph, we first split the tree in two parts with one common vertex such that the both parts are 'almost balanced' Then we randomly partition the vertices of the D in two parts with probability of a vertex lying in one part or the other depends on the sizes of the trees we obtained in the first step by splitting it on a vertex. This allows us to more or less independently look for the different parts of the tree in different parts of the partition. We finally merge them cleverly to obtain our solution.

In Algorithm 3 **find-tree**, we have several arguments other than the natural arguments T and D. Our next argument is a vertex t of T. The argument t indicates that we want to return, at the end of the current procedure, the set of vertices X_t such that there is a (t, w)-tree for every $w \in X_t$. The fact that $X_t \neq \emptyset$ means two points: we have a T-isomorphic tree in D, and the information X_t we have can be used to construct a larger tree which uses the current T-isomorphic tree as a building block. Here, X_t is a kind of 'joint'.

The basic strategy is as follows. We choose a pair T_A and T_B of subtrees of T such that $V(T_A) \cup V(T_B) = V(T)$ and T_A and T_B share only one vertex, namely v^*, the splitting vertex. We call recursively two '**find-tree**' procedures on subsets of $V(D)$ to ensure that the subtrees playing the role of T_A and T_B do not overlap. The first call (line 15) tries to find X_{v^*} and the second one (line 18), using the information X_{v^*} delivered by the first call, tries to find X_t.

We also need another argument to our algorithm **find-tree** which is useful while merging and that is:

- a pair consisting of $L \subseteq V(T)$ and $\{X_u\colon\ u \in L\}$, where $X_u \subset V(D)$ and X_u's are pairwise disjoint.

The arguments $L \subseteq V(T)$ and $\{X_u\colon\ u \in L\}$ form a set of information needed to argue the correctness of the algorithm. Essentially L is a set of vertices of the tree T which has been used as a splitting vertex at some point during the execution of our recursive procedure. Let T_D be a T-isomorphic tree; if for every $u \in L$, T_D is a (u, w)-tree for some $w \in X_u$ and $V(T_D) \cap X_u = \{w\}$, we say that T_D *meets the restrictions on L*. The algorithm **find-tree** intends to find the set X_t of vertices such that for every $w \in X_t$, there is a (t, w)-tree which meets the restrictions on L.

Deleting a splitting vertex v^* may produce several subtrees, and there might be many ways to divide them into two groups, namely (T_A, T_B). To make the algorithm more efficient, we try to obtain as 'balanced' a partition (T_A, T_B) as possible. The algorithm **tree-Bipartition** is used to produce a pretty 'balanced' bipartition of the subtrees. Moreover we introduce another argument to have a better complexity behavior. The argument v is a vertex which indicates whether there is a

Algorithm 3 find-tree$(T, D, v, t, L, \{X_u\colon u \in L\})$.

Require: An out-tree T on k vertices, a digraph D, $v \in \{\emptyset\} \cup V(T)$, a specified vertex $t \in V(T)$, a subset of vertices $L \subseteq V(T)$, a family of pairwise disjoint subsets $X_u \subseteq V(D)$ for each $u \in L$.
Ensure: A set of vertices $X_t \subseteq V(D)$ such that there is a (t, w)-tree for every $w \in X_t$.

```
1: if $|V(T) \setminus L| \geqslant 2$ then
2:     for all $u \in V(T)$: Set $w(u) := 0$ if $u \in L$, $w(u) := 1$ otherwise.
3:     if $v = \emptyset$ then Find $v^* \in V(T)$ such that the weight of every subtree $T'$ of $T - v^*$ is at most $w(T)/2$ (see Lemma 2.2) else $v^* := v$
4:     $(WH, BL)$:=tree-Bipartition$(T, t, v^*, L)$.
5:     $U_w := \bigcup_{i \in WH} V(T_i) \cup \{v^*\}$, $U_b := \bigcup_{i \in BL} V(T_i)$.
6:     for all $u \in L \cap U_w$: color all vertices of $X_u$ in white.
7:     for all $u \in L \cap (U_b \setminus \{v^*\})$: color all vertices of $X_u$ in black.
8:     $\alpha := \min\{w(U_w)/w(T), w(U_b)/w(T)\}$.
9:     if $\alpha^2 - 3\alpha + 1 \leqslant 0$ (i.e., $\alpha \geqslant (3 - \sqrt{5})/2$, see (2) and the definition of $\alpha^*$ afterwards) then $v_w := v_b := \emptyset$
10:    else if $w(U_w) < w(U_b)$ then $v_w := \emptyset$, $v_b := v^*$ else $v_w := v^*$, $v_b := \emptyset$.
11:    $X_t := \emptyset$.
12:    for $i = 1$ to $\lceil \frac{2.51}{\alpha^{\alpha k}(1-\alpha)^{(1-\alpha)k}} \rceil$ do
13:        Color the vertices of $V(D) - \bigcup_{u \in L} X_u$ in white or black such that for each vertex the probability to be colored in white is $\alpha$ if $w(U_w) \leqslant w(U_b)$, and $1 - \alpha$ otherwise.
14:        Let $V_w$ ($V_b$) be the set of vertices of $D$ colored in white (black).
15:        $S$ :=find-tree$(T[U_w], D[V_w], v_w, v^*, L \cap U_w, \{X_u\colon u \in L \cap U_w\})$
16:        if $S \neq \emptyset$ then
17:            $X_{v^*} := S$, $L := L \cup \{v^*\}$.
18:            $S'$ :=find-tree$(T[U_b \cup \{v^*\}], D[V_b \cup S], v_b, t, (L \cap U_b), \{X_u\colon u \in (L \cap U_b)\})$.
19:            $X_t := X_t \cup S'$.
20:        end if
21:    end for
22:    Return $X_t$.
23: else $\{|V(T) \setminus L| \leqslant 1\}$
24:    if $\{z\} = V(T) \setminus L$ then $X_z := V(D) - \bigcup_{u \in L} X_u$, $L := L \cup \{z\}$.
25:    $L^o := \{$all leaf vertices of $T\}$.
26:    while $L^o \neq L$ do
27:        Choose a vertex $z \in L \setminus L^o$ s.t. $N_T^+(z) \subseteq L^o$.
28:        $X_z := X_z \cap \bigcap_{u \in N_T^+(z)} N^-(X_u)$; $L^o := L^o \cup \{z\}$.
29:    end while
30:    Return $X_t$.
31: end if
```

