A Faster Algorithm for Constructing the Frequency Difference Consensus Tree

Jesper Jansson
Kyoto University, Japan

Wing-Kin Sung
The Chinese University of Hong Kong, China
Hong Kong Genome Institute, Hong Kong Science Park, China

Seyed Ali Tabatabaee
Dept. of Computer Science, University of British Columbia, Vancouver, Canada

Yutong Yang
Hong Kong Genome Institute, Hong Kong Science Park, China

Abstract

A consensus tree is a phylogenetic tree that summarizes the evolutionary relationships inferred from a collection of phylogenetic trees with the same set of leaf labels. Among the many types of consensus trees that have been proposed in the last 50 years, the frequency difference consensus tree is one of the more finely resolved types that retains a large amount of information. This paper presents a new deterministic algorithm for constructing the frequency difference consensus tree. Given k phylogenetic trees with identical sets of n leaf labels, it runs in $O(kn \log n)$ time, improving the best previously known solution.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis; Applied computing → Bioinformatics

Keywords and phrases phylogenetic tree, frequency difference consensus tree, tree algorithm, centroid path decomposition, max-Manhattan Skyline Problem

Digital Object Identifier 10.4230/LIPIcs.STACS.2024.43

Supplementary Material *Software*: `https://github.com/tswddd2/FDCT_new`
archived at `swh:1:dir:7cb120cbc61221f740e9e824e93bce0bbf64270a`

Funding This work was partially funded by JSPS KAKENHI grant 22H03550 and NSERC Discovery Grants.

Acknowledgements The authors would like to thank Varun Gupta for some ideas employed in the procedure `Fast_Label_Trees`.

1 Introduction

In phylogenetic analysis, variations in datasets, algorithms, and models of evolution typically yield different phylogenetic trees. Hence, researchers often need to analyze a collection of phylogenetic trees with the same set of leaf labels but different branching structures, and to this end, they use consensus trees. A consensus tree is a single phylogenetic tree that represents a collection of phylogenetic trees, aiming to highlight the commonly agreed-upon parts of the evolutionary history. Consensus trees have applications across various fields of science, including biology, evolutionary studies, epidemiology, and ecology. Many alternative consensus trees, each with its strengths and limitations, have been proposed; see, e.g., [7].

The *frequency difference consensus tree* (FDCT) [16] has garnered interest among researchers in recent years [15, 17, 20]. Given k phylogenetic trees with identical sets of n leaf labels, the FDCT is a phylogenetic tree consisting of each cluster that occurs more frequently


in the input trees than any single cluster incompatible with it. In this context, a cluster refers to any nonempty subset of the leaf label set and is said to occur in a phylogenetic tree iff it corresponds to the set of all leaf labels descending from a single node of the tree. Furthermore, two clusters are deemed incompatible if they cannot simultaneously occur in the same phylogenetic tree. The advantage of the FDCT compared to some other popular types of consensus trees, such as the strict consensus tree [30] and the majority rule consensus tree [25], is that it captures more of the shared branching information and has more clusters.

It is evident that $\Omega(kn)$ serves as a lower bound for the running time of any algorithm aiming to build the FDCT, given that it corresponds to the input size. Unlike certain other types of consensus trees such as the strict consensus tree and the majority rule consensus tree, there has been no algorithm proposed to construct the FDCT that can achieve a running time matching this lower bound. Before this paper, the $O(kn \log^2 n)$-time algorithm by Gawrychowski et al. [15] was the asymptotically fastest algorithm for constructing the FDCT. In this paper, we present an $O(kn \log n)$-time algorithm for constructing the FDCT, thus reducing the gap between the upper and lower bounds for the running time of the fastest algorithm to solve this problem.

The overarching structure of our new algorithm follows the framework proposed by Jansson et al. [20] for computing the FDCT. By improving the methods for solving two subproblems in [20], our algorithm achieves a running time of $O(kn \log n)$. First, our algorithm incorporates a novel divide-and-conquer solution that runs in $O(kn \log n)$ time for the weighting step, where the number of phylogenetic trees in which each cluster occurs is calculated. We remark that algorithms for computing other types of consensus trees such as the *greedy consensus tree* [7, 13] involve the same weighting step [15, 21, 32] and may derive advantages from our improved method. Second, the running time of the procedure `Filter_Clusters` is improved to $O(n \log n)$ by solving instances of the Max-Manhattan Skyline Problem [9] to identify the clusters that should be removed at each recursive stage of the algorithm (refer to Sections 3 and 5 for the detailed explanation).

1.1 Definitions and Notation

A *phylogenetic tree* is a rooted tree that represents the evolutionary relationships among different organisms. Every internal node of a phylogenetic tree has at least two unordered children and every leaf has a distinct label. The term *trees* will be employed as a shorthand for *phylogenetic trees* in the remainder of this paper. Let T be some tree. The set of nodes of T is denoted by $V(T)$. Let $\Lambda(T)$ be the set of leaf labels of T. Non-empty subsets of $\Lambda(T)$ are called *clusters*. Clusters with cardinality 1 or $|\Lambda(T)|$ are *trivial clusters*. For any node $u \in V(T)$, $T[u]$ is the subtree of T rooted at u and $\Lambda(T[u])$ is the set of leaf labels of $T[u]$, called the cluster *associated* with u. The *cluster collection* of T, denoted by $\mathcal{C}(T)$, is the set $\bigcup_{u \in V(T)} \{\Lambda(T[u])\}$. Any cluster $C \subseteq \Lambda(T)$ *occurs* in T iff $C \in \mathcal{C}(T)$. For any nodes $u, v \in V(T)$, we denote the lowest common ancestor of u and v in T by $lca^T(u, v)$. Further, for any non-empty set of nodes $U \subseteq V(T)$, we refer to the lowest common ancestor of all these nodes in T by $lca^T(U)$.

Any two clusters $C_1, C_2 \subseteq \Lambda(T)$ are said to be *compatible*, written as $C_1 \smile C_2$, iff $C_1 \subseteq C_2$, $C_2 \subseteq C_1$, or $C_1 \cap C_2 = \emptyset$. If C_1 and C_2 satisfy none of the preceding properties, then they are *incompatible*, denoted as $C_1 \not\smile C_2$. Similarly, given trees T_1 and T_2 with identical leaf label sets, and nodes $u \in V(T_1)$ and $v \in V(T_2)$, we say u is compatible with v, denoted as $u \smile v$, if the clusters associated with u and v are compatible. We now extend the notion of compatibility to trees. A cluster $C \subseteq \Lambda(T)$ is *compatible* with T (denoted as $C \smile T$) iff for every $C' \in \mathcal{C}(T)$, we have $C \smile C'$. Further, two trees T_1 and T_2 with

identical leaf label sets are *compatible* (denoted as $T_1 \smile T_2$) iff for every $C \in \mathcal{C}(T_1)$, $C \smile T_2$, i.e. iff every cluster in T_1 is compatible with T_2. This also means that every cluster in T_2 is compatible with T_1.

The *frequency difference consensus tree* (FDCT) is defined as follows. Let $\mathcal{S}$ be a set of k trees with identical sets of n leaf labels, i.e. $\mathcal{S} = \{T_1, T_2, \ldots, T_k\}$ and $\Lambda(T_1) = \Lambda(T_2) = \ldots = \Lambda(T_k) = L$ (where $|L| = n$). For any cluster $C \subseteq L$, let the *weight* of C, denoted as $w(C)$, be $|\{T : T \in \mathcal{S} \text{ and } C \in \mathcal{C}(T)\}|$, i.e., the number of trees in $\mathcal{S}$ in which C occurs. For any tree $T \in \mathcal{S}$ and any node $u \in V(T)$, we define the weight of node u as $w(u) = w(\Lambda(T[u]))$. Then, the FDCT of $\mathcal{S}$ is the tree T_{FD}, where $\mathcal{C}(T_{FD}) = \{C : C \subseteq L \text{ and } w(C) > max(\{w(C') : C' \subseteq L \text{ and } C \not\sim C'\})\}$. Thus, T_{FD} contains every cluster that occurs more frequently than any cluster incompatible with it (we refer to such clusters as *frequency difference clusters*). By Proposition 3 in [31], this tree always exists and is unique for a given $\mathcal{S}$. Figure 1 illustrates the FDCT for a collection of four phylogenetic trees.

