Journal of Computer and System Sciences 161 (2026) 103831

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

journal homepage: www.elsevier.com/locate/jcss

A faster algorithm for constructing the frequency difference consensus tree ☆

Jesper Jansson [a,*], Wing-Kin Sung [b,c], Seyed Ali Tabatabaee [d], Yutong Yang [c]

[a] Kyoto University, Kyoto, Japan
[b] The Chinese University of Hong Kong, Hong Kong, China
[c] Hong Kong Genome Institute, Hong Kong Science Park, Hong Kong, China
[d] University of British Columbia, Vancouver, Canada

ARTICLE INFO

Article history:
Received 13 August 2024
Received in revised form 29 May 2026
Accepted 9 June 2026
Available online 18 June 2026

Dataset link: https://github.com/tswddd2/FDCT_new

Keywords:
Phylogenetic tree
Frequency difference consensus tree
Tree algorithm
Centroid path decomposition
Max-Manhattan skyline problem

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 fifty years, the frequency difference consensus tree is one of the more finely resolved types that retains a large amount of information. This article 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. Furthermore, we demonstrate that the implementation of our algorithm is faster in practice than the prior implementations for the same problem.

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 [2,3]. 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, epidemiology, linguistics, anthropology, and geology. Many alternative consensus trees, each with its strengths and limitations, have been proposed [3].

The *frequency difference consensus tree* (FDCT) [4–6] has garnered interest among researchers in recent years [7–11]. 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 if 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

☆ A preliminary version of this article was presented at *the 41st International Symposium on Theoretical Aspects of Computer Science (STACS 2024)* (reference [1]).

* Corresponding author.
E-mail addresses: jj@i.kyoto-u.ac.jp (J. Jansson), kwksung@cuhk.edu.hk (W.-K. Sung), salitaba@cs.ubc.ca (S.A. Tabatabaee), tswddd@gmail.com (Y. Yang).

https://doi.org/10.1016/j.jcss.2026.103831

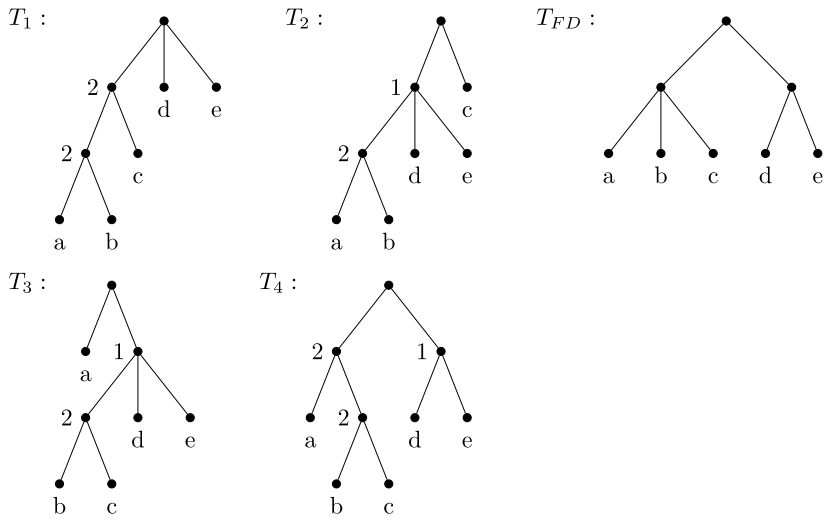


Fig. 1. Let $\mathcal{S} = \{T_1, T_2, T_3, T_4\}$ be a set of phylogenetic trees with the leaf label set $\{a, b, c, d, e\}$. T_{FD} is the FDCT of $\mathcal{S}$. In each T_i, the number beside each non-root internal node u indicates the weight $w(u)$. In this example, T_{FD} does not contain either of 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 trees in $\mathcal{S}$.

consensus trees, such as the strict consensus tree [12] and the majority rule consensus tree [13], is that it captures more of the shared branching information.

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 article, the $O(kn \log^2 n)$-time algorithm by Gawrychowski et al. [8] was the asymptotically fastest algorithm for constructing the FDCT. Here, we present an $O(kn \log n)$-time algorithm for constructing it, thus reducing the gap between the known upper and lower bounds on the running time of the fastest possible algorithm. In addition, we show that the implementation of our new algorithm outperforms the previously fastest implementation, developed by Jansson et al. [9].

The overarching structure of our new algorithm follows the framework proposed by Jansson et al. [9] for computing the FDCT. By improving the methods for solving two subproblems in [9], 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* [3,14] involve the same weighting step [8,15,16] and may derive advantages from our 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 [17] to identify the clusters that should be removed at each recursive stage of the algorithm (see Section 5 below for a 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 article.

Let T be a tree. The set of nodes of T (including its leaves) 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 $\bigcup_{u \in V(T)} \{\Lambda(T[u])\}$. For example, the cluster collection of the tree T_1 in Fig. 1 is $\{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{a, b\}, \{a, b, c\}, \{a, b, c, d, e\}\}$. A cluster $C \subseteq \Lambda(T)$ *occurs* in T if and only if $C \in \mathcal{C}(T)$; in this case, we also say that T *contains* C.

Two clusters $C_1, C_2 \subseteq \Lambda(T)$ are said to be *compatible*, written as $C_1 \sim C_2$, if and only if $C_1 \subseteq C_2$, $C_2 \subseteq C_1$, or $C_1 \cap C_2 = \emptyset$. The notion of compatibility carries over to trees in a straightforward way. A cluster $C \subseteq \Lambda(T)$ is *compatible* with T, denoted as $C \sim T$, if and only if for every $C' \in \mathcal{C}(T)$, we have $C \sim C'$. Further, two trees T_1 and T_2 with identical leaf label sets are *compatible*, denoted as $T_1 \sim T_2$, if and only if for every $C \in \mathcal{C}(T_1)$, $C \sim T_2$ holds, i.e., if and only if every cluster in T_1 is compatible with T_2. This also means that every cluster in T_2 is compatible with T_1. Clusters and/or trees are called *incompatible*, expressed using the symbol $\not\sim$, if they are not compatible. For example, in Fig. 1, $\{a, b, d\} \sim T_2$. In contrast, $\{a, b, c\} \not\sim \{a, b, d, e\}$, so $\{a, b, c\} \not\sim T_2$ and $T_1 \not\sim T_2$. Two incompatible clusters are also said to *conflict* with each other.

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 $\mathcal{C}(T_{FD}) = \{C : C \subseteq L \text{ and } w(C) > max(\{w(C') : C' \subseteq L \text{ and } C \not\succ C'\})\}$. Thus, T_{FD} contains every cluster that occurs more frequently than any cluster incompatible with it; we shall refer to such clusters as *frequency difference clusters*. By Proposition 3 in [10], T_{FD} always exists and is unique for a given $\mathcal{S}$. Fig. 1 illustrates the FDCT for a collection of four phylogenetic 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. Many of these consensus trees are explained in depth in [3]. Here, we briefly mention two types that are widely used in practice: the strict consensus tree [12] and the majority-rule consensus tree [13]. The strict consensus tree contains all clusters that occur in every one of the input trees and can be computed in optimal $O(kn)$ running time [18]. However, potentially important information might be excluded from the strict consensus tree; if a cluster is missing in just one of the input trees, it will be discarded. The majority-rule consensus tree is a relaxation of the strict consensus tree that contains all clusters occurring in more than half of the input trees. The majority-rule consensus tree can also be computed in optimal $O(kn)$ time [15].