Algorithm 4 tree-Bipartition(T, t, v^*, L).

```
1: $T_1, \ldots, T_q$ are the subtrees of $T - v^*$. $Q := \{1, \ldots, q\}$. $w(T_i) := |V(T_i) \setminus L|$, $\forall i \in Q$.
2: if $v^* = t$ then
3:     $(A, B)$:=Bipartition$(Q, \{n_i := w(T_i)\colon i \in Q\})$
4:     if $w(A) \leqslant w(B)$ then $WH := A$, $BL := B$. else $WH := B$, $BL := A$.
5: else
6:     Let $l$ be such that $t \in V(T_l)$
7:     if $w(T_l) - w(v^*) \geqslant 0$ then
8:         $(A, B)$:=Bipartition$(Q, \{n_i := w(T_i)\colon i \in Q \setminus \{l\}\} \cup \{n_l := w(T_l) - w(v^*)\})$
9:         if $l \in B$ then $WH := A$, $BL := B$. else $WH := B$, $BL := A$
10:    else $\{w(T_l) - w(v^*) < 0\}$
11:        $(A, B)$:=Bipartition$((Q \setminus \{l\}) \cup \{v^*\}, \{n_i := w(T_i)\colon i \in Q \setminus \{l\}\} \cup \{n_{v^*} := w(v^*)\})$
12:        if $v^* \in A$ then $WH := A - \{v^*\}$, $BL := B \cup \{l\}$. else $WH := B - \{v^*\}$, $BL := A \cup \{l\}$
13:    end if
14: end if
15: Return $(WH, BL)$.
```

predetermined splitting vertex. If $v = \emptyset$, we do not have a predetermined splitting vertex so we find one in the current procedure. Otherwise, we use the vertex v as a splitting vertex.

Let r be the root of T. To decide whether D contains a copy of T, it suffices to run **find-tree**$(T, D, \emptyset, r, \emptyset, \emptyset)$.

Lemma 2.4. *During the performance of find-tree$(T, D, \emptyset, r, \emptyset, \emptyset)$, the sets X_u, $u \in L$ are pairwise disjoint.*

Proof. We prove the claim inductively. For the initial call, trivially the sets X_u, $u \in L$ are pairwise disjoint since $L = \emptyset$. Suppose that for a call find-tree$(T, D, v, t, L, \{X_u\colon u \in L\})$ the sets X_v, $v \in L$ are pairwise disjoint. For the first subsequent call in line 15, the sets are obviously pairwise disjoint. Consider the second subsequent call in line 18. If $v^* \in L$ before line 17, the claim is true since we convey the argument $t := v^*$ to the first subsequent call in line 15 and thus S is contained in X_{v^*}. Otherwise, observe that $X_u \subseteq V_b$ for all $u \in L \cap U_b$ and they are pairwise disjoint. Since $X_{v^*} \cap V_b = \emptyset$, the sets X_u for all $u \in L \cap U_b$ together with X_{v^*} are pairwise disjoint. □

The algorithm **tree-Bipartition** is a subroutine used during the execution of **find-tree**. Let $T_1, \dots, T_q$ be the subtrees of $T - v^*$, where v^* is a splitting vertex of the current call to **find-tree**. At the end of **tree-Bipartition**, we obtain a partition of the subtrees, or more precisely, a partition (WH, BL) of the indices $\{1, \dots, q\}$ of the subtrees. The attained partition (WH, BL) is 'a greedily optimal bipartition' in certain sense while a nonnegative integer weight on an element of $\{1, \dots, q\}$ is set to be $w(T_i)$ with some fine-tuning.

Lemma 2.5. *Consider the algorithm* **tree-Bipartition** *and let (WH, BL) be a bipartition of $\{1, \dots, q\}$ obtained at the end of the algorithm. Then the partition $U_w := \bigcup_{i \in WH} V(T_i) \cup \{v^*\}$ and $U_b := \bigcup_{i \in BL} V(T_i)$ of $V(T)$ has the following property.*

(1) *If $v^* = t$, moving a component T_i from one partite set to the other does not decrease the difference $d(w(U_w), w(U_b))$.*
(2) *If $v^* \neq t$, either exchanging v^* and the component T_l or moving a component T_i, $i \neq v^*, l$ from one partite set to the other does not decrease the difference $d(w(U_w), w(U_b))$.*

Proof. Let us consider the property (1). The bipartition (WH, BL) is determined in the first 'if' statement in line 2 of **tree-Bipartition**. Then by Lemma 2.3 the bipartition (WH, BL) is greedily optimal, which is equivalent to the statement of (1).

Let us consider the property (2). First suppose that the bipartition (WH, BL) is determined in the 'if' statement in line 7 of **tree-Bipartition**. The exchange of v^* and the component T_l amounts to moving the element l in the algorithm **Bipartition**. Since (WH, BL) is returned by **Bipartition** and thus is a greedily optimal bipartition of Q, any move of an element in one partite set would not decrease the difference $d(WH, BL)$ and the statement of (2) holds in this case.

Secondly suppose that the bipartition (WH, BL) is determined in the 'if' statement in line 10 of **tree-Bipartition**. In this case we have $w(T_l) = 0$ and thus exchanging T_l and v^* and amounts to moving the element v^* in the algorithm **Bipartition**. By the same argument as above, any move of an element in one partite set would not decrease the difference $d(WH, BL)$ and again the statement of (2) holds. □

Consider the following equation:

$$\alpha^2 - 3\alpha + 1 = 0. \tag{2}$$

Let $\alpha^* := (3 - \sqrt{5})/2$ be one of its roots. In line 10 of the algorithm **find-tree**, if $\alpha < \alpha^*$ we decide to pass the present splitting vertex v^* as a splitting vertex to the next recursive call which gets, as an argument, a subtree with greater weight among the two subtrees $T[U_w]$ and $T[U_b \cup \{v^*\}]$. Lemma 2.6 justifies this execution. It claims that if $\alpha < \alpha^*$, then in the next recursive call with a subtree of weight $(1 - \alpha)w(T)$, we have a more balanced bipartition with v^* as a splitting vertex. Actually, the bipartition in the next step is good enough so as to compensate for the increase in the running time incurred by the biased ('$\alpha < \alpha^*$') bipartition in the present step. We will show this later.

Lemma 2.6. *Suppose that v^* has been chosen to split T for the present call to* **find-tree** *such that the weight of every subtree of $T - v^*$ is at most $w(T)/2$ and that $w(T) \geqslant 5$. Let α be defined as in line 8 and assume that $\alpha < \alpha^*$. Let $\{U_1, U_2\} = \{U_w, U_b\}$ such that $w(U_2) \geqslant w(U_1)$ and let $\{T_1, T_2\} = \{T[U_w], T[U_b \cup \{v^*\}]\}$ such that $U_1 \subseteq V(T_1)$ and $U_2 \subseteq V(T_2)$. Let α' play the role of α in the recursive call using the tree T_2. In this case the following holds: $\alpha' \geqslant (1 - 2\alpha)/(1 - \alpha) > \alpha^*$.*

Proof. Let $T_1, T_2, U_1, U_2, \alpha, \alpha'$ be defined as in the statement. Note that $\alpha = w(U_1)/w(T)$. Let $d = w(U_2) - w(U_1)$ and note that $w(U_1) = (w(T) - d)/2$ and that the following holds

$$\frac{1 - 2\alpha}{1 - \alpha} = \frac{w(T) - 2w(U_1)}{w(T) - w(U_1)} = \frac{2d}{w(T) + d}.$$

We now consider the following cases.