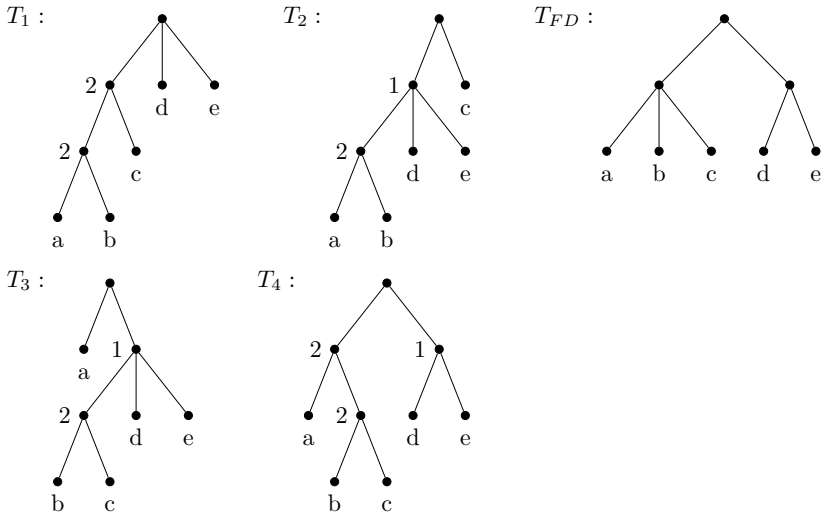

Figure 1 Let $\mathcal{S} = \{T_1, T_2, T_3, T_4\}$. T_{FD} is the FDCT of $\mathcal{S}$. In each T_i, the number beside each non-root internal node u indicates the weight $w(u)$. In this example, T_{FD} does not contain the clusters $\{a, b\}$ and $\{b, c\}$ with weight 2 because they are incompatible with each other. On the other hand, T_{FD} contains the cluster $\{d, e\}$ with weight 1. Furthermore, T_{FD} contains the cluster $\{a, b, c\}$ with weight 2, even though that cluster is incompatible with two input trees.

1.2 Previous Work

A variety of types of consensus trees have been developed over the last half-century, starting with the Adams consensus tree [2] in 1972. Some of these consensus trees are summarized in [7]. Here, we describe two well-studied types of consensus trees: the strict consensus tree [30] and the majority-rule consensus tree [25]. The strict consensus tree keeps only the clusters that occur in all input trees and is easily computed in optimal $O(kn)$ running time [10]. However, some potentially important clusters might be discarded from this consensus tree, if one of the input trees does not contain them. The *majority-rule consensus tree* is a generalization of the previous consensus tree and contains all clusters that occur in more than half of the trees. The majority-rule consensus tree can also be computed in optimal $O(kn)$ time [21].

The *frequency difference consensus tree* (FDCT) was introduced by Goloboff et al. [16] as a more informative alternative that contains not only the clusters that occur in the majority of trees but also the other frequency difference clusters. Dong et al. [11] provided a comparison of the FDCT and a few other types of consensus trees. Barrett et al. [3] employed the idea of using the frequency difference metric while analyzing angiosperm phylogeny. Steel and Velasco [31] investigated a generalization of the FDCT to *supertrees*, i.e. consensus trees built from input trees that do not necessarily have the same leaf label sets. They showed that, unlike some other commonly used definitions, the FDCT easily generalizes to a viable supertree definition. Moreover, the FDCT has been utilized in various other studies over the years [14, 18, 19, 23, 24, 26, 27, 28, 29].

Several research works have focused on computing the FDCT of a set of k trees, each labeled by the same set of n leaf labels. An implementation of the FDCT can be found in the free software package TNT [17]; however, the algorithm employed by TNT and its complexity remain undisclosed. Jansson et al. [20] gave a deterministic $min\{O(kn^2), O(kn(k+\log^2 n))\}$-time algorithm for constructing the FDCT. This algorithm was implemented in the open-source FACT package [21] and experimentally shown [20] to be significantly faster than TNT's implementation. Subsequently, Gawrychowski et al. [15] developed a faster method for the weighting step in [20], yielding an improved running time of $O(kn \log^2 n)$ for constructing the FDCT.

1.3 Organization of the Article

This paper is organized as follows. Section 2 contains some results from previous works that are utilized later in this paper. Section 3 gives the framework of the $O(kn \log n)$-time algorithm for computing the FDCT. Sections 4 and 5 present algorithms for solving the subproblems of the FDCT construction. Finally, Section 6 provides the concluding remarks.

2 Preliminaries

2.1 The *delete* Operation

The *delete* operation on a non-root internal node u in a tree T makes all children of u become children of u's parent and then removes u along with all edges connected to it. After applying this operation on u, the cluster $\Lambda(T[u])$ is removed from the cluster collection of T. If c denotes the number of u's children, then the *delete* operation on u takes $O(c)$ time.

2.2 The Lowest Common Ancestor

We restate the following lemma outlining the *lowest common ancestor (lca)* operation from [4]:

▶ **Lemma 1.** *Given any tree T, the lca data structure can be constructed in $O(n)$ time, where $n = |V(T)|$. Then, for any nodes $u, v \in V(T)$, the query $lca^T(u, v)$ can be answered in constant time.*

2.3 Restriction of Trees

For any tree T and any cluster $C \subseteq \Lambda(T)$, we define $T|C$ (called T *restricted to* C) as the tree T' with $V(T') = \{lca^T(u, v) : u, v \in C\}$ such that $lca^T(C') = lca^{T'}(C')$ for all $C' \subseteq C$. Intuitively, T' has the leaf label set C and consists of all nodes in T that are *lca*'s of the leaves in C, and the ancestral relationships between the nodes in T' are the same as they were in T. Figure 2 provides an example of restricting a tree to different clusters.

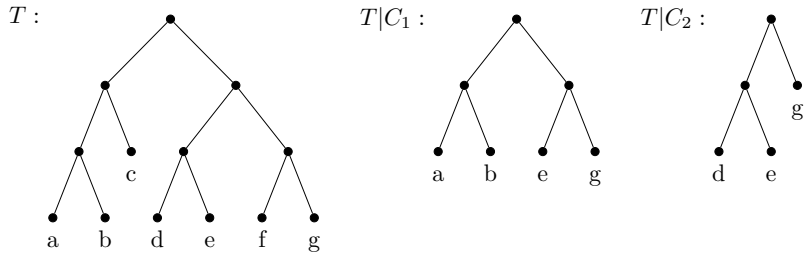

Figure 2 Illustration of restricting a tree to different clusters, where $C_1 = \{a, b, e, g\}$ and $C_2 = \{d, e, g\}$.

2.4 Expanded Restriction of Trees

Recall that every node in the tree T is associated with a weight. For any subset $C \subseteq L$, when we compute the restricted tree $T|C$, some nodes in T are deleted. This leads to losing information about the weights of the clusters associated with these nodes. To address this, we extend the concept of restricted trees following the definition given in [20] and allow some nodes to be marked as *spoiled* (see below for details). For any $C \subseteq \Lambda(T)$, we obtain the weighted tree $T||C$ (called the *expanded restriction of* T *to* C) as follows:

1. Let $T' = T|C$.
2. For every node u in T', set the weight of u equal to its weight in T and mark u as spoiled if $\Lambda(T'[u]) \neq \Lambda(T[u])$ or if u is a spoiled node in T.
3. For every edge (u, v) in T', let P be the path in T between u and v, excluding u and v. If P contains at least one node, then create a new node z in T' (referred to as a *path node*), replace the edge (u, v) with the two edges (u, z) and (z, v), assign the weight of z to the highest weight among all nodes in P, and mark z as spoiled.
4. Let $T||C = T'$.