The frequency difference consensus tree (FDCT) was introduced by Goloboff et al. [4,5] as an even more informative alternative that contains not only the clusters that occur in the majority of trees but also the other frequency difference clusters. Since then, it has been utilized in many scientific studies such as [19–27]. Dong et al. [7] provided a comparison of the FDCT to other types of consensus trees and established how their cluster collections are related. Steel and Velasco [10] 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 consensus trees, the FDCT easily generalizes to a viable supertree definition. Further motivation for using the FDCT in biology was given by Velasco in [11], where it was referred to as the "plurality consensus tree".

As for algorithms to compute the FDCT, an implementation can be found in the software package TNT [6], but the actual algorithm employed by TNT and its time complexity remain undisclosed. Jansson et al. [9] presented a deterministic $\min\{O(kn^2), O(kn(k + \log^2 n))\}$-time algorithm for the FDCT. This algorithm was implemented in the open-source FACT package [15] and shown experimentally in [9] to be significantly faster than TNT's implementation. Subsequently, Gawrychowski et al. [8] developed a faster method for the weighting step in [9], yielding an improved running time complexity of $O(kn \log^2 n)$ for constructing the FDCT.

1.3. Organization of the article

Section 2 below reviews some results from the literature that will be used later. Section 3 explains the framework of the $O(kn \log n)$-time algorithm for computing the FDCT. Sections 4 and 5 present algorithms for solving two subproblems of the FDCT construction efficiently. Section 6 describes an implementation and experimental evaluation of the new algorithm. Finally, Section 7 gives some concluding remarks.

2. Preliminaries

2.1. The delete operation

The *delete* operation on a non-root internal node u in a tree T modifies T by first removing u and all edges incident to u from T, and then attaching each node that used to be a child of u to the node that used to be the parent of u. As a result, the cluster $\Lambda(T[u])$ disappears from T's cluster collection $\mathcal{C}(T)$ while all other clusters in $\mathcal{C}(T)$ are left intact. If c is the number of children of u then the *delete* operation on u takes $O(c)$ time.

2.2. Restriction of trees

Let T be a tree. For any non-empty set of nodes $U \subseteq V(T)$, their lowest common ancestor in T is denoted by $lca^T(U)$. For any cluster $C \subseteq \Lambda(T)$, define $T|C$ (referred to as the *restriction of T to C*) to be 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. Fig. 2 gives an example of restricting a tree to different clusters.

We shall use the following result from Section 8 in [28] concerning the complexity of constructing restrictions of trees:

Lemma 1. *[28] Let T be a phylogenetic tree with n leaves. After $O(n)$ time preprocessing, $T|C$ for any nonempty $C \subseteq \Lambda(T)$ can be constructed in $O(|C|)$ time.*

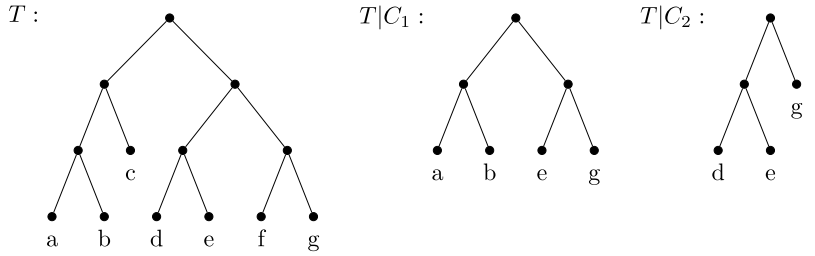


Fig. 2. Illustration of restricting a tree T to different clusters C_1 and C_2, where $C_1 = \{a, b, e, g\}$ and $C_2 = \{d, e, g\}$.

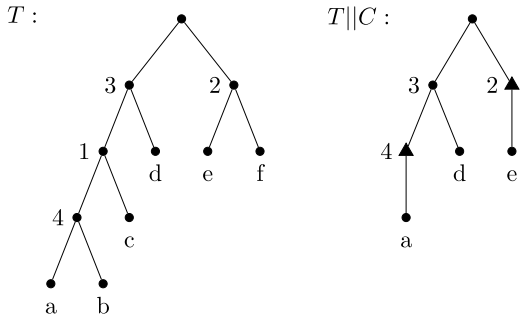


Fig. 3. Illustration of the expanded restriction of a tree T to a cluster C, where $C = \{a, d, e\}$. Internal nodes are labeled with their weights. In this example, all internal nodes in $T||C$ will be spoiled even if T did not contain any spoiled nodes, and the two internal nodes drawn as triangles are path nodes.

2.3. Expanded restriction of trees

When computing $T|C$ for a tree T and a subset C of its leaf labels as defined in Section 2.2, some nodes in T will be removed. After doing so, it is not possible to directly see which nodes in $T|C$ have lost one or more leaf descendants. Moreover, in $T|C$, information about the weights of the clusters associated with the removed nodes has been lost. To address these two issues, we follow [9] and extend the concept of restrictions of trees to allow nodes to be marked as *spoiled* and also insert special nodes called *path nodes* into the trees. If a node is spoiled, it means that its set of leaf descendants no longer contains all the elements that it had in the original T. The purpose of the path nodes is to compactly represent the weights of the clusters that were lost when building $T|C$ and that may conflict with clusters from other trees that are subsets of C. To be precise, for any tree T either from the input $\mathcal{S}$ (in which case no nodes are already spoiled) or previously obtained as the expanded restriction of some tree (in which case some nodes may already be spoiled) and any $C \subseteq \Lambda(T)$, we define 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 $|\Lambda(T'[u])| < |\Lambda(T[u])|$ 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. Fig. 3 shows an example of the expanded restriction of trees.

Lemma 2. *Let T be a weighted phylogenetic tree with n leaves. After $O(n \log n)$ time preprocessing, $T||C$ for any nonempty $C \subseteq \Lambda(T)$ can be constructed in $O(|C|)$ time.*

Proof. First, apply the following preprocessing to T. Conduct an $O(n)$-time bottom-up traversal of T to precompute and store the depth of every node in T as well as the number of leaf descendants it has. Also do the $O(n)$-time preprocessing of T from Lemma 1, an $O(n)$-time preprocessing of T after which level-ancestor queries (i.e., queries that ask for the ancestor in T at a specified depth of a specified node) are supported in $O(1)$ time [29,30], and an $O(n \log n)$-time preprocessing of T that lets the maximum weight among all nodes on the path between any two given nodes a and b in T be reported in $O(1)$ time [31, Theorem 2]. In total, the preprocessing takes $O(n \log n)$ time.