Case 1. $d = 0$: In this case $\alpha = 1/2 > \alpha^*$, a contradiction.

Case 2. $d = 1$: In this case $\alpha^* > \alpha = w(U_1)/(2w(U_1) + 1)$, which implies that $w(U_1) \leqslant 1$. Therefore $w(U_2) \leqslant 2$ and $w(T) \leqslant 3$, a contradiction.

Case 3. $d \geqslant 2$: Let $C_1, C_2, \dots, C_q$ denote the components in $T - v^*$ and without loss of generality assume that $V(C_1) \cup V(C_2) \cup \dots \cup V(C_a) = U_2$ and $V(C_{a+1}) \cup V(C_{a+2}) \cup \dots \cup V(C_q) = U_1$. Note that by Lemma 2.5 we must have $w(C_i) \geqslant d$ or $w(C_i) = 0$ for all $i = 1, 2, \dots, q$ except possibly for one set C_l (containing t), which may have $w(C_l) = 1$ (if $w(v^*) = 1$).

Let C_r be chosen such that $w(C_r) \geqslant d$, $1 \leqslant r \leqslant a$ and $w(C_r)$ is minimum possible with these constraints. We first consider the case when $w(C_r) > w(U_2) - w(C_r)$. By the above (and the minimality of $V(C_r)$) we note that $w(U_2) \leqslant w(C_r) + 1$ (as either C_j, which is defined above, or v^* may belong to $V(T_2)$, but not both). As $w(U_2) = (w(T) + d)/2 \geqslant w(T)/2 + 1$ we note that $w(C_r) \geqslant w(T)/2 + d/2 - 1$. As $w(C_r) \leqslant w(T)/2$ (By the statement in our theorem) this implies that $d = 2$ and $w(C_r) = w(T)/2$ and $w(U_2) = w(C_r) + 1$. If U_1 contains at least two distinct components with weight at least d then $w(U_1) > w(U_2)$, a contradiction. If U_1 contains no component of weight at least d then $w(U_1) \leqslant 1$ and $w(T) \leqslant 4$, a contradiction. So U_1 contains exactly one component of weight at least d. By the minimality of $w(C_r)$ we note that $w(U_1) \geqslant w(C_r) = w(U_2) - 1$, a contradiction to $d \geqslant 2$.

Therefore we can assume that $w(C_r) \leqslant w(U_2) - w(C_r)$, which implies the following (the last equality is proved above)

$$\alpha' \geqslant \frac{w(C_r)}{w(U_2)} \geqslant \frac{d}{(w(T)+d)/2} = \frac{1-2\alpha}{1-\alpha}.$$

As $\alpha < \alpha^*$, we note that $\alpha' \geqslant (1-2\alpha)/(1-\alpha) > (1-2\alpha^*)/(1-\alpha^*) = \alpha^*$. $\square$

For the selection of the splitting vertex v^* we have two criteria in the algorithm **find-tree**: (i) *'found'* criterion: the vertex is found so that the weight of every subtree T' of $T - v^*$ is at most $w(T)/2$. (ii) *'taken-over'* criterion: the vertex is passed on to the present step as the argument v by the previous step of the algorithm. The following statement is an easy consequence of Lemma 2.6.

Corollary 2.7. *Suppose that $w(T) \geqslant 5$. If v^* is selected with 'taken-over' criterion, then $\alpha > \alpha^*$.*

Proof. For the initial call find-tree$(T, D, \emptyset, r, \emptyset, \emptyset)$ we have $v = \emptyset$ and thus, the splitting vertex v^* is selected with the 'found' criterion. We will prove the claim by induction. Consider the first vertex v^* selected with then 'taken-over' criterion during the performance of the algorithm. Then in the previous step, the splitting vertex was selected with 'found' criterion and thus in the present step we have $\alpha > \alpha^*$ by Lemma 2.6.

Now consider a vertex v^* selected with the 'taken-over' criterion. Then in the previous step, the splitting vertex was selected with the 'found' criterion since otherwise, by the induction hypothesis we have $\alpha > \alpha^*$ in the previous step, and $\emptyset$ has been passed on as the argument v for the present step. This is a contradiction. $\square$

Due to Corollary 2.7 the vertex v^* selected in line 3 of the algorithm **find-tree** functions properly as a splitting vertex. In other words, we have more than one subtree of $T - v^*$ in line 4 with positive weights.

Lemma 2.8. *If $w(T) \geqslant 2$, then for each of U_w and U_b found in line 5 of by* **find-tree** *we have $w(U_w) > 0$ and $w(U_b) > 0$.*

Proof. For the sake of contradiction suppose that one of $w(U_w)$ and $w(U_b)$ is zero. Let us assume $w(U_w) = 0$ and $w(U_b) = w(T)$. If v^* is selected with 'found' criteria, each component in $T[U_b]$ has a weight at most $w(T)/2$ and $T[U_b]$ contains at least two components of positive weights. Then we can move one component with a positive weight from U_b to U_w which will reduce the difference $d(U_w, U_b)$, a contradiction. The same argument applies when $w(U_w) = w(T)$ and $w(U_b) = 0$.

Consider the case when v^* is selected with 'taken-over' criteria. There are three possibilities.

Case 1. $w(T) \geqslant 5$: In this case we obtain a contradiction with Corollary 2.7.

Case 2. $w(T) = 4$: In the previous step using T', where $T \subseteq T'$, the splitting vertex v^* was selected with 'found' criteria. Then by the argument in the first paragraph, we have $w(T') \geqslant 5$. A contradiction follows from Lemma 2.6.

Case 3. $2 \leqslant w(T) \leqslant 3$: First suppose that $w(v^*) = 0$. Note that $T[U_w] - v^*$ or $T[U_b]$ contains a component of weight $w(T)$ since otherwise we can move a component with a positive weight from one partite set to the other and reduce $d(U_w, U_b)$. Considering the previous step using T', where $T \subseteq T'$, the out-tree T is the larger of T'_w and T'_b. We pass the splitting vertex v^* to the larger of the two only when $\alpha > \alpha^*$. So when $w(T) = 3$, we have $3 > (1-\alpha^*)w(T')$ and thus $w(T') \leqslant 4$, and when $w(T) = 2$ we have $2 > (1-\alpha^*)w(T')$ and thus $w(T') \leqslant 3$. In either case, however, $T' - v^*$ contains a component with a weight greater than $w(T')/2$, contradicting to the choice of v^* in the previous step (recall that v^* is selected with 'found' criteria in the previous step using T').

Secondly suppose that $w(v^*) = 1$. Then $w(U_w) = w(T)$ and $w(U_b) = 0$. We can reduce the difference $d(U_w, U_b)$ by moving the component with a positive weight from U_w to U_b, a contradiction.