Intuitively, a node u in $T|C$ that was not already spoiled in T becomes spoiled in $T||C$ if at least one leaf label in $\Lambda(T[u])$ is not in C. It follows that if a node becomes spoiled in $T||C$, then all of its ancestors become spoiled. Furthermore, every path node is a spoiled node (but not vice versa). The purpose of the path nodes in $T||C$ is to compactly represent the weights of the clusters that were lost when building the restriction of T to C and that may conflict with clusters that are subsets of C. Figure 3 shows an example of the expanded restriction of trees.

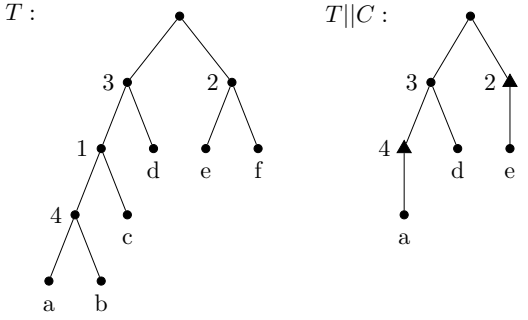

Figure 3 Illustration of the expanded restriction of a tree, where $C = \{a, d, e\}$. Internal nodes are labeled with their weights. Nodes represented by triangles are path nodes.

We extend the definition of compatibility to spoiled nodes. Suppose that $C_1, C_2 \subseteq \Lambda(T)$ and that u is a spoiled node in $T||C_1$. We have $C_2 \smile u$ iff C_2 and $\Lambda((T||C_1)[u])$ are disjoint or $C_2 \subseteq \Lambda((T||C_1)[u])$. Observe that if $\Lambda((T||C_1)[u]) \subsetneq C_2$, then $C_2 \not\smile u$, i.e., the set inclusion relations in the definition of compatibility are asymmetric for spoiled nodes.

The expanded restrictions can be computed efficiently according to the following lemma:

▶ **Lemma 2.** *Let T be a weighted phylogenetic tree with n leaves. After $O(n \log n)$ time preprocessing, for any partition $C_1, C_2, \ldots, C_q$ of $\Lambda(T)$, the trees $T||C_i$ for all $i \in \{1, 2, \ldots, q\}$ can be constructed in a total of $O(n)$ time.*

Proof. By Lemma 5.2 of [12], the trees $T|C_i$ for all $i \in \{1, 2, \ldots, q\}$ can be constructed in a total of $O(n)$ time. By Theorem 2 of [22], after an $O(n \log n)$-time preprocessing of T, the maximum weight of all nodes along the path between any two nodes u and v in T can be found in $O(1)$ time. Furthermore, after an $O(n)$-time preprocessing of T, the second node on the path from any node u in T to any other node v in T can be found in $O(1)$ time (this can be achieved by finding level ancestors [5, 6]). Hence, for every edge (u, v) in $T|C_i$ (for any $i \in \{1, 2, \ldots, q\}$), we can compute the maximum weight of all nodes along the path between u and v in T, excluding u and v, in $O(1)$ time. Consequently, $T||C_i$ can be constructed in $O(|C_i|)$ time from $T|C_i$ for any $i \in \{1, 2, \ldots, q\}$, which completes the proof. ◀

2.5 Merging Trees

Given two trees T_1 and T_2 where $\Lambda(T_1) = \Lambda(T_2) = L$ and $T_1 \smile T_2$, `Merge_Trees`(T_1, T_2) returns a tree T such that $\Lambda(T) = L$ and $\mathcal{C}(T) = \mathcal{C}(T_1) \cup \mathcal{C}(T_2)$. Jansson et al. [21] showed that `Merge_Trees` can be computed in $O(|L|)$ time. Figure 4 gives an example for `Merge_Trees`.

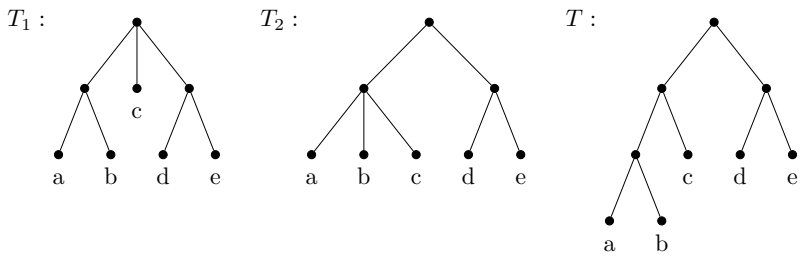

Figure 4 Illustration of merging trees where $T =$ `Merge_Trees`(T_1, T_2).

2.6 The Max-Manhattan Skyline Problem

Given a set $\mathcal{S}$ of $O(n)$ subintervals of $[1, n-1]$ with positive integer heights of size $O(n)$, the Max-Manhattan Skyline Problem asks for a table f such that for $t \in [1, n-1]$, $f[t] = \max\{height([i, j]) : t \in [i, j], [i, j] \in \mathcal{S}\}$. Crochemore *et al.* [9] gave an $O(n)$-time algorithm for the Min-Manhattan Skyline Problem, defined in a similar way as the Max-Manhattan Skyline Problem, except that it seeks the minimum height instead of the maximum in the definition of the output table f. Their algorithm first sorts the intervals according to their heights in non-decreasing order. Then, for each interval $[i, j]$ in this order, it sets the values of $f[t]$ for all positions $t \in [i, j]$ that have not yet been assigned a value to $height([i, j])$. By modifying Crochemore *et al.*'s algorithm [9] to sort the intervals in non-increasing order, we immediately have:

▶ **Lemma 3.** *The* Max-Manhattan Skyline Problem *can be solved in $O(n)$ time.*

2.7 Centroid Paths and Side Trees

A *centroid path* [8] of a tree T is a path of the form $\pi = \langle p_\alpha, p_{\alpha-1}, \ldots, p_1 \rangle$, where p_α is any node in T, the node p_{i-1} for every $i \in \{2, \ldots, \alpha\}$ is any child of p_i with the maximum number of leaf descendants (ties are broken arbitrarily), and p_1 is a leaf. Suppose that π is a centroid path of T. For any $u \in V(T)$ such that u does not belong to π but the parent of u does, $T[u]$ is called a *side tree* of π. From these definitions, we can derive the following lemma:

▶ **Lemma 4.** *Let T be a tree and τ a side tree of a centroid path that starts at the root of T. Then $|\Lambda(\tau)| \leq |\Lambda(T)|/2$.*

By computing a centroid path π starting at the root of T and recursively applying this procedure to the side trees of π, we obtain a *centroid path decomposition* of T. It follows from Lemma 4 that the number of recursion levels needed to complete such a decomposition is $O(\log |\Lambda(T)|)$.

3 Algorithm `Fast_Frequency_Difference`

The pseudocode of our new algorithm, named `Fast_Frequency_Difference`, is shown in Algorithm 1. Its overall structure follows the framework developed in [20] for constructing the FDCT, which we review next.

Algorithm 1 The algorithm `Fast_Frequency_Difference` for constructing the FDCT, which maintains the overall structure established by the algorithm `Frequency_Difference` in [20].

Algorithm `Fast_Frequency_Difference`

Input: A set $\mathcal{S} = \{T_1, T_2, \ldots, T_k\}$ of trees with $\Lambda(T_1) = \Lambda(T_2) = \ldots = \Lambda(T_k) = L$.

Output: The frequency difference consensus tree of $\mathcal{S}$.

```
   /* Preprocessing */
1  Fast_Compute_Weights(S)

   /* Main algorithm */
2  T := T_1
3  for j := 2 to k do
       A := Fast_Filter_Clusters(T, T_j)
       B := Fast_Filter_Clusters(T_j, T)
       T := Merge_Trees(A, B)
   endfor
4  for j := 1 to k do
       T := Fast_Filter_Clusters(T, T_j)
5  return T
End Fast_Frequency_Difference
```

Suppose that the input is $\mathcal{S} = \{T_1, T_2, \ldots, T_k\}$. The basic idea of the algorithm in [20] is to initially let T be a copy of T_1 and then consider the other trees one by one while updating the clusters of T accordingly. More precisely, when considering any such tree T_j, the algorithm deletes every cluster in T that is incompatible with a cluster in T_j of equal or higher weight, and also inserts every cluster from T_j into T that could potentially be a frequency cluster but is not already in T. This strategy produces a tree T whose set of clusters is a superset of the set of the frequency difference clusters, so the algorithm applies a final postprocessing step to delete all non-frequency difference clusters from T.

