A Faster Algorithm for Computing the Kernel of Maximum Agreement Subtrees
Biing-Feng Wang, Krister Swenson

To cite this version:
Biing-Feng Wang, Krister Swenson. A Faster Algorithm for Computing the Kernel of Maximum Agreement Subtrees. IEEE/ACM Transactions on Computational Biology and Bioinformatics, Institute of Electrical and Electronics Engineers, In press, pp.1-1. 10.1109/TCBB.2019.2922955. lirmm-02410429

HAL Id: lirmm-02410429
https://hal-lirmm.ccsd.cnrs.fr/lirmm-02410429
Submitted on 13 Dec 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
A Faster Algorithm for Computing the Kernel of Maximum Agreement Subtrees

Biing-Feng Wang and Krister M. Swenson

Abstract—The maximum agreement subtree method determines the consensus of a collection of phylogenetic trees by identifying maximum cardinality subsets of leaves for which all input trees agree. The trees induced by these maximum cardinality subsets are maximum agreement subtrees (MASTs). A single MAST may be misleading, since there can exist two MASTs which share almost no leaves; nevertheless, it may be impossible to inspect all MASTs, since the number of MASTs can be exponential in the number of leaves. To overcome this drawback, Swenson et al. suggested to further summarize the information common to all MASTs by their intersection, which is called the kernel agreement subtree (KAST). The construction of the KAST is the focus of this paper. Swenson et al. had an $O(kn^3 + n^4 + nd^4)$ time algorithm for computing the KAST of $k$ trees on $n$ leaves, in which at least one tree has maximum degree $d$. In this paper, an $O(kn^3 + n^4)$-time algorithm is presented. We demonstrate the efficiency of our algorithm on simulated trees as well as on ribosomal RNA alignments, where trees with 13,000 taxa took only hours to process, whereas the previous algorithm did not terminate after a week of computation.

Index Terms—Algorithms, phylogenetic trees, consensus trees, agreement subtrees

1. Introduction

The reconstruction of evolutionary relationships among sets of genes or sets of species is fundamental. These relationships often take the form of a phylogenetic tree. When combined with geographical data, phylogenies are essential to understanding the movements and interactions of populations [12]. When combined with geographical and temporal data, they are essential to understanding the spread of epidemics [22]. While the utility of phylogenies is evident in areas like phylogeography and epidemiology, they are also at the heart of seemingly less related fields like functional genomics. Indeed, large-scale studies linking a biological trait to a function rely on phylogenetic relationships to differentiate between "Selected Effect" and "Causal Role" [26].

It is, however, often difficult to find the true phylogenetic tree for a set of taxa. Many methods for constructing phylogenetic trees have been proposed, and many different evolutionary characters can be considered [11], [17], [23], [24], [40]. The use of different methods or different evolutionary characters, however, may result in different trees on the same set of taxa. These trees need to be compared or summarized.

One approach is to calculate a numerical index of agreement or a distance between rival trees. Many tree comparison metrics have been proposed for this purpose, such as the co-phenetic correlation coefficient [41], the path-difference distance [43], [47], the nearest-neighbor interchange (NNI) distance [14], [48], the Robinson-Foulds (RF) distance [15], [39], the quartet distance [8], and the matching distance [33].

Another approach is to compute a new tree that represents the information shared by the rival trees. Such an approach is called a consensus method and the computed tree is called a consensus tree. Since Adams [2] introduced the first consensus method in 1972, a great variety of different consensus methods have been developed and studied. Interested readers may refer to [6], [9] for excellent surveys.

Most consensus methods assume that all the input trees have equal taxon sets, and output a tree having the same taxon set as the input trees. Two widely-used examples are the strict consensus tree and the majority-rule consensus tree [36]. Removing an edge from a phylogenetic tree yields a bipartition of the entire taxon set. The strict consensus tree contains exactly those bipartitions common to all input trees while the majority-rule consensus tree contains exactly those bipartitions that appear in more than half of the input trees.

A rogue leaf in a collection of phylogenetic trees is one whose position is obviously different from tree to tree. These two consensus methods are susceptible to the presence of rogue leaves. Even a small number of rogue leaves may substantially increase tree distances, and deteriorate the resolutions (i.e. the number of internal edges) of the strict and majority-rule consensus trees [1], [38], [45], [49].