Therefore for each of U_w and U_b found in line 5 of by **find-tree** we have $w(U_w) > 0$ and $w(U_b) > 0$. $\square$

Lemma 2.9. *Given a digraph D, an out-tree T and a specified vertex $t \in V(T)$, consider the set X_t (in line 22) returned by the algorithm* **find-tree**$(T, D, v, t, L, \{X_u\colon u \in L\})$. *We assume that the sets X_u, $u \in L$ are pairwise disjoint. If $w \in X_t$ then D contains a (t, w)-tree that meets the restrictions on L. Conversely, if D contains a (t, w)-tree for a vertex $w \in V(D)$ that meets the restrictions on L, then X_t contains w with probability larger than $1 - 1/e > 0.6321$.*

Proof. Lemma 2.8 guarantees that the splitting vertex v^* selected at any recursive call of **find-tree** really 'splits' the input out-tree T into two nontrivial parts, unless $w(T) \leqslant 1$.

First we show that if $w \in X_t$ then D contains a (t, w)-tree for a vertex $w \in V(D)$ that meets the restrictions on L. When $|V(T) \setminus L| \leqslant 1$, using Lemma 2.4 it is straightforward to check from the algorithm that the claim holds. Assume that the claim is true for all subsequent calls to **find-tree**. Since $w \in S'$ for some S' returned by a call in line 18, the subgraph $D[V_b \cup X_{v^*}]$ contains a $T[U_b \cup \{v^*\}]$-isomorphic (t, w)-tree T_D^b meeting the restrictions on $(L \cap U_b) \cup \{v^*\}$ by induction hypothesis. Moreover, $X_{v^*} \neq \emptyset$ when $S' \ni w$ is returned and this implies that there is a vertex $u \in X_{v^*}$ such that T_D^b is a (v^*, u)-tree. Since $u \in X_{v^*}$, induction hypothesis implies that the subgraph $D[V_w]$ contains a $T[U_w]$-isomorphic (v^*, u)-tree, say T_D^w.

Consider the subgraph $T_D := T_D^w \cup T_D^b$. To show that T_D is a T-isomorphic (t,w)-tree in D, it suffices to show that $V(T_D^w) \cap V(T_D^b) = \{u\}$. Indeed, $V(T_D^w) \subseteq V_w$, $V(T_D^b) \subseteq V_b \cup X_{v^*}$ and $V_w \cap V_b = \emptyset$. Thus if two trees T_D^w and T_D^b share vertices other than u, these common vertices should belong to X_{v^*}. Since T_D^b meets the restrictions on $(L \cap U_b) \cup \{v^*\}$, we have $X_{v^*} \cap V(T_D^b) = \{u\}$. Hence u is the only vertex that two trees T_D^w and T_D^b have in common. We know that u plays the role of v^* in both trees. Therefore we conclude that T_D is T-isomorphic, and since w plays the role of t, it is a (t,w)-tree. Obviously T_D meets the restrictions on L.

Secondly, we shall show that if D contains a (t,w)-tree for a vertex $w \in V(D)$ that meets the restrictions on L, then X_t contains w with probability larger than $1 - 1/e > 0.6321$. When $|V(T) \setminus L| \leqslant 1$, the algorithm **find-tree** is deterministic and returns X_t which is exactly the set of all vertices w for which there exists a (t,w)-tree meeting the restrictions on L. Hence the claim holds for the base case, and we may assume that the claim is true for all subsequent calls to **find-tree**.

Suppose that there is a (t,w)-tree T_D meeting the restrictions on L and that this is a (v^*, w')-tree, that is, the vertex w' plays the role of v^*. Then the vertices of T_D corresponding to U_w, say T_D^w, are colored white and those of T_D corresponding to U_b, say T_D^b, are colored black as intended with probability $\geqslant (\alpha^\alpha(1-\alpha)^{1-\alpha})^k$. When we hit the right coloring for T, the digraph $D[V_w]$ contains the subtree T_D^w of T_D which is $T[U_w]$-isomorphic and which is a (v^*, w')-tree. By induction hypothesis, the set S obtained in line 15 contains w' with probability larger than $1 - 1/e$. Note that T_D^w meets the restrictions on $L \cap U_w$.

If $w' \in S$, the restrictions delivered onto the subsequent call for **find-tree** in line 17 contains w'. Since T_D meets the restrictions on L confined to $U_b - v^*$ and it is a (v^*, w')-tree with $w' \in S = X_{v^*}$, the subtree T_D^b of T_D which is $T[U_b \cup \{v^*\}]$-isomorphic meets all the restrictions on L. Hence by induction hypothesis, the set S' returned in line 18 contains w with probability larger than $1 - 1/e$.

The probability ρ that S', returned by **find-tree** in line 18 at an iteration of the loop, contains w is, thus,

$$\rho > \big(\alpha^\alpha(1-\alpha)^{1-\alpha}\big)^k \times (1-1/e)^2 > 0.3995\big(\alpha^\alpha(1-\alpha)^{1-\alpha}\big)^k.$$

After looping $\lceil (0.3995(\alpha^\alpha(1-\alpha)^{1-\alpha})^k)^{-1} \rceil$ times in line 12, the probability that X_t contains w is at least

$$1 - (1-\rho)^{1/(0.3995(\alpha^\alpha(1-\alpha)^{1-\alpha})^k)} > 1 - \big(1 - 0.3995\big(\alpha^\alpha(1-\alpha)^{1-\alpha}\big)^k\big)^{1/(0.3995(\alpha^\alpha(1-\alpha)^{1-\alpha})^k)} > 1 - \frac{1}{e}.$$

Observe that the probability ρ does not depend on α and the probability of coloring a vertex white/black. □

The complexity of algorithm **find-tree** is analyzed in the following theorem.

Theorem 2.10. *Algorithm* **find-tree** *has running time $O(n^2 k^\rho C^k)$, where $w(T) = k$ and $|V(D)| = n$, and C and ρ are defined and bounded as follows*:

$$C = \left(\frac{1}{\alpha^{*\alpha^*}(1-\alpha^*)^{1-\alpha^*}}\right)^{1/\alpha^*}, \qquad \rho = \frac{\ln(1/6)}{\ln(1-\alpha^*)}, \quad \rho \leqslant 3.724, \ \textit{and } C \leqslant 5.7039.$$

Proof. Let $L(T,D)$ denote the number of times the 'if'-statement in line 1 of algorithm **find-tree** is false (in all recursive calls to **find-tree**). We will prove that $L(T,D) \leqslant R(k) = Bk^\rho C^k + 1$, $B \geqslant 1$ is a constant whose value will determined later in the proof. This would imply that the number of calls to **find-tree** where the 'if'-statement in line 1 is true is also bounded by $R(k)$ as if line 1 is true then we will have at least two calls to **find-tree** (in fact it will have at least three as $\lceil \frac{2.51}{\alpha^{\alpha k}(1-\alpha)^{(1-\alpha)k}} \rceil \geqslant 3$ and we always have a call in line 15). We can therefore think of the search tree of Algorithm 3 as an out-tree where all internal nodes have out-degree at least two and therefore the number of leaves is greater than the number of internal nodes.