To update T in each iteration, the procedure `Merge_Trees`, described in Section 2.5, and a procedure called `Filter_Clusters` are used. The latter takes as input two trees T_A and T_B with identical leaf label sets and outputs a copy of T_A from which every cluster that is incompatible with a cluster in T_B of equal or higher weight has been deleted.

Our new algorithm `Fast_Frequency_Difference` improves the time complexity of the previous algorithm from [20] by replacing the preprocessing step for computing the weights of all clusters occurring in $\mathcal{S}$ (Step 1) and the procedure `Filter_Clusters` (used in Steps 3 and 4) by more efficient solutions, referred to as `Fast_Compute_Weights` and `Fast_Filter_Clusters` below.

In Section 4, we will prove the following theorem concerning the correctness and time complexity of the procedure `Fast_Compute_Weights`:

▶ **Theorem 5.** *Given a set $\mathcal{S} = \{T_1, T_2, \ldots, T_k\}$ of trees with identical sets of n leaf labels, the procedure* `Fast_Compute_Weights`$(\mathcal{S})$ *calculates the weights of all clusters occurring in $\mathcal{S}$ in $O(kn \log n)$ time.*

Moreover, Section 5 contains the proof for the following theorem regarding the correctness and time complexity of the procedure `Fast_Filter_Clusters`:

▶ **Theorem 6.** *Given two weighted trees T_A and T_B with identical sets of n leaf labels, the procedure* `Fast_Filter_Clusters`(T_A, T_B) *filters out clusters as needed in $O(n \log n)$ time.*

On the grounds of the two theorems stated above, we can prove the main theorem of this paper:

▶ **Theorem 7.** *Given a set $\mathcal{S} = \{T_1, T_2, \ldots, T_k\}$ of trees with identical sets of n leaf labels, the algorithm* `Fast_Frequency_Difference`$(\mathcal{S})$ *constructs the FDCT of $\mathcal{S}$ in $O(kn \log n)$ time.*

Proof. Assuming that the improved procedures function as intended, the correctness of `Fast_Frequency_Difference` follows from that of `Frequency_Difference` proved by [20].

Now, we analyze the time complexity of the algorithm. Step 1 makes a call to the procedure `Fast_Compute_Weights`, which takes $O(kn \log n)$ time, according to Theorem 5. Furthermore, Steps 3 and 4 make $O(k)$ calls to the procedures `Fast_Filter_Clusters` and `Merge_Trees`. By Theorem 6, each call to the procedure `Fast_Filter_Clusters` takes $O(n \log n)$ time. Furthermore, the procedure `Merge_Trees` runs in $O(n)$ time [21]. Hence, Steps 3 and 4 take $O(kn \log n)$ time. Consequently, the running time of the algorithm is $O(kn \log n)$. ◀

4 Procedure `Fast_Compute_Weights`

We break down the improved procedure `Fast_Compute_Weights` into two phases called labeling and counting, similar to the strategy used in [15]. The procedure `Fast_Label_Trees` is responsible for the labeling phase. This procedure assigns an integer label to each node u in every tree in $\mathcal{S}$, denoted by $id(u)$, such that $id(u) \in \{1, \ldots, 2kn\}$ and that for any other node u' in any tree in $\mathcal{S}$, $id(u) = id(u')$ iff the clusters associated with u and u' are the same. Next, during the counting phase, the labels are sorted using counting sort, the count of each distinct label is determined, and the weight of each cluster is obtained from the count of the label of its associated node. Algorithm 2 presents the pseudocode for the procedure `Fast_Compute_Weights`.

Algorithm 2 The procedure `Fast_Compute_Weights`.

Algorithm `Fast_Compute_Weights`

Input: A set $\mathcal{S} = \{T_1, T_2, \ldots, T_k\}$ of trees with $\Lambda(T_1) = \Lambda(T_2) = \ldots = \Lambda(T_k) = L$.

Output: Compute the weight of each cluster C occurring in $\mathcal{S}$, i.e., the number of trees in $\mathcal{S}$ in which C occurs.

/* Labeling phase */

1 `Fast_Label_Trees`($\mathcal{S}$)

/* Counting phase */

2 Sort the obtained labels using counting sort.

3 Determine the count of each distinct label.

4 Set the weight of each cluster to the count of the label of its associated node.

End `Fast_Compute_Weights`

The procedure `Fast_Label_Trees` uses a divide-and-conquer approach to carry out the labeling phase. Let L' and L'' form a partition of L such that the difference between $|L'|$ and $|L''|$ is at most one. `Fast_Label_Trees`($\{T_1, \ldots, T_k\}$) recursively calls `Fast_Label_Trees`($\{T_1|L', \ldots, T_k|L'\}$) and `Fast_Label_Trees`($\{T_1|L'', \ldots, T_k|L''\}$) to obtain the labels for all nodes in $T_i|L'$ and $T_i|L''$ for all $i \in \{1, 2, \ldots, k\}$. Then, for each node u in any tree $T_i \in \mathcal{S}$, the pair $(id(\varphi_u^{L'}), id(\varphi_u^{L''}))$ is assigned to u, where φ_u^C for any $C \subseteq L$ denotes the node in $T_i|C$ that corresponds to u (if such node does not exist in $T_i|C$, then φ_u^C is set to Φ, a special node with $id(\Phi) = 0$). Next, all pairs are sorted using radix sort, and a positive integer rank is assigned to each unique pair. Finally, for each node u in any tree $T_i \in \mathcal{S}$, $id(u)$ is set to the rank of the pair $(id(\varphi_u^{L'}), id(\varphi_u^{L''}))$. The pseudocode for the procedure `Fast_Label_Trees` is given in Algorithm 3. Moreover, Figure 5 illustrates one iteration of `Fast_Label_Trees`.

Algorithm 3 The procedure `Fast_Label_Trees`.

Algorithm `Fast_Label_Trees`

Input: A set $\mathcal{S} = \{T_1, T_2, \ldots, T_k\}$ of trees with $\Lambda(T_1) = \Lambda(T_2) = \ldots = \Lambda(T_k) = L$.

Output: Label each node u in a tree in $\mathcal{S}$ with $id(u) \in \{1, \ldots, 2k|L|\}$ such that two nodes in different trees receive the same label iff the clusters associated with them are the same.

1 **if** $|L| = 1$ **then**

/* Base case (each tree has only one node) */

For each node u in any tree in $\mathcal{S}$, set $id(u) = 1$.

return

endif

2 Partition L into L' and L'', such that the difference between $|L'|$ and $|L''|$ is at most one.

3 For all $i \in [1 \ldots k]$, let $T_i' = T_i|L'$ and $T_i'' = T_i|L''$.

4 `Fast_Label_Trees`($\{T_1', T_2', \ldots, T_k'\}$).

5 `Fast_Label_Trees`($\{T_1'', T_2'', \ldots, T_k''\}$).

6 For each node u in any tree $T_i \in \mathcal{S}$, assign the pair $(id(\varphi_u^{L'}), id(\varphi_u^{L''}))$ to u.

7 Sort all the obtained pairs using radix sort, remove duplicates, and assign a rank to each unique pair.

8 For each node u in any tree $T_i \in \mathcal{S}$, set $id(u)$ to the rank of the pair $(id(\varphi_u^{L'}), id(\varphi_u^{L''}))$.