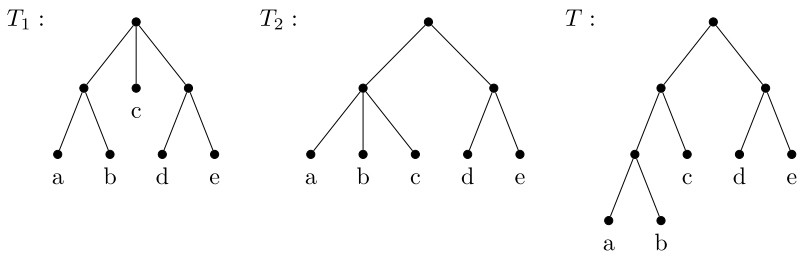


Fig. 4. Illustration of merging two trees with $T = \texttt{Merge\_Trees}(T_1, T_2)$.

Next, for any nonempty $C \subseteq \Lambda(T)$, the tree $T||C$ is constructed according to steps 1.–4. in the definition above. In the first step, $T' = T|C$ is constructed in $O(|C|)$ time according to Lemma 1. Identifying and marking the spoiled nodes in the second step also takes $O(|C|)$ time by using a bottom-up traversal of T' to count the number of leaf descendants of each node u and comparing it to the (precomputed) number of leaf descendants of u in the original T. Finally, to compute the weight of each path node z in T' in the third step, let u and v be its parent and child in T', respectively, and set the weight of z to the maximum weight of all nodes along the path in T between u and v, excluding u and v. The latter can be obtained in $O(1)$ time by letting a be the level-ancestor of v in T at depth $1 + x$, where x is the depth of u (in other words, a is the second node on the path in T from u to v), letting b be the parent of v in T, and taking the maximum weight of all nodes on the path between a and b. There are $O(|C|)$ path nodes in T', so in total, this uses $O(|C|)$ time. □

To extend the definition of compatibility given in Section 1.1 to spoiled nodes, suppose that $C_1, C_2 \subseteq \Lambda(T)$ and that u is a spoiled node in $T||C_1$. Then $C_2 \smile u$ if and only if 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 for spoiled nodes are asymmetric.

2.4. Merging trees

Given two compatible trees T_1 and T_2 with identical leaf label sets, a procedure named $\texttt{Merge\_Trees}(T_1, T_2)$ from Section 2.4 in [15] returns a tree containing all clusters that occur in T_1 or T_2. See Fig. 4 for an example.

We summarize the properties of the procedure `Merge_Trees` in the following lemma.

Lemma 3. *[15] Given two trees T_1 and T_2 such that $\Lambda(T_1) = \Lambda(T_2) = L$ and $T_1 \smile T_2$, $\texttt{Merge\_Trees}(T_1, T_2)$ returns a tree T with $\Lambda(T) = L$ and $\mathcal{C}(T) = \mathcal{C}(T_1) \cup \mathcal{C}(T_2)$ in $O(|L|)$ time.*

2.5. 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. [17, Section 5.1] gave an $O(n)$-time algorithm for the Min-Manhattan Skyline Problem, defined in the same way as the Max-Manhattan Skyline Problem above, 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 [17] to sort the intervals in non-increasing order instead, we immediately have:

Lemma 4. *The Max-Manhattan Skyline Problem can be solved in $O(n)$ time.*

2.6. Centroid paths and side trees

A *centroid path* [28] of a tree T is a path in T 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 5. *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 5 that the number of recursion levels needed to complete such a decomposition is $O(\log |\Lambda(T)|)$.

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 Apply the preprocessing in Lemmas 1 and 2 to each tree in $\mathcal{S}$.
2 `Fast_Compute_Weights`($\mathcal{S}$)

/* Main algorithm */
3 $T := T_1$
4 **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
5 **for** $j := 1$ **to** k **do**
  $T :=$ `Fast_Filter_Clusters`(T, T_j)
6 **return** T
End `Fast_Frequency_Difference`

Fig. 5. The algorithm `Fast_Frequency_Difference` for constructing the FDCT, whose overall structure is the same as the algorithm `Frequency_Difference` in [9].

3. Algorithm `Fast_Frequency_Difference`

The pseudocode of our new algorithm, named `Fast_Frequency_Difference`, is shown in Fig. 5. Its overall structure follows the framework developed in [9] for constructing the FDCT, which we review next.

Suppose that the input is $\mathcal{S} = \{T_1, T_2, \ldots, T_k\}$. The basic idea of the algorithm in [9] 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. 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 2.4, 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 algorithm from [9] by replacing the preprocessing step for computing the weights of all clusters occurring in $\mathcal{S}$ (Step 2) and the procedure `Filter_Clusters` (used in Steps 4 and 5) by more efficient solutions, referred to as `Fast_Compute_Weights` and `Fast_Filter_Clusters` below.

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

Theorem 1. *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.*

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

Theorem 2. *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) *computes a tree T with $\mathcal{C}(T) = \{\Lambda(T_A[u]) : u \in V(T_A)$ and $w(u) > w(x)$ for every $x \in V(T_B)$ with $\Lambda(T_A[u]) \not\succ \Lambda(T_B[x])\}$ in $O(n \log n)$ time.*

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

Theorem 3. *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. The improved procedures function as intended by Theorems 1 and 2, so the correctness of the algorithm `Fast_Frequency_Difference` follows from that of `Frequency_Difference` proved by [9].

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`

Fig. 6. The procedure `Fast_Compute_Weights`.

Now, we analyze the time complexity. The preprocessing in Step 1 uses $O(n \log n)$ time for each tree in $\mathcal{S}$ by Lemmas 1 and 2, contributing a total of $O(kn \log n)$ time. Step 2 makes a call to the procedure `Fast_Compute_Weights`, which takes $O(kn \log n)$ time according to Theorem 1. Steps 4 and 5 make $O(k)$ calls to the procedures `Fast_Filter_Clusters` and `Merge_Trees`. By Theorem 2, each call to the procedure `Fast_Filter_Clusters` takes $O(n \log n)$ time, and by Lemma 3, each call to the procedure `Merge_Trees` takes $O(n)$ time. Hence, Steps 4 and 5 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 procedure `Fast_Compute_Weights` into two phases called labeling and counting, similar to the strategy used in [8]. 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 $\mathcal{S}$, $id(u) = id(u')$ if and only if 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. Fig. 6 presents the pseudocode for the procedure `Fast_Compute_Weights`.

The pseudocode for `Fast_Label_Trees`($\{T_1, \ldots, T_k\}$) which carries out the labeling phase is given in Fig. 7. It uses a divide-and-conquer approach. First, it partitions the leaf label set L into two parts L' and L'', where the difference between $|L'|$ and $|L''|$ is at most one. It then calls `Fast_Label_Trees`($\{T_1|L', \ldots, T_k|L'\}$) and `Fast_Label_Trees`($\{T_1|L'', \ldots, T_k|L''\}$) recursively 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 each 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 $C \subseteq L$ is the node in the tree $T_i|C$ that corresponds to u. Formally, if $\Lambda(T_i[u]) \cap C \neq \emptyset$ then φ_u^C is defined as the $v \in V(T_i|C)$ for which $\Lambda((T_i|C)[v]) = \Lambda(T_i[u]) \cap C$, i.e., as the node in $T_i|C$ whose leaf descendants are exactly the same as those leaf descendants of node u that also belong to C; on the other hand, if $\Lambda(T_i[u]) \cap C = \emptyset$ then no such node exists in $T_i|C$, and in this case, φ_u^C is defined to be an imaginary node Φ that satisfies $id(\Phi) = 0$. 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 each tree $T_i \in \mathcal{S}$, $id(u)$ is set to the rank of the pair $(id(\varphi_u^{L'}), id(\varphi_u^{L''}))$.