Observe that each iteration of the for-loop in line 12 of algorithm **find-tree** makes at most two recursive calls to **find-tree** and the time spent in each iteration of the for-loop is at most $O(n^2)$. As the time spent in each call of **find-tree** outside the for-loop is also bounded by $O(n^2)$ we obtain the desired complexity bound $O(n^2 k^\rho C^k)$.

Thus, it remains to show that $L(T,D) \leqslant R(k) = Bk^\rho C^k + 1$. First note that if $k = 0$ or $k = 1$ then line 1 is false exactly once (as there are no recursive calls) and $\min\{R(1), R(0)\} \geqslant 1 = L(T,D)$. If $k \in \{3,4\}$, then line 1 is false a constant number of times by Lemma 2.8 and let B be the minimal integer such that $L(T,D) \leqslant R(k) = Bk^\rho C^k + 1$ for both $k = 3$ and 4. Thus, we may now assume that $k \geqslant 5$ and proceed by induction on k.

Let $R'(\alpha,k) = (6((1-\alpha)k)^\rho C^{(1-\alpha)k})/(\alpha^{\alpha k}(1-\alpha)^{(1-\alpha)k})$. Let α be defined as in line 8 of algorithm **find-tree**. We will consider the following two cases separately.

Case 1. $\alpha \geqslant \alpha^*$: In this case we note that the following holds as $k \geqslant 2$ and $(1-\alpha) \geqslant \alpha$,

$$\begin{aligned}
L(T,D) &\leqslant \big\lceil 2.51/\big(\alpha^{\alpha k}(1-\alpha)^{(1-\alpha)k}\big)\big\rceil \times \big(R(\alpha k) + R\big((1-\alpha)k\big)\big) \\
&\leqslant 3/\big(\alpha^{\alpha k}(1-\alpha)^{(1-\alpha)k}\big) \times \big(2 \cdot R\big((1-\alpha)k\big)\big) \\
&= R'(\alpha,k).
\end{aligned}$$

By the definition of ρ we observe that $(1-\alpha^*)^\rho = 1/6$, which implies that the following holds by the definition of C:

$$R'(\alpha^*, k) = 6\big(\big(1-\alpha^*\big)k\big)^\rho C^{(1-\alpha^*)k} \times C^{\alpha^* k} = k^\rho C^k = R(k).$$

Observe that

$$\ln\big(R'(\alpha,k)\big) = \ln(6) + \rho\big[\ln(k) + \ln(1-\alpha)\big] + k\big[(1-\alpha)\ln(C) - \alpha\ln(\alpha) - (1-\alpha)\ln(1-\alpha)\big].$$

We now differentiate $\ln(R'(\alpha,k))$ which gives us the following:

$$\begin{aligned}\frac{\partial(\ln(R'(\alpha,k)))}{\partial(\alpha)} &= \rho\frac{-1}{1-\alpha} + k\big(-\ln(C) - \big(1+\ln(\alpha)\big) + \big(1+\ln(1-\alpha)\big)\big) \\ &= \frac{-\rho}{1-\alpha} + k\left(\ln\left(\frac{1-\alpha}{\alpha C}\right)\right).\end{aligned}$$

Since $k \geqslant 0$ we note that the above equality implies that $R'(\alpha,k)$ is a decreasing function in α in the interval $\alpha^* \leqslant \alpha \leqslant 1/2$. Therefore $L(T,D) \leqslant R'(\alpha,k) \leqslant R'(\alpha^*,k) = R(k)$, which proves Case 1.

Case 2. $\alpha < \alpha^*$: In this case we will specify the splitting vertex when we make recursive calls using the larger of U_w and U_b (defined in line 5 of algorithm **find-tree**). Let α' denote the α-value in such a recursive call. By Lemma 2.6 we note that the following holds:

$$\frac{1}{2} \geqslant \alpha' \geqslant \frac{1-2\alpha}{1-\alpha} > \alpha^*.$$

Analogously to Case 1 (as $R'(\alpha', (1-\alpha)k)$ is a decreasing function in α' when $1/2 \geqslant \alpha' \geqslant \alpha^*$) we note that the L-values for these recursive calls are bounded by the following, where $\beta = \frac{1-2\alpha}{1-\alpha}$ (which implies that $(1-\alpha)(1-\beta) = \alpha$):

$$\begin{aligned}R'\big(\alpha', (1-\alpha)k\big) &\leqslant R'\big(\beta, (1-\alpha)k\big) \\ &= 3/\big(\big(\beta^\beta(1-\beta)^{(1-\beta)}\big)^{(1-\alpha)k}\big) \times 2 \times R\big((1-\beta)(1-\alpha)k\big) \\ &= 6R(\alpha k)/\big(\big(\beta^\beta(1-\beta)^{(1-\beta)}\big)^{(1-\alpha)k}\big).\end{aligned}$$

Thus, in the worst case we may assume that $\alpha' = \beta = (1-2\alpha)/(1-\alpha)$ in all the recursive calls using the larger of U_w and U_b. The following now holds (as $k \geqslant 2$)

$$\begin{aligned}L(T,D) &\leqslant \big\lceil 2.51/\big(\alpha^{\alpha k}(1-\alpha)^{(1-\alpha)k}\big)\big\rceil \times \big(R(\alpha k) + R'\big(\alpha', (1-\alpha)k\big)\big) \\ &\leqslant 3/\big(\alpha^{\alpha k}(1-\alpha)^{(1-\alpha)k}\big) \times R(\alpha k) \times \big(1 + 6/\big(\big(\beta^\beta(1-\beta)^{(1-\beta)}\big)^{(1-\alpha)k}\big)\big) \\ &\leqslant 3R(\alpha k)/\big(\alpha^{\alpha k}(1-\alpha)^{(1-\alpha)k}\big) \times 7/\big(\big(\beta^\beta(1-\beta)^{(1-\beta)}\big)^{(1-\alpha)k}\big).\end{aligned}$$

Let $R^*(\alpha,k)$ denote the bottom right-hand side of the above equality (for any value of α). By the definition of ρ we note that $\rho = \frac{2\ln(1/6)}{2\ln(1-\alpha^*)} = \frac{\ln(1/36)}{\ln(\alpha^*)}$, which implies that $(\alpha^*)^\rho = 1/36$. By the definition of C and the fact that if $\alpha = \alpha^*$ then $\beta = (1-2\alpha^*)/(1-\alpha^*) = \alpha^*$, we obtain the following:

$$\begin{aligned}R^*\big(\alpha^*,k\big) &= 3R\big(\alpha^* k\big)/\big(\alpha^{*\alpha^* k}\big(1-\alpha^*\big)^{(1-\alpha^*)k}\big) \times 7/\big(\big(\alpha^{*\alpha^*}\big(1-\alpha^*\big)^{(1-\alpha^*)}\big)^{(1-\alpha^*)k}\big) \\ &= 21 \cdot R\big(\alpha^* k\big) \cdot C^{\alpha^* k} \cdot C^{\alpha^*(1-\alpha^*)k} \\ &= 21\alpha^{*\rho} k^\rho C^{\alpha^* k} \times C^{(2\alpha^* - \alpha^{*2})k} \\ &= 21\alpha^{*\rho} R(k) \\ &< R(k).\end{aligned}$$