End `Fast_Label_Trees`

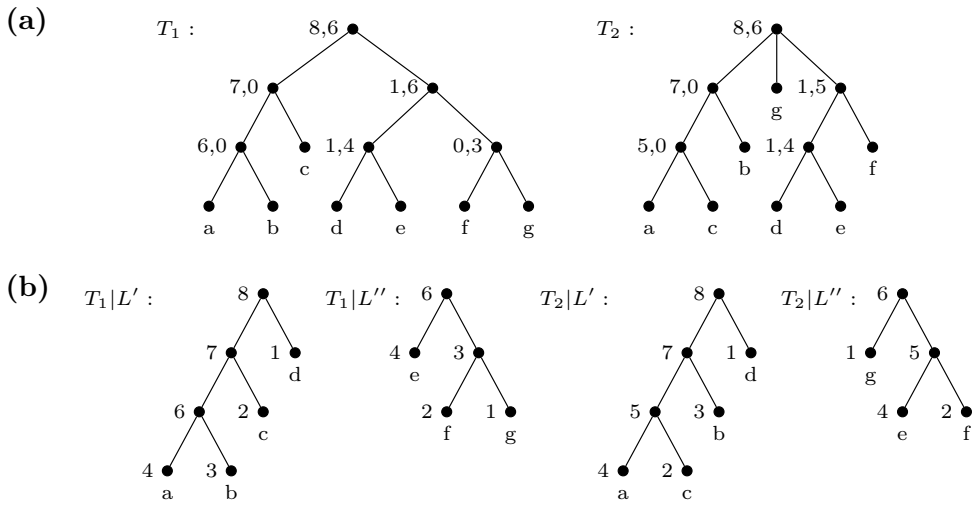

Figure 5 Illustration of one iteration of `Fast_Label_Trees`($\{T_1, T_2\}$), where $L' = \{a, b, c, d\}$ and $L'' = \{e, f, g\}$. Part (a) shows the trees T_1 and T_2, where the pair assigned to each node is presented beside it (except for the leaves). Part (b) shows the recursively labeled trees $T_1|L'$, $T_1|L''$, $T_2|L'$, and $T_2|L''$, where the labels are presented beside the nodes.

The following lemma proves the correctness of the procedure `Fast_Label_Trees`:

▶ **Lemma 8.** *Given a set $\mathcal{S} = \{T_1, T_2, \ldots, T_k\}$ of trees with identical sets of n leaf labels, the following statements hold after running the procedure* `Fast_Label_Trees`($\mathcal{S}$)*:*

1. *For any node $u \in V(T_i)$ where $T_i \in \mathcal{S}$, we have $id(u) \in \{1, \ldots, 2k|L|\}$.*
2. *For any two nodes $u \in V(T_i)$ and $v \in V(T_j)$ where $T_i, T_j \in \mathcal{S}$, we have $id(u) = id(v)$ iff $\Lambda(T_i[u]) = \Lambda(T_j[v])$.*

Proof. We start by showing that the first statement holds. It can be easily seen that $|V(T_i)| < 2|L|$ for any $T_i \in \mathcal{S}$. Thus, the total number of nodes in all trees is less than $2k|L|$. Consequently, the label of each node is in $\{1, \ldots, 2k|L|\}$.

To prove the second statement, we use induction on $|L|$. The base case of $|L| = 1$ holds because all nodes have the same cluster associated with them and receive the same label. The induction hypothesis states that if $|L| \leq k$ for some $k \geq 1$, then we have $id(u) = id(v)$ iff $\Lambda^{T_i}(u) = \Lambda^{T_j}(v)$. Now, we want to prove the statement for $|L| = k + 1$.

If $\Lambda(T_i[u]) = \Lambda(T_j[v])$, we have $\Lambda((T_i|L')[\varphi_u^{L'}]) = \Lambda((T_j|L')[\varphi_v^{L'}])$ and $\Lambda((T_i|L'')[\varphi_u^{L''}]) = \Lambda((T_j|L'')[\varphi_v^{L''}])$. Hence, considering that $|L'| \leq k$ and $|L''| \leq k$, we can apply the induction hypothesis to state that $id(\varphi_u^{L'}) = id(\varphi_v^{L'})$ and $id(\varphi_u^{L''}) = id(\varphi_v^{L''})$. Consequently, we have $(id(\varphi_u^{L'}), id(\varphi_u^{L''})) = (id(\varphi_v^{L'}), id(\varphi_v^{L''}))$ and thereby, $id(u) = id(v)$.

On the other hand, if $id(u) = id(v)$, we have $(id(\varphi_u^{L'}), id(\varphi_u^{L''})) = (id(\varphi_v^{L'}), id(\varphi_v^{L''}))$. As a result, we have $id(\varphi_u^{L'}) = id(\varphi_v^{L'})$ and $id(\varphi_u^{L''}) = id(\varphi_v^{L''})$. Therefore, considering that $|L'| \leq k$ and $|L''| \leq k$, we can use the induction hypothesis to deduce that $\Lambda((T_i|L')[\varphi_u^{L'}]) = \Lambda((T_j|L')[\varphi_v^{L'}])$ and $\Lambda((T_i|L'')[\varphi_u^{L''}]) = \Lambda((T_j|L'')[\varphi_v^{L''}])$. Thus, we have $\Lambda(T_i[u]) = \Lambda(T_j[v])$. ◀

Next, we analyze the time complexity of the procedure `Fast_Label_Trees`:

▶ **Lemma 9.** *Given a set $\mathcal{S} = \{T_1, T_2, \ldots, T_k\}$ of trees with identical sets of n leaf labels, the procedure* `Fast_Label_Trees`($\mathcal{S}$) *runs in $O(kn \log n)$ time.*

Proof. Let $T(n)$ be the running time of `Fast_Label_Trees`$(\mathcal{S})$. By Lemma 5.2 of [12], the construction of T_i' and T_i'' takes $O(n)$ time for each $T_i \in \mathcal{S}$, and thereby, a total of $O(kn)$ time for all of the trees. Computing $id(\varphi_u^{L'})$ and $id(\varphi_u^{L''})$ for each node u in some tree $T_i \in \mathcal{S}$ can be done by a bottom up traversal of T_i, along with T_i' and T_i'', in a total of $O(kn)$ time. The number of obtained pairs is $O(kn)$. Furthermore, we can deduce from Lemma 8 that all values in the pairs are in $\{0, 1, \ldots, O(kn)\}$. Thus, sorting these pairs using radix sort and assigning labels to the nodes take $O(kn)$ time. Therefore, we have $T(n) = 2T(n/2) + O(kn)$, and consequently, $T(n) = O(kn \log n)$. ◀

Now, we can prove Theorem 5, regarding the correctness and time complexity of the procedure `Fast_Compute_Weights`:

▶ **Theorem 5.** *Given a set $\mathcal{S} = \{T_1, T_2, \ldots, T_k\}$ of trees with identical sets of n leaf labels, the procedure* `Fast_Compute_Weights`$(\mathcal{S})$ *calculates the weights of all clusters occurring in $\mathcal{S}$ in $O(kn \log n)$ time.*

Proof. We start by proving that the procedure `Fast_Compute_Weights` works correctly. The correctness of Step 1, making a call to the procedure `Fast_Label_Trees`, follows from Lemma 8. In the following steps, the weight of each cluster is set to the count of the label of its associated node, indicating the number of trees in $\mathcal{S}$ in which that cluster occurs.