See Fig. 8 for an illustration of how `Fast_Label_Trees` operates.

The next two lemmas prove the correctness of the procedure `Fast_Label_Trees` and analyze its time complexity.

Lemma 6. *Given a set $\mathcal{S} = \{T_1, T_2, \ldots, T_k\}$ of trees such that $\Lambda(T_1) = \Lambda(T_2) = \ldots = \Lambda(T_k) = L$ for some set of L of leaf labels, the following two statements hold after running* `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 $id(u) = id(v)$ if and only if $\Lambda(T_i[u]) = \Lambda(T_j[v])$.*

Proof. We start by showing that the first statement holds. In any $T_i \in \mathcal{S}$, every internal node has at least two children by the definition of a phylogenetic tree, so $|V(T_i)| < 2|L|$. Thus, the total number of nodes in all trees is less than $2k|L|$, and so 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 $id(u) = id(v)$ if and only if $\Lambda(T_i[u]) = \Lambda(T_j[v])$. Now, we need to prove the statement for $|L| = k + 1$.

If $\Lambda(T_i[u]) = \Lambda(T_j[v])$ then we obtain $\Lambda((T_i|L')[\varphi_u^{L'}]) = \Lambda((T_j|L')[\varphi_v^{L'}])$ 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, $(id(\varphi_u^{L'}), id(\varphi_u^{L''})) = (id(\varphi_v^{L'}), id(\varphi_v^{L''}))$ and thereby, $id(u) = id(v)$.

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 if and only if 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 each tree in $\mathcal{S}$, set $id(u) = 1$.
  return
endif
2 Partition L into L' and L'' so that the difference between $|L'|$ and $|L''|$ is at most one.
3 For all $i \in [1 \ldots k]$, construct $T_i' = T_i|L'$ 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 each 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 each 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`

Fig. 7. The procedure `Fast_Label_Trees`.

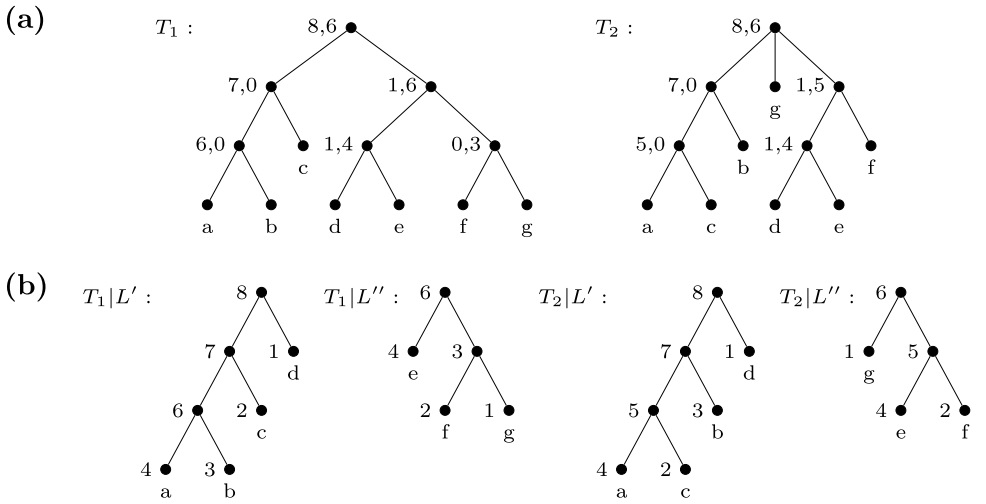


Fig. 8. Illustration of how labels are computed by `Fast_Label_Trees`. Part (a) shows two trees T_1 and T_2, with the pair assigned to each internal node in Step 6 written beside it. (For clarity, the pairs assigned to the leaves are not shown.) The pairs were obtained by combining the recursively computed node labels for the trees $T_1|L'$, $T_1|L''$, $T_2|L'$, and $T_2|L''$ with $L' = \{a, b, c, d\}$ and $L'' = \{e, f, g\}$, shown in part (b). Note that the labels in part (a) are the ones before the radix sort in Step 7, while the labels shown in part (b) are the labels that result after Step 8 one recursion level lower.

Conversely, if $id(u) = id(v)$ then $(id(\varphi_u^{L'}), id(\varphi_u^{L''})) = (id(\varphi_v^{L'}), id(\varphi_v^{L''}))$, which gives $id(\varphi_u^{L'}) = id(\varphi_v^{L'})$ and $id(\varphi_u^{L''}) = id(\varphi_v^{L''})$. Then, 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 $\Lambda(T_i[u]) = \Lambda(T_j[v])$. □

Lemma 7. *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 $T(n)$ stand for the time complexity of `Fast_Label_Trees` for an input set of k trees sharing a leaf label set of size n. Due to Step 1, $T(1) = O(k)$.

For $n \geq 2$, we sum up the contributions of Steps 2–8. Step 2 takes $O(n)$ time. According to Lemma 1, after the preprocessing by the algorithm `Fast_Frequency_Difference` in Section 3, T_i' and T_i'' can be constructed in $O(n)$ time for each $T_i \in \mathcal{S}$. For this reason, Step 3 takes $O(kn)$ time. Steps 4 and 5 take $T(n/2) + T(n/2) = 2T(n/2)$ time by definition. In Step 6, computing $id(\varphi_u^{L'})$ for each node u in each tree T_i can be done in a total of $O(kn)$ time using bottom-up traversals in the following way. For each $T_i \in \mathcal{S}$, first do a bottom-up traversal of T_i to count how many leaf descendants belonging to

L' that each node has. Next, for every leaf ℓ in T_i whose leaf label is in L', let $\varphi_\ell^{L'}$ be the leaf in T_i' that has the same leaf label as ℓ, and initialize $\varphi_u^{L'} = \Phi$ for all other nodes u in T_i. After that, do a bottom-up traversal of T_i that for each node u in T_i retrieves the integer label $id(\varphi_u^{L'})$ that was computed in Step 4, and additionally, if $\varphi_u^{L'} \neq \Phi$ and $\varphi_v^{L'} = \Phi$ for the parent v of u in T_i, updates $\varphi_v^{L'}$ to either $\varphi_u^{L'}$ or the parent of $\varphi_u^{L'}$ in T_i' depending on if u and v have equal or different numbers of leaf descendants from L'. Computing all integer labels $id(\varphi_u^{L''})$ in the same way also takes $O(kn)$ time. (Alternatively, one can augment the algorithm used in Lemma 1 for constructing $T|C$ so that it also saves a pointer from each node u in T to the node φ_u^C in $T|C$ without increasing its asymptotic time complexity, so that `Fast_Label_Trees` can obtain $id(\varphi_u^{L'})$ and $id(\varphi_u^{L''})$ directly in $O(1)$ time for each node u.) The number of obtained pairs of the form $(id(\varphi_u^{L'}), id(\varphi_u^{L''}))$ is $O(kn)$, and furthermore, we know from Lemma 6 and the fact that $id(\Phi) = 0$ that all values in the pairs are in $\{0, 1, \ldots, O(kn)\}$, so sorting the pairs using radix sort and assigning labels to the nodes in Steps 7 and 8 takes $O(kn)$ time.