We will now simplify $R^*(\alpha,k)$ further, before we differentiate $\ln(R^*(\alpha,k))$. Note that $\beta = \frac{1-2\alpha}{1-\alpha}$ implies that $(1-\alpha)(1-\beta) = \alpha$ and $\beta(1-\alpha) = 1-2\alpha$,

$$\begin{aligned}R^*(\alpha,k) &= 21R(\alpha k)/\big(\alpha^{\alpha k}(1-\alpha)^{(1-\alpha)k}\big) \times 1/\big(\big(\beta^\beta(1-\beta)^{(1-\beta)}\big)^{(1-\alpha)k}\big) \\ &= 21(\alpha k)^\rho C^{\alpha k}/\big(\alpha^{\alpha k}(1-\alpha)^{(1-\alpha)k}\big) \times 1/\left(\left(\frac{1-2\alpha}{1-\alpha}\right)^{(1-2\alpha)k}\left(\frac{\alpha}{1-\alpha}\right)^{\alpha k}\right) \\ &= 21(\alpha k)^\rho\big(C^\alpha/\big(\alpha^{2\alpha}(1-2\alpha)^{(1-2\alpha)}\big)\big)^k.\end{aligned}$$

Thus, we have the following:

$$\ln\big(R^*(\alpha,k)\big) = \ln(21) + \rho\big(\ln(k) + \ln(\alpha)\big) + k\big(\alpha\ln(C) - 2\alpha\ln(\alpha) - (1-2\alpha)\ln(1-2\alpha)\big).$$

We now differentiate $\ln(R^*(\alpha,k))$ which gives us the following:

$$\begin{aligned}\frac{\partial(\ln(R^*(\alpha,k)))}{\partial(\alpha)} &= \frac{\rho}{\alpha} + k\big(\ln(C) - 2\big(1+\ln(\alpha)\big) + 2\big(1+\ln(1-2\alpha)\big)\big)\\ &= \frac{\rho}{\alpha} + k\bigg(\ln\bigg(\frac{C(1-2\alpha)^2}{\alpha^2}\bigg)\bigg).\end{aligned}$$

Since $k \geqslant 0$ we note that the above equality implies that $R^*(\alpha,k)$ is an increasing function in α in the interval $1/3 \leqslant \alpha \leqslant \alpha^*$. Therefore $L(T,D) \leqslant R^*(\alpha,k) \leqslant R^*(\alpha^*,k) < R(k)$, which proves Case 2. $\square$

Theorem 2.11. *There is an $O(n^2 5.704^k)$ time randomized algorithm that solves the k-*Out-Tree *problem.*

*2.3. Derandomization of our randomized algorithm for k-*Out-Tree

In this subsection we discuss the derandomization of the algorithm **find-tree** using the general method presented by Chen et al. [4] and based on the construction of (n,k)-universal sets studied in [11].

Definition 2.12. An (n,k)-universal set $\mathcal{F}$ is a set of functions from $[n]$ to $\{0,1\}$, such that for every subset $S \subseteq [n]$, $|S| = k$ the set $\mathcal{F}|_S = \{f|_S\colon\ f \in \mathcal{F}\}$ is equal to the set 2^S of all the functions from S to $\{0,1\}$.

Such an universal set can play in **find-tree** the role of the random colorings. In the same article [4], Chen et al. also give an algorithm to generate one:

Proposition 2.13. *(See [4].) There is an $O(n2^{k+12\log^2 k})$ time deterministic algorithm that constructs an (n,k)-universal set of size bounded by $n2^{k+12\log^2 k+2}$.*

Using this universal set alone, however, would not enable us to obtain a deterministic fixed-parameter algorithm for **find-tree**, as the size of the family (and, thus, the number of iterations in the main loop of the algorithm) would now also depend on n, besides k. Hence, Chen et al. make use (see [4]) of a family of pre-coloring functions $(g_{n,k,z})_{z\leqslant 2n}$ to obtain a fixed-parameter algorithm. To explain it, let us first give a result from Fredman et al. [8].

Proposition 2.14. *Let n and k be integers, $n \geqslant k$, and let q_0 be the smallest prime number such that $n \leqslant q_0 < 2n$. For any k-subset S in $Z_n = 0,\dots,n-1$, there is an integer z, $0 \leqslant z \leqslant q_0$, such that the function $g_{n,k,z}$ over Z_n, defined as $g_{n,k,z}(a) = (az \bmod q_0) \bmod k^2$, is injective from S.*

By the above proposition, computing a (k^2,k)-universal set $\mathcal{F}_{k^2,k}$ instead of an (n,k)-universal set is enough for our purposes. Indeed, if we are looking for a k-subgraph S in our graph, there exists $1 \leqslant z \leqslant 2n$ such that $g_{n,k,z}$ is injective on S, thus ensuring that the family $\mathcal{F}'_{k,n,z} = \mathcal{F}_{k^2,k} \circ g_{n,k,z} = \{f \circ g_{n,k,z},\ f \in \mathcal{F}_{k^2,k}\}$ is such that $\mathcal{F}'_{k,n,z}|_S$ is equal to the set 2^S.

This way, derandomizing **find-tree** amounts to running it at most $2n$ times (once for each possible value of z), each time using as a set of coloring functions the family $\mathcal{F}'_{k,n,z}$. Two lines of the algorithm will then need to be modified:

```
13 for each function $f \in F'_{k,n,z}$ do
14 $\forall i$ such that $x_i \in V(D) - \bigcup_{u\in L} X_u$, let $v_i$ be colored in white if $f(i) = 0$ and in black if $f(i) = 1$.
```

Besides, we also need to pre-compute a (k^2,k')-universal set for any $k' \leqslant k$, as this will be needed in the recursions steps of the algorithm. By Proposition 2.13, this can be done in time $O(k^3 2^{k+12\log^2 k})$. Note that these modifications make the algorithm **find-tree** deterministic.

Then, from Proposition 2.14 we deduce that if a digraph D contains an out-tree T meeting the requirements, then there exists a z such that $g_{n,k,z}$ is injective on $V(T)$. During the iteration of the algorithm corresponding to z there will be an $f \in \mathcal{F}'_{k,n,z}$ such that the vertices corresponding to U_w in D with be colored in white while the vertices corresponding to U_b will be colored in black. Using induction on k, we can prove that this deterministic algorithm correctly returns the required out-tree provided that such an out-tree exists in the digraph.