Now, we analyze the time complexity. As shown in Lemma 9, assigning labels to each node in Step 1 takes $O(kn \log n)$ time. Considering that there are $O(kn)$ labels in total and each label is in $\{1, \ldots, 2kn\}$, Step 2 (counting sort) takes $O(kn)$ time. Furthermore, it is easy to see that Steps 3 and 4 take $O(kn)$ time each. Therefore, the running time of the procedure is $O(kn \log n)$. ◀

5 Procedure `Fast_Filter_Clusters`

On a high level, our new procedure `Fast_Filter_Clusters`, with an improved running time of $O(n \log n)$, follows a similar approach as the $O(n \log^2 n)$-time procedure `Filter_Clusters` in [20]. The objective is to build a tree T whose cluster collection is $\mathcal{C}(T) = \{\Lambda(T_A[u]) : u \in V(T_A)$ and $w(u) > w(x)$ for every $x \in V(T_B)$ with $\Lambda(T_A[u]) \not\sim \Lambda(T_B[x])\}$, i.e., to delete every cluster u in T_A that conflicts with at least one cluster in T_B with a weight greater than or equal to that of u. To do this, both procedures apply the centroid path decomposition technique to divide T_A into a centroid path $\pi = \langle p_\alpha, p_{\alpha-1}, \ldots, p_1 \rangle$, where p_α is the root of T_A, and the set $\sigma(\pi)$ of side trees of π. Since each cluster in T_A is either located inside a side tree of π or associated with a node belonging to π, the cluster collection $\mathcal{C}(T_A)$ may be expressed recursively as:

$$\mathcal{C}(T_A) = \bigcup_{\tau \in \sigma(\pi)} \mathcal{C}(\tau) \cup \bigcup_{p_i \in \pi} \{\Lambda(T_A[p_i])\}. \tag{1}$$

Following this key observation, to check all clusters of T_A in order to decide which ones to delete, the procedures handle the side trees of π recursively and the clusters associated with π directly.

The main difference between `Filter_Clusters` from [20] and `Fast_Filter_Clusters` presented here is how they handle the clusters $\bigcup_{p_i \in \pi} \{\Lambda(T_A[p_i])\}$. The former uses a dynamic data structure to keep track of the currently conflicting nodes from T_B while traversing π upwards and retrieving the heaviest conflicting cluster at each step, taking $O(n \log n)$ time to process all the clusters associated with π. In addition, the time taken to set up the recursive calls to the side trees is $O(n)$. Since the total time spent on each recursion level is $O(n \log n)$ and there are $O(\log n)$ recursion levels, the time complexity of `Filter_Clusters` is

$O(n \log^2 n)$. In contrast, `Fast_Filter_Clusters` detects conflicts between clusters associated with π and clusters in T_B by representing clusters as suitably defined integer intervals. It then solves an instance of the Max-Manhattan Skyline Problem to identify the heaviest conflicting clusters. We will show that this method requires $O(n)$ time to handle all of the clusters associated with π. Thus, each recursion level takes $O(n)$ time, and the total running time becomes $O(n \log n)$.

The pseudocode of `Fast_Filter_Clusters` is presented in Algorithm 4.

Algorithm 4 The procedure `Fast_Filter_Clusters`.

Algorithm `Fast_Filter_Clusters`

Input: Two weighted trees T_A and T_B with $\Lambda(T_A) = \Lambda(T_B) = L$ such that every $u \in V(T_A) \cup V(T_B)$ has a positive integer weight $w(u)$, and that some nodes in T_B may be spoiled.

Output: A tree T with $\Lambda(T) = L$ and $\mathcal{C}(T) = \{\Lambda(T_A[u]) : u \in V(T_A)$ and $w(u) > w(x)$ for every $x \in V(T_B)$ with $\Lambda(T_A[u]) \not\sim \Lambda(T_B[x])\}$.

1 Compute a centroid path $\pi = \langle p_\alpha, p_{\alpha-1}, \ldots, p_1 \rangle$ of T_A, where p_α is the root of T_A and p_1 is a leaf, and compute the set $\sigma(\pi)$ of side trees of π.

/* Handling the side trees */

2 **for** each side tree $\tau \in \sigma(\pi)$ **do**
  Construct $T_B||\Lambda(\tau)$.
  Temporarily change the node weights in τ and $T_B||\Lambda(\tau)$ by sorting them in nondecreasing order and setting each node weight equal to its rank.
  Let $\tau' :=$ `Fast_Filter_Clusters`$(\tau, T_B||\Lambda(\tau))$.
  Replace τ by τ' in T_A and restore the node weights in τ'.
endfor

/* Handling the centroid path */

3 **for** $i = 1$ **to** α **do**
  Compute $n_i := |\Lambda(T_A[p_i])|$.

4 Temporarily relabel the leaf labels in L by the positive integers $\{1, 2, \ldots, n_\alpha\}$ in a way that makes π become a stratifying path in T_A.

5 Compute and store, for every $v \in V(T_B)$, the values $m(v) = \min\{\Lambda(T_B[v])\}$ and $M(v) = \max\{\Lambda(T_B[v])\}$. For any $v \in V(T_B)$, if v is a spoiled node, then set $M(v) = n_\alpha + 1$.

6 Compute and store, for every $v \in V(T_B)$, the value *filled*(v) equivalent to the largest integer x such that $\{1, 2, \ldots, x\} \subseteq \Lambda(T_B[v])$.

7 Create a set I of weighted intervals over $\{1, 2, \ldots, n_\alpha + 1\}$ as follows:
  For each $v \in V(T_B)$, make an interval $[v_\ell, v_r]$ with weight $w(v)$, where $v_\ell = 2 \cdot \max\{\textit{filled}(v) + 1, m(v)\}$ and $v_r = 2 \cdot M(v)$. Insert the weighted interval into I.

8 Solve the Max-Manhattan Skyline Problem on I and let f be the solution.

9 **for** $i = \alpha$ **downto** 2 **do**
  if $w(p_i) \leq f[2 \cdot n_i + 1]$ **then**
    Apply a *delete* operation on p_i in T_A.
  endif
endfor

10 Restore the leaf labels of L to the values that they had before Step 4.

11 **return** T_A

End `Fast_Filter_Clusters`

Before describing the correctness of `Fast_Filter_Clusters`, we introduce a lemma that can be used to speed up the detection of conflicting clusters. Let T be a phylogenetic tree such that $\Lambda(T)$ is a set of positive integers $\{1, 2, \ldots, n\}$. Moreover, let $\pi = \langle p_\alpha, p_{\alpha-1}, \ldots, p_1 \rangle$ denote a path in T, where p_α is the root of T and p_1 is a leaf. For each $i \in \{1, 2, \ldots, \alpha\}$, define $n_i := |\Lambda(T[p_i])|$. Note that $n_1 = 1$ and $n_\alpha = n$. The path π is called a *stratifying path* if $\Lambda(T[p_i]) = \{1, 2, \ldots, n_i\}$ for every $i \in \{1, 2, \ldots, \alpha\}$. Furthermore, for any $C \subseteq \Lambda(T)$, define *filled*(C) as the largest integer x such that $\{1, 2, \ldots, x\} \subseteq C$. In the following lemma, we determine whether a specified cluster C and the cluster associated with a specified node p_i on a stratifying path π are compatible:

▶ **Lemma 10.** *Let T be a phylogenetic tree with $\Lambda(T) = \{1, 2, \ldots, n\}$. Also, let $\pi = \langle p_\alpha, p_{\alpha-1}, \ldots, p_1 \rangle$ be a stratifying path in T. For any $C \subseteq \Lambda(T)$ and $i \in \{1, 2, \ldots, \alpha\}$, we have $C \not\sim \Lambda(T[p_i])$ iff $\max\{filled(C) + 1, \min(C)\} < n_i + 0.5 < \max(C)$ holds, where $\Lambda(T[p_i]) = \{1, 2, \ldots, n_i\}$.*

Proof. First suppose that $C \not\sim \Lambda(T[p_i])$. This means that there exist $x, y, z \in \Lambda(T)$ such that $x, y \in C$, $z \notin C$, $x, z \in \Lambda(T[p_i])$, and $y \notin \Lambda(T[p_i])$. Then:

- $x \in \Lambda(T[p_i])$ implies $x \leq n_i$. Since $x \in C$, we have $\min(C) < n_i + 0.5$.
- $y \notin \Lambda(T[p_i])$ gives $y > n_i$. Since $y \in C$ is an integer, we deduce that $n_i + 0.5 < \max(C)$.
- $z \in \Lambda(T[p_i])$ implies $z \leq n_i$. Furthermore, $z \notin C$ gives $z \geq filled(C) + 1$. Combining the two inequalities, we get $filled(C) + 1 < n_i + 0.5$.