This yields the recurrence $T(n) = 2T(n/2) + O(kn)$, which has the solution $T(n) = O(kn \log n)$. □

Now, we can prove Theorem 1, regarding the correctness and time complexity of the procedure `Fast_Compute_Weights`:

Theorem 1. *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 6. 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.

Finally, we analyze the time complexity. As shown in Lemma 7, 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. 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`

The pseudocode of our new procedure `Fast_Filter_Clusters`, with an improved running time of $O(n \log n)$, is summarized in Fig. 9. On a high level, it follows the same approach as the $O(n \log^2 n)$-time procedure `Filter_Clusters` in [9]. 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 $\Lambda(T_A[u]) \not\succ \Lambda(T_B[x])\}$, i.e., to delete every cluster in T_A that conflicts with at least one cluster in T_B that has a greater or equal weight. 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 difference between `Filter_Clusters` from [9] and `Fast_Filter_Clusters` presented here is how they handle the clusters $\bigcup_{p_i \in \pi} \{\Lambda(T_A[p_i])\}$. The former detects conflicts between such clusters and the clusters in T_B by traversing π in the upward direction while using a priority queue to keep track of all nodes from T_B whose associated clusters are incompatible with the current node on π. At each step, the priority queue is updated and the heaviest incompatible cluster is retrieved from it, leading to a total of $O(n \log n)$ 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 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 incompatible 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 $O(n \log n)$.

We now introduce the key lemma that speeds up the detection of incompatible clusters by allowing clusters to be interpreted as intervals. Let T be a 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 $\min(C)$ as the smallest integer x such that $x \in C$, $\max(C)$ as the largest integer x such that $x \in C$, and *filled*(C) as the largest integer x such that $\{1, 2, \ldots, x\} \subseteq C$. Using this notation, we

Algorithm `Fast_Filter_Clusters`

Input: Two trees T_A and T_B with $\Lambda(T_A) = \Lambda(T_B) = L$, where every $u \in V(T_A) \cup V(T_B)$ has a positive integer weight $w(u)$ and where 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 $\{1, 2, \ldots, n_\alpha\}$ so that π becomes 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), defined as the largest integer x such that $\{1, 2, \ldots, x\} \subseteq \Lambda(T_B[v])$.
7 Create a set I of weighted intervals over $\{1, 2, \ldots, n_\alpha\}$ that represent the clusters of T_B as follows:
  For each $v \in V(T_B)$, make an interval $[v_\ell, v_r]$ with weight $w(v)$, where $v_\ell = \max\{$*filled*$(v) + 1, m(v)\}$ and $v_r = M(v) - 1$.
  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 $w(p_i) \leq f[n_i]$ **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`

Fig. 9. The procedure `Fast_Filter_Clusters`.

can express whether a specified cluster C and the cluster associated with a specified node p_i on a stratifying path π are compatible in terms of some easily checkable conditions:

Lemma 8. *Let T be a phylogenetic tree with $\Lambda(T) = \{1, 2, \ldots, n\}$ and 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 $C \not\sim \Lambda(T[p_i])$ if and only if $\max\{$filled$(C) + 1, \min(C)\} \leq n_i \leq \max(C) - 1$ 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 $\min(C) \leq n_i$.
- $y \notin \Lambda(T[p_i])$ gives $y > n_i$. Since $y \in C$ is an integer, we deduce that $n_i \leq \max(C) - 1$.
- $z \in \Lambda(T[p_i])$ implies $z \leq n_i$. Furthermore, $z \notin C$ gives $z \geq$ *filled*$(C) + 1$. Combining the two inequalities, we get *filled*$(C) + 1 \leq n_i$.

Conversely, suppose that $C \backsim \Lambda(T[p_i])$. 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 $n_i \leq \max(C) - 1$ is false.
- $\Lambda(T[p_i]) \subseteq C$: Then *filled*$(C) \geq n_i$, and hence *filled*$(C) + 1 \leq n_i$ 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, $\min(C) \leq n_i$ is false. □

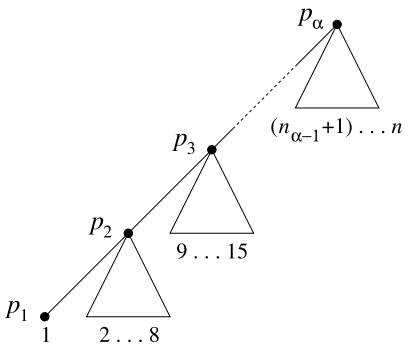


Fig. 10. Illustration of Lemma 8. Let T be the tree with a stratifying path $\langle p_\alpha, p_{\alpha-1}, \ldots, p_1\rangle$ shown above. First, consider the node p_2 and a cluster $C = \{5, 6, 7, 9\}$. Since $\min(C) = 5$, $\max(C) = 9$, $\mathit{filled}(C) = 0$, and $n_2 = 8$, Lemma 8 gives $C \not\sim \Lambda(T[p_2])$. Next, consider p_2 and $C' = \{1, 2, 3, 4, 5, 6, 7, 9\}$. Since $\min(C') = 1$, $\max(C') = 9$, $\mathit{filled}(C') = 7$, and $n_2 = 8$, Lemma 8 again yields $C \not\sim \Lambda(T[p_2])$. Finally, consider p_2 and $C'' = \{1, 2, \ldots, 9\}$. Since $\min(C'') = 1$, $\max(C'') = 9$, $\mathit{filled}(C'') = 9$, and $n_2 = 8$, the inequality in Lemma 8 does not hold, which implies $C'' \sim \Lambda(T[p_2])$. Also note that if C, C', and C'' are clusters of T_B associated with non-spoiled nodes then they will define the intervals $[5, 8]$, $[8, 8]$, and $[10, 8] = \emptyset$, respectively, in Step 7 of `Fast_Filter_Clusters`, and that the point n_2 is contained in the first two of these three intervals.