Let us briefly sketch how the running time is derived. We consider the following type of recurrence relations:

$$T(k,n) \leqslant X_0 2^k \times \big(T\big((1-\alpha)k,n\big) + T(\alpha k,n)\big).$$

Here X_0 is a constant determined by the size of the initial out-tree we are considering, and it adds to the exponent of $T(k,n)$ with $o(k)$ factor. On the other hand, the value of α asymptotically evolves around α^* as we see in the randomized version of algorithm. As a result, $T(k,n)$ is a function of the form $(2^{1/\alpha^*})^{k+o(k)}$. Overall the computation is similar to that described in the proof of Theorem 2.10. Thus, we obtain the following:

Theorem 2.15. *There is an $O(n^2 6.139^{k+o(k)}) = O(n^2 6.14^k)$ time deterministic algorithm that solves the k-*Out-Tree *problem.*

3. Algorithm for *k*-Int-Out-Branching

A *k-internal out-tree* is an out-tree with at least k internal vertices. We call a k-internal out-tree *minimal* if none of its proper subtrees is a k-internal out-tree, or *minimal k-tree* in short. The Rooted Minimal k-Tree problem is as follows: given a digraph D, a vertex u of D and a minimal k-tree T, where k is a parameter, decide whether D contains an out-tree rooted at u and isomorphic to T. Recall that k-Int-Out-Branching is the following problem: given a digraph D and a parameter k, decide whether D contains an out-branching with at least k internal vertices. Finally, the k-Int-Out-Tree problem is stated as follows: given a digraph D and a parameter k, decide whether D contains an out-tree with at least k internal vertices.

Lemma 3.1. *Let T be a k-internal out-tree. Then T is minimal if and only if $|\mathrm{Int}(T)| = k$ and every leaf $u \in \mathrm{Leaf}(T)$ is the only child of its parent $N^-(u)$.*

Proof. Assume that T is minimal. It cannot have more than k internal vertices, because otherwise by removing any of its leaves, we obtain a subtree of T with at least k internal vertices. Thus $|\mathrm{Int}(T)| = k$. If there are sibling leaves u and w, then removing one of them provides a subtree of T with $|\mathrm{Int}(T)|$ internal vertices.

Now, assume that $|\mathrm{Int}(T)| = k$ and every leaf $u \in \mathrm{Leaf}(T)$ is the only child of its parent $N^-(u)$. Observe that every subtree of T can be obtained from T by deleting a leaf of T, a leaf in the resulting out-tree, etc. However, removing any leaf v from T decreases the number of internal vertices, and thus creates subtrees with at most $k-1$ internal vertices. Thus, T is minimal. □

In fact, Lemma 3.1 can be used to generate all non-isomorphic minimal k-trees. First, build an (arbitrary) out-tree T^0 with k vertices. Then extend T^0 by adding a vertex x' for each leaf $x \in \mathrm{Leaf}(T^0)$ with an arc (x,x'). The resulting out-tree T' satisfies the properties of Lemma 3.1. Conversely, by Lemma 3.1, any minimal k-tree can be constructed in this way.

Generating Minimal *k*-Tree (GMT) Procedure

a. Generate a k-vertex out-tree T^0 and a set $T' := T^0$.
b. For each leaf $x \in \mathrm{Leaf}(T')$, add a new vertex x' and an arc (x,x') to T'.

Due to the following simple observation, to solve k-Int-Out-Tree for a digraph D it suffices to solve Rooted Minimal k-Tree for each vertex $u \in V(D)$ and each minimal k-tree T rooted at u.

Lemma 3.2. *Any k-internal out-tree rooted at r contains a minimal k-tree rooted at r as a subdigraph.*

Similarly, the next two lemmas show that to solve k-Out-Branching for a digraph D it suffices to solve Rooted Minimal k-Tree for each vertex $u \in S$ and each minimal k-tree T rooted at u, where S is the unique strong connectivity component of D without incoming arcs.

Lemma 3.3. *(See [2].) A digraph D has an out-branching rooted at vertex $r \in V(D)$ if and only if D has a unique strong connectivity component S of D without incoming arcs and $r \in S$. One can check whether D has a unique strong connectivity component and find one, if it exists, in time $O(m+n)$, where n and m are the number of vertices and arcs in D, respectively.*

The next lemma is a folklore.

Lemma 3.4. *Suppose a given digraph D with n vertices and m arcs has an out-branching rooted at vertex r. Then any minimal k-tree rooted at r can be extended to a k-internal out-branching rooted at r in time $O(m+n)$.*

Since k-Int-Out-Tree and k-Int-Out-Branching can be solved similarly, we will only deal with the k-Int-Out-Branching problem. We will assume that our input digraph contains a unique strong connectivity component S. Our algorithm called *IOBA* for solving k-Int-Out-Branching for a digraph D runs in two stages. In the first stage, we generate *all* minimal k-trees. We use the GMT procedure described above to achieve this. At the second stage, for each $u \in S$ and each minimal k-tree T, we check whether D contains an out-tree rooted at u and isomorphic to T using our algorithm from the previous section.

We return TRUE if and only if we succeed in finding an out-tree H of D rooted at $u \in S$ which is isomorphic to a minimal k-tree.

In the literature, mainly rooted (undirected) trees and not out-trees are studied. However, every rooted tree can be made an out-tree by orienting every edge away from the root and every out-tree can be made a rooted tree by disregarding all orientations. Thus, rooted trees and out-trees are equivalent and we can use results obtained for rooted trees for out-trees.

Otter [13] showed that the number of non-isomorphic out-trees on k vertices is $t_k = O^*(2.95^k)$. We can generate all non-isomorphic rooted trees on k vertices using the algorithm of Beyer and Hedetniemi [3] of runtime $O(t_k)$. Using the GMT procedure we generate all minimal k-trees. We see that the first stage of IOBA can be completed in time $O^*(2.95^k)$.

In the second stage of IOBA, we try to find a copy of a minimal k-tree T in D using our algorithm from the previous section. The running time of our algorithm is $O^*(6.14^k)$. Since the number of vertices of T is bounded from above by $2k-1$, the overall running time for the second stage of the algorithm is $O^*(2.95^k \cdot 6.14^{2k-1})$. Thus, the overall time complexity of the algorithm is $O^*(2.95^k \cdot 6.14^{2k-1}) = O^*(112^k)$.

We can reduce the complexity with a more refined analysis of the algorithm. The major contribution to the large constant 112 in the above simple analysis comes from the running time of our algorithm from the previous section. There we use the upper bound on the number of vertices in a minimal k-tree. Most of the minimal k-trees have less than $k-1$ leaves, which implies that the upper bound $2k-1$ on the order of a minimal k-tree is too big for the majority of the minimal k-trees. Let $T(k)$ be the running time of IOBA. Then we have

$$T(k) = O^*\biggl(\sum_{k+1 \leqslant k' \leqslant 2k-1} \bigl(\#\text{ of minimal } k\text{-trees on } k' \text{ vertices}\bigr) \times \bigl(6.14^{k'}\bigr)\biggr). \tag{3}$$

A minimal k-tree T' on k' vertices has $k'-k$ leaves, and thus the out-tree T^0 from which T' is constructed has k vertices of which $k'-k$ are leaves. Hence the number of minimal k-trees on k' vertices is the same as the number of non-isomorphic out-trees on k vertices with $k'-k$ leaves. Here an interesting counting problem arises. Let $g(k,l)$ be the number of non-isomorphic out-trees on k vertices with l leaves. Enumerate $g(k,l)$. To our knowledge, such a function has not been studied yet. Leaving it as a challenging open question, here we give an upper bound on $g(k,l)$ and use it for a better analysis of $T(k)$. In particular we are interested in the case when $l \geqslant k/2$.