On the other hand, suppose that $C \smile \Lambda(T[p_i])$. By the definition of cluster compatibility, at least one of the following three cases holds:

- $C \subseteq \Lambda(T[p_i])$: Then $x \leq n_i$ for all $x \in C$, i.e., $\max(C) \leq n_i$. Thus, the inequality $n_i + 0.5 < \max(C)$ is false.
- $\Lambda(T[p_i]) \subseteq C$: Then $filled(C) \geq n_i$, and hence $filled(C) + 1 < n_i + 0.5$ is false.
- $C \cap \Lambda(T[p_i]) = \emptyset$: Then $x > n_i$ for all $x \in C$, i.e., $\min(C) > n_i$. Thus, $\min(C) < n_i + 0.5$ is false. ◀

Figure 6 provides an illustration of Lemma 10.

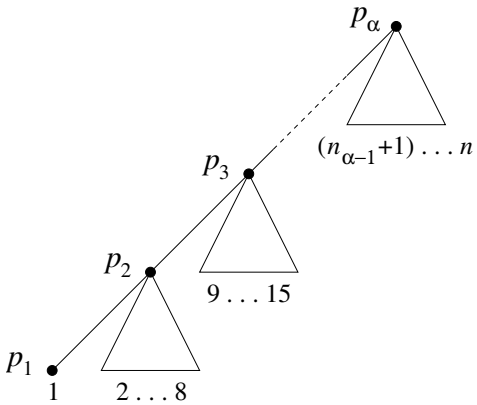

Figure 6 Illustration of Lemma 10. Let T be the tree with a stratifying path $\langle p_\alpha, p_{\alpha-1}, \ldots, p_1 \rangle$ shown above. Consider the node p_2 and a cluster $C = \{5, 6, 8, 9\}$. Since $\min(C) = 5$, $\max(C) = 9$, $filled(C) = 0$, and $n_2 = 8$, Lemma 10 implies $C \not\sim \Lambda(T[p_2])$. Next, consider p_2 and $C' = \{1, 2, \ldots, 12\}$. Since $\min(C') = 1$, $\max(C') = 12$, $filled(C') = 13$, and $n_2 = 8$, the inequality in Lemma 10 does not hold. Thus, $C' \smile \Lambda(T[p_2])$.

Now, we prove the correctness of the procedure `Fast_Filter_Clusters`:

▶ **Lemma 11.** *Given two weighted trees T_A and T_B with identical sets of n leaf labels, the procedure* `Fast_Filter_Clusters`(T_A, T_B) *works correctly.*

Proof. In Step 1, the procedure computes a centroid path $\pi = \langle p_\alpha, p_{\alpha-1}, \ldots, p_1 \rangle$, where p_α is the root of T_A, and the set $\sigma(\pi)$ of side trees of π. According to Equation (1), any cluster C in T_A that should be removed is in either π or one of its side trees. In the former case, C will be removed in Steps 3–10, and in the latter case, C will be removed during some recursive call in Step 2.

Step 2 handles the side trees in $\sigma(\pi)$ by recursively calling the procedure for each $\tau \in \sigma(\pi)$ and replacing τ in T_A by the obtained tree τ'. Before each recursive call, the procedure normalizes the weights of the nodes in τ and $T_B||\Lambda(\tau)$ to make them positive integers of size $O(|\Lambda(\tau)|)$. This is achieved by sorting the weights in non-decreasing order and then setting the weight of each node equal to its rank in this order, where equally ranked nodes get identical weights.

Steps 3–10 handle the centroid path π as follows. After doing a bottom-up traversal of T_A to compute $n_i := |\Lambda(T_A[p_i])|$ for all $p_i \in \pi$, the leaf labels are modified to make π a stratifying path in T_A. As a consequence, for any node p_i on the centroid path π, its associated cluster $\Lambda(T_A[p_i])$ makes an integer interval of the form $[1, n_i]$. According to Lemma 10, $\Lambda(T_A[p_i])$ and any cluster C associated with a non-spoiled node in T_B conflict with each other iff $\max\{\mathit{filled}(C) + 1, \min(C)\} < n_i + 0.5 < \max(C)$. Therefore, one can determine whether $\Lambda(T_A[p_i]) \not\succ C$ by constructing an interval whose left endpoint is $2 \cdot \max\{\mathit{filled}(C) + 1, \min(C)\}$ and whose right endpoint is $2 \cdot \max(C)$, and then checking if it contains the point $2 \cdot n_i + 1$. Otherwise, if C is associated with a spoiled node in T_B, the condition for a conflict becomes $\max\{\mathit{filled}(C)+1, \min(C)\} < n_i+0.5$, and the corresponding interval's right endpoint is set to $2 \cdot (n_\alpha + 1)$. Steps 5–9 determine conflicts simultaneously between all clusters associated with π and all clusters in T_B by solving an instance of the Max-Manhattan Skyline Problem. By the above, its solution f has the property that $f[2 \cdot n_i + 1]$ is the weight of the heaviest cluster in T_B that conflicts with any cluster of the form $\Lambda(T_A[p_i])$. If this number is greater than or equal to $w(p_i)$, then the procedure deletes p_i from T_A. ◀

Next, we show that `Fast_Filter_Clusters` runs in $O(n \log n)$ time:

▶ **Lemma 12.** *Given two weighted trees T_A and T_B with identical sets of n leaf labels, the procedure* `Fast_Filter_Clusters`(T_A, T_B) *runs in $O(n \log n)$ time.*

Proof. Step 1 can be completed in $O(n)$ time [8]. Step 2 uses $O(n)$ time to construct the $T_B||\Lambda(\tau)$-trees according to Lemma 2 and by applying radix sort to normalize the node weights. Step 2 also makes a recursive call for each τ. Next, bottom-up traversals of T_A and T_B are used to implement Steps 3–6, taking an additional $O(n)$ time. Creating the intervals that represent clusters in Step 7 takes $O(n)$ time. Also, solving the Max-Manhattan Skyline Problem in Step 8 takes $O(n)$ time, according to Lemma 3. This is because there are $O(n)$ intervals and their weights are positive integers of size $O(n)$. The necessary *delete* operations on π are carried out in top-down order, which means that the parent of any node in T_A is changed at most once and thereby, Step 9 takes $O(n)$ time in total. Finally, Step 10 restores the original leaf labels in $O(n)$ time.

The time complexity of `Fast_Filter_Clusters`(T_A, T_B) is thus $g(n) + \sum_{\tau \in \sigma(\pi)} h(\tau)$, where $g(n)$ is the execution time, excluding any recursive calls, and $h(\tau)$ is the running time of `Fast_Filter_Clusters`$(\tau, T_B||\Lambda(\tau))$ for any side tree τ of π. According to the discussion above, $g(n) = O(n)$. For any recursion level j, let σ_j denote the set of all side trees that are computed for all the centroid paths on this level. The total time taken on the recursion

level $j+1$ for the non-recursive parts is $\sum_{\tau\in\sigma_j} g(|\Lambda(\tau)|)$, and since the trees in σ_j are disjoint, $\sum_{\tau\in\sigma_j} g(|\Lambda(\tau)|) = g(n) = O(n)$. By Lemma 4, every τ satisfies $|\Lambda(\tau)| \leq n/2$, and hence there are $O(\log n)$ recursion levels. Therefore, the total running time is $O(n \log n)$. ◀

Combining Lemmas 11 and 12 provides the proof of Theorem 6:

▶ **Theorem 6.** *Given two weighted trees T_A and T_B with identical sets of n leaf labels, the procedure* `Fast_Filter_Clusters`(T_A, T_B) *filters out clusters as needed in $O(n \log n)$ time.*

6 Concluding Remarks

In this paper, we introduced an $O(kn \log n)$-time algorithm for computing the FDCT, leading to an asymptotically faster approach compared to the best previously known algorithm (Gawrychowski et al. [15]). The improved procedure `Fast_Compute_Weights`, presented as part of our new algorithm, can also be employed in algorithms for building the greedy consensus tree [15, 21, 32], replacing the slower versions of the procedure. Closing the gap between the upper bound of $O(kn \log n)$ and the lower bound of $\Omega(kn)$ for the running time of the fastest FDCT construction algorithm remains an important open problem.