Fig. 10 illustrates Lemma 8 and how it is applied in `Fast_Filter_Clusters` to define intervals that represent conflicts between clusters associated with the centroid path of T_A and clusters in T_B. We formalize the technique as a corollary to Lemma 8:

Corollary 1. *Let T_A and T_B be two phylogenetic trees with $\Lambda(T_A) = \Lambda(T_B) = L$, where some nodes in T_B may be spoiled, and let $\pi = \langle p_\alpha, p_{\alpha-1}, \ldots, p_1\rangle$ be a stratifying path in T_A such that $\Lambda(T[p_i]) = \{1, 2, \ldots, n_i\}$ for every $i \in \{1, 2, \ldots, \alpha\}$. Consider any $v \in V(T_B)$. Define $v_\ell = \max\{\mathit{filled}(\Lambda(T_B[v])) + 1,\ \min(\Lambda(T_B[v]))\}$. Also, if v is not spoiled then define $v_r = \max(\Lambda(T_B[v])) - 1$; otherwise (i.e., if v is spoiled), define $v_r = n_\alpha$. Then for each $i \in \{1, 2, \ldots, \alpha\}$, $\Lambda(T_A[p_i]) \not\sim \Lambda(T_B[v])$ if and only if the point n_i is contained in the interval $[v_\ell, v_r]$.*

Proof. If v is not spoiled then according to Lemma 8, $\Lambda(T_A[p_i]) \not\sim \Lambda(T_B[v])$ if and only if $\max\{\mathit{filled}(C) + 1, \min(C)\} \le n_i \le \max(C) - 1$, where $C = \Lambda(T_B[v])$. By the definition of v_ℓ and v_r, this inequality is equivalent to $v_\ell \le n_i \le v_r$.

On the other hand, if v is spoiled then $\Lambda(T_A[p_i]) \not\sim \Lambda(T_B[v])$ is also true in case $\Lambda(T_B[v]) \subsetneq \Lambda(T_A[p_i])$. Thus, $\Lambda(T_A[p_i]) \sim C$, where $C = \Lambda(T_B[v])$, if and only if: (i) $\Lambda(T_A[p_i]) \subseteq C$; or (ii) $C \cap \Lambda(T[p_i]) = \emptyset$. As in the proof of Lemma 8, (i) is satisfied if and only if $\mathit{filled}(C) \ge n_i$, while (ii) is satisfied if and only if $\min(C) > n_i$. This gives $\Lambda(T_A[p_i]) \sim C$ if and only if $\mathit{filled}(C) + 1 > n_i$ or $\min(C) > n_i$ holds. Stated differently, $\Lambda(T_A[p_i]) \not\sim C$ if and only if $\mathit{filled}(C) + 1 \le n_i$ and $\min(C) \le n_i$. As it no longer matters whether $n_i \le \max(C) - 1$ here, the condition for a conflict simplifies to $\max\{\mathit{filled}(C) + 1, \min(C)\} \le n_i$. Finally, since $n_i \le n_\alpha$ and $v_r = n_\alpha$, we obtain $\Lambda(T_A[p_i]) \not\sim \Lambda(T_B[v])$ if and only if $v_\ell \le n_i \le v_r$. □

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

Lemma 9. *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 in the range $\{1, 2, \ldots, O(|\Lambda(\tau)|)\}$. 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 $p_i \in \pi$ in Step 3, the leaf labels are modified to make π a stratifying path in T_A in Step 4. As a consequence, for any specified node p_i on the centroid path π, its associated cluster is incompatible with all clusters in T_B whose intervals defined according to Corollary 1 contain the point n_i. In particular, the heaviest cluster that is incompatible with $\Lambda(T_A[p_i])$ corresponds to the heaviest interval that n_i belongs to. Steps 5–8 find the weights of the heaviest incompatible clusters from T_B for all clusters associated with π simultaneously by creating and solving an instance of the Max-Manhattan Skyline Problem. Its solution f has the property that for any $p_i \in \pi$, the value of $f[n_i]$ is the weight of the heaviest cluster in T_B that is incompatible with the cluster $\Lambda(T_A[p_i])$. If $f[n_i]$ is greater than or equal to $w(p_i)$, then the procedure deletes p_i from T_A in Step 9.

We conclude that the resulting T_A contains no cluster that conflicts with some cluster in T_B that has an equal or greater weight. □

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

Lemma 10. *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 [28]. To construct any $T_B||\Lambda(\tau)$-tree takes $O(|\Lambda(\tau)|)$ time according to Lemma 2, as the required preprocessing was already done by the algorithm `Fast_Frequency_Difference` in Section 3. The side trees' leaf label sets are disjoint, so Step 2 uses $\sum_{\tau \in \sigma(\pi)} O(|\Lambda(\tau)|) = O(n)$ time to construct the $T_B||\Lambda(\tau)$-trees. Then, applying radix sort to normalize the node weights takes $O(n)$ time. Step 2 also makes a recursive call for each side tree τ. Next, bottom-up traversals of T_A and T_B are used to implement Steps 3–6, taking an additional $O(n)$ time. To create the intervals that represent the clusters of T_B in Step 7 takes $O(n)$ time, and to solve the Max-Manhattan Skyline Problem in Step 8 also takes $O(n)$ time according to Lemma 4 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, so Step 9 takes $O(n)$ time in total. Finally, Step 10 restores the original leaf labels in $O(n)$ time.

In summary, the time complexity of `Fast_Filter_Clusters`(T_A, T_B) is $g(n) + \sum_{\tau \in \sigma(\pi)} h(\tau)$, 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 5, every τ satisfies $|\Lambda(\tau)| \leq n/2$, and hence there are $O(\log n)$ recursion levels. This shows that the total running time is $O(n \log n)$. □

Combining Lemmas 9 and 10, we get Theorem 2:

Theorem 2. *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) *computes a tree T with $\mathcal{C}(T) = \{\Lambda(T_A[u]) : u \in V(T_A)$ and $w(u) > w(x)$ for every $x \in V(T_B)$ with $\Lambda(T_A[u]) \not\succ \Lambda(T_B[x])\}$ in $O(n \log n)$ time.*

6. Implementation

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. It was actualized by adding roughly 2000 lines of C++ code to the source code of the $\min\{O(kn^2), O(kn(k + \log^2 n))\}$-time algorithm [9] for the same problem, available at https://github.com/Mesh89/FACT2 and also included in the FACT package [15], which was previously the fastest implementation. To implement the new $O(kn \log n)$-time algorithm, we followed the descriptions in this article and also used `dynamic_bitset` from the Boost libraries [32].

We evaluated the performance of the implementation of our $O(kn \log n)$-time algorithm (which we refer to as *New*) by measuring its running time for randomly generated inputs of different sizes and comparing it with the running times of the implementations of the $O(kn^2)$-time and $O(kn(k + \log^2 n))$-time algorithms [9] (which we call *Old_kn2* and *Old_k2n*, respectively) for the same inputs. While both *Old_kn2* and *Old_k2n* run the procedure `Filter_Clusters` in $O(n \log^2 n)$ time, they are different in the way they compute the cluster weights. The algorithm used by *Old_kn2* for computing cluster the weights runs in $O(kn^2)$ time, whereas *Old_k2n* computes the cluster weights in $O(k^2 n)$ time.

We conducted all the experiments on a 2021 MacBook Pro running macOS Monterey (version 12.0.1) and equipped with Apple M1 Pro and 16 GB of RAM. The compiler was Apple clang version 13.1.6, and the source code was compiled with the C++17 standard.