One way to overcome the problem caused by rogue leaves is to use a consensus subtree method, which allows some taxa to be removed from the input trees. The most popular of such methods is the agreement subtree method, introduced by Finden and Gordon [25]. This method determines the consensus of a collection of trees by identifying maximum cardinality subsets of leaves for which all input trees agree. The trees induced by these maximum cardinality subsets are maximum agreement subtrees (MASTs). The problem of constructing MASTs has been extensively studied in the literature [3], [7], [13], [18]–[21], [25], [27], [29], [30], [32], [44].

The agreement subtree method avoids the problem caused by rogue leaves. However, unlike the strict and majority-rule consensus trees, MASTs of a collection of trees are not necessarily unique. The number of MASTs can be exponential in the number of leaves [30] and there can exist two MASTs which share almost no leaves [45]. Therefore, the agreement subtree method has the following disadvantage: using a single...
MAST to represent the information shared by rival trees can be misleading, while it may be impossible to inspect all MASTs.

Aiming at providing a baseline method to report a subtree of high confidence that is not susceptible to rogue leaves, Swenson et al. [45] suggested using the summary of the information common to all MASTs, as represented by their intersection. This is called the kernel agreement subtree (KAST). Like the agreement subtree, this new consensus subtree method avoids the problem caused by rogue leaves. In addition, as with the strict consensus tree, it reports a single subtree of highest confidence. Experimental results showed that the KAST can be used as a baseline method to find subtrees of confidence, to report subsets of input trees for which we are confident, and to be an indicator of randomness in the input [45]. In summary, the KAST is complementary to the strict consensus tree, since it reports a leaf set of high confidence while the strict consensus tree reports an edge set of high confidence.

A fast algorithm for computing the KAST is of interest. In the age of modern phylogenetic inference researchers are now addressing datasets with thousands [5], [31], [35], or even tens-of-thousands of taxa [34], and equally as many trees [50] derived from diverse evolutionary characters [46]. Version 4.0a123 of the popular phylogenetic program PAUP* [46] includes an implementation of the KAST that does not scale to datasets of these sizes.

The focus of this paper is speeding up the construction of the KAST. Let $T$ be a set of $k$ phylogenetic trees on a set of $n$ taxa. Assume that at least one of the trees in $T$ has degree bounded by a constant $d$. The current best known algorithms for finding a MAST of $T$ are due to Farach et al. [19] and Bryant [7], which require $O(kn^2 + n^3)$ time. Although the number of all MASTs can be exponential, Swenson et al. [45] showed that the KAST of $T$ can be computed in polynomial time through a modification of Bryant's MAST algorithm. They did not analyze the complexity of their algorithm, yet claimed a running time similar to that of Bryant's by ignoring the time necessary to compute set operations. In Section 4, a detailed analysis is given, which shows that their algorithm takes $O(kn^3 + n^4 + n^{d+1})$ time. In this paper, a faster algorithm is presented. The presented algorithm requires $O(kn^3 + n^d)$ time, which matches the current best upper bound for the MAST problem.

The experimental section applies our algorithm to simulated data, where rogue taxa are added to a fixed topology. On a pair of trees with 2,000 taxa our new algorithm is on average over 40 times faster than the previous, and uses more than an order of magnitude less RAM. The importance of our innovation is glaring when applying our algorithm to trees produced from large ribosomal RNA alignments. Four of these tests did not terminate using the old algorithm, after a week of computation using more than 100 gigabytes of RAM on a server. Our new algorithm took at most 8 hours and 9 gigabytes of RAM for all datasets. A positive side effect of our new algorithm is significant savings in memory consumption.

The remainder of this paper is organized as follows. Section 2 introduces notation and definitions. Section 3 reviews Bryant's MAST algorithm. Section 4 describes Swenson et al.'s KAST algorithm. Section 5 gives a faster algorithm for the KAST problem, which requires $O(kn^3 + n^4 + n^d)$ time. For $d \geq 4$, the algorithm in Section 5 is as fast as the current best MAST algorithms. However, for the cases of $d = 2$ and 3, it is slower by a factor of $n$. We note that these are the most important special cases, because in practice, phylogenetic trees usually have very small degrees, typically no larger than three [3]. Section 6 shows how to solve the KAST problem in $O(kn^3 + n^d)$ time. Section 7 presents our experimental results, while Section 8 concludes the paper.