We implemented our $O(kn \log n)$-time algorithm for constructing the FDCT. The source code can be found at `https://github.com/tswddd2/FDCT_new`. We achieved this implementation by adding approximately 2000 lines of C++ code to the source code of the $min\{O(kn^2), O(kn(k + \log^2 n))\}$-time algorithm [20] for the same problem, available at `https://github.com/Mesh89/FACT2` and also included in the FACT package [21], which was previously the fastest implementation. To implement the new $O(kn \log n)$ algorithm, we followed the descriptions in this article and used `dynamic_bitset` from the Boost libraries [1]. Preliminary experiments to evaluate the performance of our new algorithm indicate that it is faster in practice than the $O(kn^2)$-time and $O(kn(k + \log^2 n))$-time algorithms from [20]. The detailed results will be reported in the journal version of this paper.

References

1 The Boost C++ Libraries. `https://www.boost.org/`. Accessed on September 14, 2023.

2 Edward N Adams III. Consensus techniques and the comparison of taxonomic trees. *Systematic Biology*, 21(4):390–397, 1972.

3 Craig F Barrett, Jerrold I Davis, Jim Leebens-Mack, John G Conran, and Dennis W Stevenson. Plastid genomes and deep relationships among the commelinid monocot angiosperms. *Cladistics*, 29(1):65–87, 2013.

4 Michael A Bender and Martin Farach-Colton. The LCA problem revisited. In *Latin American Symposium on Theoretical Informatics*, pages 88–94. Springer, 2000.

5 Michael A Bender and Martın Farach-Colton. The level ancestor problem simplified. *Theoretical Computer Science*, 321(1):5–12, 2004.

6 Omer Berkman and Uzi Vishkin. Finding level-ancestors in trees. *Journal of Computer and System Sciences*, 48(2):214–230, 1994.

7 David Bryant. A classification of consensus methods for phylogenetics. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 61:163–184, 2003.

8 Richard Cole, Martin Farach-Colton, Ramesh Hariharan, Teresa Przytycka, and Mikkel Thorup. An $O(n \log n)$ algorithm for the maximum agreement subtree problem for binary trees. *SIAM Journal on Computing*, 30(5):1385–1404, 2000.

9 Maxime Crochemore, Costas S Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Extracting powers and periods in a word from its runs structure. *Theoretical Computer Science*, 521:29–41, 2014.

10 William HE Day. Optimal algorithms for comparing trees with labeled leaves. *Journal of Classification*, 2:7–28, 1985.

11 Jianrong Dong, David Fernández-Baca, FR McMorris, and Robert C Powers. Majority-rule (+) consensus trees. *Mathematical Biosciences*, 228(1):10–15, 2010.

12 Martin Farach and Mikkel Thorup. Fast comparison of evolutionary trees. *Information and Computation*, 123(1):29–37, 1995.

13 J Felsenstein. Phylip version 3.6. *Software package, Department of Genome Sciences, University of Washington, Seattle, USA*, 2005.

14 Nicolás García, Alan W Meerow, Douglas E Soltis, and Pamela S Soltis. Testing deep reticulate evolution in Amaryllidaceae tribe Hippeastreae (Asparagales) with ITS and chloroplast sequence data. *Systematic Botany*, 39(1):75–89, 2014.

15 Paweł Gawrychowski, Gad M Landau, Wing-Kin Sung, and Oren Weimann. A faster construction of greedy consensus trees. *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, 2018.

16 Pablo A Goloboff, James S Farris, Mari Källersjö, Bengt Oxelman, Martín J Ramírez, and Claudia A Szumik. Improvements to resampling measures of group support. *Cladistics*, 19(4):324–332, 2003.

17 Pablo A Goloboff, James S Farris, and Kevin C Nixon. TNT, a free program for phylogenetic analysis. *Cladistics*, 24(5):774–786, 2008.

18 Jon Gorrie. *Does culture evolve? Testing evolutionary theories of culture through a case study of El Khiam points from three sites in the Pre Pottery Neolithic A of the Southern Levant*. PhD thesis, Oxford Brookes University, 2021.

19 Gang Han, Luis M Chiappe, Shu-An Ji, Michael Habib, Alan H Turner, Anusuya Chinsamy, Xueling Liu, and Lizhuo Han. A new raptorial dinosaur with exceptionally long feathering provides insights into dromaeosaurid flight performance. *Nature Communications*, 5:4382, 2014.

20 Jesper Jansson, Ramesh Rajaby, Chuanqi Shen, and Wing-Kin Sung. Algorithms for the majority rule (+) consensus tree and the frequency difference consensus tree. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 15(1):15–26, 2018.

21 Jesper Jansson, Chuanqi Shen, and Wing-Kin Sung. Improved algorithms for constructing consensus trees. *Journal of the ACM*, 63(3):28, 2016.

22 Sung Kwon Kim, Jung-Sik Cho, and Soo-Cheol Kim. Path maximum query and path maximum sum query in a tree. *IEICE TRANSACTIONS on Information and Systems*, 92(2):166–171, 2009.

23 Max Cardoso Langer, Blair Wayne McPhee, Júlio César de Almeida Marsola, Lúcio Roberto-da Silva, and Sérgio Furtado Cabreira. Anatomy of the dinosaur *Pampadromaeus barberenai* (Saurischia–Sauropodomorpha) from the Late Triassic Santa Maria Formation of southern Brazil. *PLOS ONE*, 14(2):e0212543, 2019.

24 Charlotte Lindqvist, Jan De Laet, Robert R Haynes, Lone Aagesen, Brian R Keener, and Victor A Albert. Molecular phylogenetics of an aquatic plant lineage, Potamogetonaceae. *Cladistics*, 22(6):568–588, 2006.

25 Timothy Margush and Fred R McMorris. Consensus n-trees. *Bulletin of Mathematical Biology*, 43(2):239–244, 1981.

26 Carlos Molineri. A cladistic revision of *Tortopus* Needham & Murphy with description of the new genus *Tortopsis* (Ephemeroptera: Polymitarcyidae). *Zootaxa*, 2481:1–36, 2010.

27 Carlos Molineri and Frederico F Salles. Phylogeny and biogeography of the ephemeral *Campsurus* Eaton (Ephemeroptera, Polymitarcyidae). *Systematic Entomology*, 38(2):265–277, 2013.

28 Carlos Molineri, Frederico F Salles, and Janice G Peters. Phylogeny and biogeography of Asthenopodinae with a revision of *Asthenopus*, reinstatement of *Asthenopodes*, and the description of the new genera *Hubbardipes* and *Priasthenopus* (Ephemeroptera, Polymitarcyidae). *ZooKeys*, 478:45–128, 2015.

29 Martín O Pereyra, Boris L Blotto, Diego Baldo, Juan C Chaparro, Santiago R Ron, Agustín J Elias-Costa, Patricia P Iglesias, Pablo J Venegas, Maria Tereza C Thome, Jhon Jairo Ospina-Sarria, et al. Evolution in the genus *Rhinella*: a total evidence phylogenetic analysis of Neotropical true toads (Anura: Bufonidae). *Bulletin of the American Museum of Natural History*, 447(1):1–156, 2021.

30 Robert R Sokal and F James Rohlf. Taxonomic congruence in the Leptopodomorpha reexamined. *Systematic Zoology*, 30(3):309–325, 1981.

31 Mike Steel and Joel D Velasco. Axiomatic opportunities and obstacles for inferring a species tree from gene trees. *Systematic Biology*, 63(5):772–778, 2014.

32 Hongxun Wu. Near-optimal algorithm for constructing greedy consensus tree. In *47th International Colloquium on Automata, Languages, and Programming (ICALP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.