Following [9], we considered two scenarios for generating the input trees for the evaluation: *Scenario 1 (Correlated Input Trees)* and *Scenario 2 (Independent Input Trees)*. In short, the input trees in Scenario 1 have more clusters in common, whereas the number of different clusters is higher in Scenario 2. The specific details on how we generated the input trees in each scenario, given k and n, are as follows:

- **Scenario 1 (Correlated Input Trees):** In this scenario, we first generate a binary tree with leaf label set $\{1, \ldots, n\}$ in the uniform model [33]. We then perform a *delete* operation on every internal node of the binary tree, except for the root, with a probability of 0.2 for each operation. Let T_r denote the resulting non-binary tree. Next, we generate k trees based on T_r. Each tree is obtained by repeating $0.05 \cdot n$ times the process of selecting a random non-root node u and a random internal node v such that v is not a descendant of u, detaching the subtree rooted at u, and attaching it to v.
- **Scenario 2 (Independent Input Trees):** In the first step of this scenario, we independently generate k binary trees with leaf label set $\{1, \ldots, n\}$ in the uniform model [33]. Then, we perform a *delete* operation on every internal node of each binary tree, except for the roots, with a probability of 0.2 for each operation. This yields k non-binary trees.

Table 1
Scenario 1 with $k = 1000$ (fixed) and varying values of n.

n	*New*	*Old_kn2*	*Old_k2n*
100	4.81	5.66	6.69
400	22.31	27.62	31.93
1000	56.76	69.93	80.71
2000	123.42	154.16	174.24
4000	262.68	323.06	361.63
6000	424.96	514.60	562.64
8000	598.33	724.73	770.95
10000	780.76	949.44	969.51

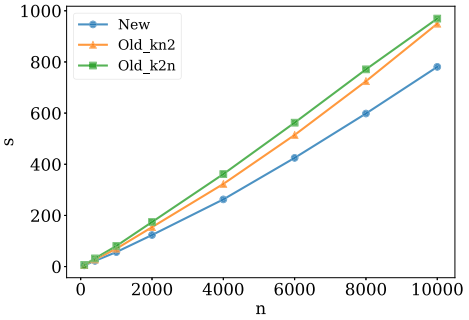


Table 2
Scenario 1 with $n = 10000$ (fixed) and varying values of k.

k	*New*	*Old_kn2*	*Old_k2n*
10	5.68	8.16	7.32
40	24.60	35.11	31.60
100	61.32	85.37	77.34
200	133.81	184.16	170.64
400	279.31	367.13	350.16
600	438.04	555.76	544.92
800	627.75	777.55	759.43
1000	785.20	956.64	975.90

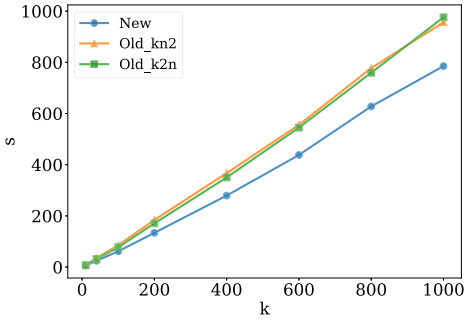


Table 3
Scenario 2 with $k = 1000$ (fixed) and varying values of n.

n	*New*	*Old_kn2*	*Old_k2n*
100	3.73	3.95	5.34
400	16.59	17.46	22.67
1000	45.77	46.26	59.55
2000	99.96	102.37	122.56
4000	229.48	241.12	275.51
6000	357.72	371.38	412.07
8000	438.49	462.20	511.78
10000	576.88	659.28	646.88

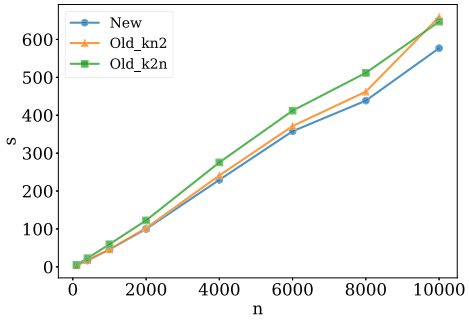
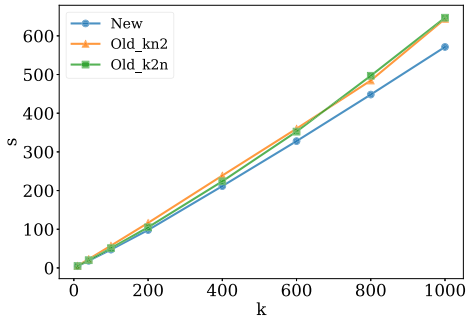


Table 4
Scenario 2 with $n = 10000$ (fixed) and varying values of k.

k	*New*	*Old_kn2*	*Old_k2n*
10	4.40	5.54	4.89
40	18.42	22.72	19.89
100	47.29	57.58	50.98
200	97.92	116.46	105.08
400	211.52	238.77	223.89
600	327.79	360.06	352.45
800	448.20	484.50	496.87
1000	571.32	643.90	646.91



We measured the running time of each algorithm in each scenario for different values of (k, n). The experiments were performed in two settings: firstly, with a fixed value of k at 1000 and varying n up to 10000; secondly, with a fixed value of n at 10000 and varying k up to 1000. For each of the specified values of (k, n), we recorded the average running time over 3 randomly generated inputs of size (k, n).

Tables 1–4 present the average running times (in seconds). Table 1 contains the results for Scenario 1 (Correlated Input Trees) with a fixed value of k and varying values of n. Table 2 shows the running times for Scenario 1 with a fixed value of n and varying values of k. Table 3 includes the running times for Scenario 2 (Independent Input Trees) with a fixed value of k and varying values of n. Finally, Table 4 presents the results for Scenario 2 with a fixed value of n and varying values of k.

We observe that *New* works faster than *Old_kn2* and *Old_k2n* for all the examined values of (k, n). This observation is consistent with our theoretical results stating that *New* is asymptotically faster than both *Old_kn2* and *Old_k2n*. According to our theoretical results, if we conducted experiments using higher values for k and n, the speed advantage of *New* would become more significant compared to the other two algorithms. We can also see that all of the algorithms work faster in Scenario 2 (Independent Input Trees) than in Scenario 1 (Correlated Input Trees). That is because when trees are generated independently of each other, they tend to have numerous incompatible pairs of clusters with each cluster occurring in only a

very small number of trees. Therefore, the procedure `Filter_Clusters` removes many of the clusters, leading to smaller subproblems and fewer recursion levels, and hence faster running times.

7. Concluding remarks

We have introduced an $O(kn \log n)$-time algorithm for computing the FDCT, which is asymptotically faster than the best previously known algorithm [8]. Moreover, we have implemented our algorithm and demonstrated that it outperforms the previously fastest implementation from [9]. The improved procedure `Fast_Compute_Weights`, presented as part of our new algorithm, can also be used in algorithms for building the greedy consensus tree [8,15,16], 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.

CRediT authorship contribution statement

Jesper Jansson: Writing – original draft. **Wing-Kin Sung:** Writing – original draft. **Seyed Ali Tabatabaee:** Writing – original draft. **Yutong Yang:** Writing – original draft.