Consider an out-tree T^0 on $k \geqslant 3$ vertices which has αk internal vertices and $(1-\alpha)k$ leaves. We want to obtain an upper bound on the number of such non-isomorphic out-trees T^0. Let T^c be the subtree of T^0 obtained after deleting all its leaves and suppose that T^c has βk leaves. Assume that $\alpha \leqslant 1/2$ and notice that αk and βk are integers. Clearly $\beta < \alpha$.

Each out-tree T^0 with $(1-\alpha)k$ leaves can be obtained by appending $(1-\alpha)k$ leaves to T^c so that each of the vertices in $\mathrm{Leaf}(T^c)$ has at least one leaf appended to it. Imagine that we have $\beta k = |\mathrm{Leaf}(T^c)|$ and $\alpha k - \beta k = |\mathrm{Int}(T^c)|$ distinct boxes. Then what we are looking for is the number of ways to put $(1-\alpha)k$ balls into the boxes so that each of the first βk boxes is nonempty. Again this is equivalent to putting $(1-\alpha-\beta)k$ balls into αk distinct boxes. It is an easy exercise to see that this number equals $\binom{k-\beta k-1}{\alpha k-1}$.

Note that the above number does not give the exact value for the non-isomorphic out-trees on k vertices with $(1-\alpha)k$ leaves. This is because we treat an out-tree T^c as a labeled one, which may lead us to distinguishing two assignments of balls even though the two corresponding out-trees T^0's are isomorphic to each other.

A minimal k-tree obtained from T^0 has $(1-\alpha)k$ leaves and thus $(2-\alpha)k$ vertices. With the upper bound $O^*(2.95^{\alpha k})$ on the number of T^c's by [13], by (3) we have the following:

$$\begin{aligned}
T(k) &= O^*\biggl(\sum_{\alpha \leqslant 1/2}\sum_{\beta<\alpha} 2.95^{\alpha k}\binom{k-\beta k-1}{\alpha k-1}(6.14)^{(2-\alpha)k}\biggr) + O^*\biggl(\sum_{\alpha>1/2} 2.95^{\alpha k}(6.14)^{(2-\alpha)k}\biggr)\\
&= O^*\biggl(\sum_{\alpha \leqslant 1/2}\sum_{\beta<\alpha} 2.95^{\alpha k}\binom{k}{\alpha k}(6.14)^{(2-\alpha)k}\biggr) + O^*\bigl(2.95^k(6.14)^{3k/2}\bigr)\\
&= O^*\biggl(\sum_{\alpha \leqslant 1/2}\biggl(2.95^{\alpha}\frac{1}{\alpha^{\alpha}(1-\alpha)^{1-\alpha}}(6.14)^{(2-\alpha)}\biggr)^k\biggr) + O^*\bigl(44.9^k\bigr).
\end{aligned}$$

The term in the sum over $\alpha \leqslant 1/2$ above is maximized when $\alpha = \frac{2.95}{2.95+6.14}$, which yields $T(k) = O^*(55.8^k)$. Thus, we conclude with the following theorem.

Theorem 3.5. k-INT-OUT-BRANCHING *is solvable in time* $O^*(55.8^k)$.

4. Conclusion

In this paper we refine the approach of Chen et al. [4] based on randomized Divide-and-Conquer technique. Our technique is based on a more complicated coloring and within this technique we refined the result of Alon et al. [1] for the

k-Out-Tree problem. It is interesting to see if this technique can be used to obtain faster algorithms for other parameterized problems.

As a byproduct of our work, we obtained the first $O^*(2^{O(k)})$-time algorithm for k-Int-Out-Branching. We used the classical result of Otter [13] that the number of non-isomorphic trees on k vertices is $O^*(2.95^k)$. An interesting combinatorial problem is to refine this bound for trees having $\lfloor \alpha k \rfloor$ leaves for some $\alpha < 1$.

References

[1] N. Alon, R. Yuster, U. Zwick, Color-coding, J. ACM 42 (1995) 844–856.
[2] J. Bang-Jensen, G. Gutin, Digraphs: Theory, Algorithms and Applications, 2nd ed., Springer-Verlag, London, 2008.
[3] T. Beyer, S.M. Hedetniemi, Constant time generation of rooted trees, SIAM J. Comput. 9 (1980) 706–712.
[4] J. Chen, J. Kneis, S. Lu, D. Mölle, S. Richter, P. Rossmanith, S.-H. Sze, F. Zhang, Randomized divide-and-conquer: Improved path, matching, and packing algorithms, SIAM J. Comput. 38 (2009) 2526–2547.
[5] F.R.K. Chung, Separator theorems and their applications, in: Paths, Flows, and VLSI-Layout, Bonn, 1988, Springer-Verlag, Berlin, 1990, pp. 17–34.
[6] N. Cohen, F. Fomin, G. Gutin, E.J. Kim, S. Saurabh, A. Yeo, Algorithm for finding k-vertex out-trees and its application to k-internal out-branching problem, in: Proc. COCOON'09, in: Lecture Notes in Comput. Sci., vol. 5609, 2009, pp. 37–46.
[7] A. Demers, A. Downing, Minimum leaf spanning tree, US Patent no. 6,105,018, August 2000.
[8] M.L. Fredman, J. Komlos, E. Szemeredi, Storing a sparse table with $O(1)$ worst case access time, J. ACM 31 (1984) 538–544.
[9] G. Gutin, I. Razgon, E.J. Kim, Minimum leaf out-branching problems, in: Proc. AAIM'08, in: Lecture Notes in Comput. Sci., vol. 5034, 2008, pp. 235–246.
[10] I. Koutis, Faster algebraic algorithms for path and packing problems, in: Proc. ICALP'08, in: Lecture Notes in Comput. Sci., vol. 5125, 2008, pp. 575–586.
[11] M. Naor, L.J. Schulman, A. Srinivasan, Splitters and near-optimal derandomization, in: Proc. 17th Ann. Symp. Found. Comput. Sci., 1995, pp. 182–193.
[12] A. Nilli, Perfect hashing and probability, Combin. Probab. Comput. 3 (1994) 407–409.
[13] R. Otter, The number of trees, Ann. of Math. 49 (1948) 583–599.
[14] E. Prieto, C. Sloper, Either/Or: Using vertex cover structure in designing FPT-algorithms – The case of k-internal spanning tree, in: Proc. WADS'2003, in: Lecture Notes in Comput. Sci., vol. 2748, 2003, pp. 465–483.
[15] E. Prieto, C. Sloper, Reducing to independent set structure – The case of k-internal spanning tree, Nordic J. Comput. 15 (2005) 308–318.
[16] R. Williams, Finding a path of length k in $O^*(2^k)$ time, Inform. Process. Lett. 109 (2009) 315–318.