Declaration of generative AI and AI-assisted technologies in the writing process

During the preparation of this work, the authors used “Grammarly” to improve the grammar and the clarity of some sentences. After using this tool, the authors reviewed and edited the content and take full responsibility for the content of the publication.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partially funded by JSPS KAKENHI grant 22H03550/23K24807 and NSERC Discovery Grants. The authors would also like to thank Varun Gupta for some ideas employed in the procedure `Fast_Label_Trees`.

Data availability

The source code is available at: https://github.com/tswddd2/FDCT_new.

References

[1] J. Jansson, W.-K. Sung, S.A. Tabatabaee, Y. Yang, A faster algorithm for constructing the frequency difference consensus tree, in: 41st International Symposium on Theoretical Aspects of Computer Science (STACS 2024), in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 289, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2024, pp. 43:1–43:17, https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.STACS.2024.43.
[2] E.N. Adams III, Consensus techniques and the comparison of taxonomic trees, Syst. Biol. 21 (4) (1972) 390–397.
[3] D. Bryant, A classification of consensus methods for phylogenetics, DIMACS Ser. Discret. Math. Theor. Comput. Sci. 61 (2003) 163–184.
[4] P.A. Goloboff, Minority rule supertrees? MRP, Compatibility, and Minimum Flip may display the least frequent groups, Cladistics 21 (3) (2005) 282–294.
[5] P.A. Goloboff, J.S. Farris, M. Källersjö, B. Oxelman, M.J. Ramírez, C.A. Szumik, Improvements to resampling measures of group support, Cladistics 19 (4) (2003) 324–332.
[6] P.A. Goloboff, J.S. Farris, K.C. Nixon, TNT, a free program for phylogenetic analysis, Cladistics 24 (5) (2008) 774–786.
[7] J. Dong, D. Fernández-Baca, F. McMorris, R.C. Powers, Majority-rule (+) consensus trees, Math. Biosci. 228 (1) (2010) 10–15.
[8] P. Gawrychowski, G.M. Landau, W.-K. Sung, O. Weimann, A faster construction of greedy consensus trees, in: 45th International Colloquium on Automata, Languages, and Programming (ICALP 2018), in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 107, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2018, pp. 63:1–63:14, https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.ICALP.2018.63.
[9] J. Jansson, R. Rajaby, C. Shen, W.-K. Sung, Algorithms for the majority rule (+) consensus tree and the frequency difference consensus tree, IEEE/ACM Trans. Comput. Biol. Bioinform. 15 (1) (2018) 15–26.
[10] M. Steel, J.D. Velasco, Axiomatic opportunities and obstacles for inferring a species tree from gene trees, Syst. Biol. 63 (5) (2014) 772–778.
[11] J.D. Velasco, The foundations of concordance views of phylogeny, Philos. Theory Pract. Biol. 11 (20) (2019).
[12] R.R. Sokal, F.J. Rohlf, Taxonomic congruence in the Leptopodomorpha re-examined, Syst. Zool. 30 (3) (1981) 309–325.
[13] T. Margush, F.R. McMorris, Consensus n-trees, Bull. Math. Biol. 43 (2) (1981) 239–244.
[14] J. Felsenstein, PHYLIP Version 3.6, Software Package, Department of Genome Sciences, University of Washington, Seattle, USA, 2005.
[15] J. Jansson, C. Shen, W.-K. Sung, Improved algorithms for constructing consensus trees, J. ACM 63 (3) (2016) 28:1–28:24.
[16] H. Wu, Near-optimal algorithm for constructing greedy consensus tree, in: 47th International Colloquium on Automata, Languages, and Programming (ICALP 2020), in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 168, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2020, pp. 105:1–105:14, https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.ICALP.2020.105.

[17] M. Crochemore, C.S. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter, T. Waleń, Extracting powers and periods in a word from its runs structure, Theor. Comput. Sci. 521 (2014) 29–41.
[18] W.H. Day, Optimal algorithms for comparing trees with labeled leaves, J. Classif. 2 (1985) 7–28.
[19] W.D. Allmon, U.E. Smith, What, if anything, can we learn from the fossil record about speciation in marine gastropods? Biological and geological considerations, Amer. Malacolog. Bull. 29 (1) (2011) 247–276.
[20] N. García, A.W. Meerow, D.E. Soltis, P.S. Soltis, Testing deep reticulate evolution in Amaryllidaceae tribe Hippeastreae (Asparagales) with ITS and chloroplast sequence data, Syst. Bot. 39 (1) (2014) 75–89.
[21] J. 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, Ph.D. thesis, Oxford Brookes University, 2021.
[22] G. Han, L.M. Chiappe, S.-A. Ji, M. Habib, A.H. Turner, A. Chinsamy, X. Liu, L. Han, A new raptorial dinosaur with exceptionally long feathering provides insights into dromaeosaurid flight performance, Nat. Commun. 5 (2014) 4382.
[23] M.C. Langer, B.W. McPhee, J.C.d.A. Marsola, L. Roberto-da Silva, S.F. Cabreira, Anatomy of the dinosaur *Pampadromaeus barberenai* (Saurischia–Sauropodomorpha) from the Late Triassic Santa Maria Formation of southern Brazil, PLoS ONE 14 (2) (2019) e0212543.
[24] C. Lindqvist, J. De Laet, R.R. Haynes, L. Aagesen, B.R. Keener, V.A. Albert, Molecular phylogenetics of an aquatic plant lineage, Potamogetonaceae, Cladistics 22 (6) (2006) 568–588.
[25] C. Molineri, A cladistic revision of *Tortopus* Needham & Murphy with description of the new genus *Tortopsis* (Ephemeroptera: Polymitarcyidae), Zootaxa 2481 (2010) 1–36.
[26] C. Molineri, F.F. Salles, Phylogeny and biogeography of the ephemeral *Campsurus* Eaton (Ephemeroptera, Polymitarcyidae), Syst. Entomol. 38 (2) (2013) 265–277.
[27] C. Molineri, F.F. Salles, J.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 (2015) 45–128.
[28] R. Cole, M. Farach-Colton, R. Hariharan, T. Przytycka, M. Thorup, An $O(n \log n)$ algorithm for the maximum agreement subtree problem for binary trees, SIAM J. Comput. 30 (5) (2000) 1385–1404.
[29] M.A. Bender, M. Farach-Colton, The level ancestor problem simplified, Theor. Comput. Sci. 321 (1) (2004) 5–12.
[30] O. Berkman, U. Vishkin, Finding level-ancestors in trees, J. Comput. Syst. Sci. 48 (2) (1994) 214–230.
[31] S.K. Kim, J.-S. Cho, S.-C. Kim, Path maximum query and path maximum sum query in a tree, IEICE Trans. Inf. Syst. 92 (2) (2009) 166–171.
[32] The Boost C++ Libraries, https://www.boost.org/. (Accessed 14 September 2023).
[33] A. McKenzie, M. Steel, Distributions of cherries for two models of trees, Math. Biosci. 164 (1) (2000) 81–92.