2. Notation and definitions

A phylogenetic tree is a tree in which the leaves are uniquely labeled by a set of taxa. For convenience, a leaf of a phylogenetic tree is simply identified with its label. A phylogenetic tree can be rooted or unrooted. In this paper, only rooted trees are considered. The degree of a node is its number of children. We assume that the degree of every non-leaf node is at least two, so that the number of non-leaf nodes is bounded by the number of leaves. As in all MAST literature, a node with degree $d$ greater than two is considered a "hard polypotomy", the alternative being a "soft polypotomy" which ambiguously represents all possible tree topologies on the $d$ children.

Consider a phylogenetic tree $T$. If the path from a node $a$ to the root passes through a node $b$, we call $b$ an ancestor of $a$ and call $a$ a descendant of $b$. A proper descendant of a node $v$ is a descendant of $v$ which is not $v$ itself. A proper ancestor is defined similarly. For any two nodes $a$, $b$ of $T$, the lowest common ancestor (LCA) of $a$ and $b$ is the ancestor of $a$ and $b$ that is a descendant of all ancestors of $a$ and $b$. For any subset $S$ of the leaves of $T$, we follow convention by denoting $T_S$ as the subtree of $T$ induced by $S$. $T_S$ is the tree with leaf set $S$ and interior node set $\{x : x$ is the LCA of some pair of leaves in $S\}$ inheriting the ancestor relation from $T$ (i.e. for all $a, b \in S$, the LCAs of $a, b$ in $T$ and $T_S$ are the same). Two trees $T_1$ and $T_2$ on the same label set are isomorphic if there is a 1-1 mapping between their internal nodes such that the LCA of any two leaves $a$, $b$ in $T_1$ is mapped to the LCA of $a$, $b$ in $T_2$.

Figure 1. The three MASTs of $T = \{T_1, T_2\}$.

Let $T = \{T_1, T_2, \ldots, T_k\}$ be a set of phylogenetic trees on the same set $L$ of $n$ labels. Throughout this paper, we assume that at least one of the trees in $T$ has maximum degree $d$, where $d \geq 2$ is a constant. An agreement subtree of $T$ is a tree $T$ such that $T, T_1|S, T_2|S, \ldots$, and $T_k|S$ are mutually isomorphic, where $S$ is the leaf set of $T$. The leaf set of an agreement subtree is called an agreement set. The size of an agreement subtree is the cardinality of its leaf set. A maximum agreement

1545-5965 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
subtree (MAST) is an agreement subtree of maximum size. For example, for the set $\mathcal{T} = \{T_1, T_2\}$ in Figure 1(a), the three MASTs of $\mathcal{T}$ are depicted in Figure 1(b). The MAST problem is to find a MAST of $\mathcal{T}$. The kernel agreement subtree (KAST) of $\mathcal{T}$, denoted by $\text{KAST} (\mathcal{T})$, is the intersection of the leaf sets of all MASTs of $\mathcal{T}$. For example, in Figure 1, we have $\text{KAST} (\mathcal{T}) = \{ a, b, d, e, f \} \cap \{ a, c, d, e, f \} \cap \{ b, c, d, e, f \} = \{ d, e, f \}$. The KAST problem is to compute the KAST of $\mathcal{T}$.

![Figure 2. Rooted and fan triples.](image)

In the following, we introduce some notation and definitions that are used throughout this paper. Let $\mathcal{T}$ be a phylogenetic tree. The leaf set of $\mathcal{T}$ is denoted by $L(\mathcal{T})$. For each node $v$, the subtree rooted at $v$ is denoted by $T(v)$ and the subtrees rooted at the children of $v$ are called the subtrees of $v$. The subtrees of the root are the maximal subtrees of $\mathcal{T}$. Let $a, b, c$ be leaves of $\mathcal{T}$. Figure 2 depicts the four possible topologies of the subtree induced by $\{ a, b, c \}$. For the first three cases, we say that $a, b, c$ form a rooted triple in $\mathcal{T}$. We use $ab$ to denote the root of the triple in which the LCA of $a, b$ is a descendant of the LCA of $a, c$. For the last case, we say that $a, b, c$ form a fan triple in $\mathcal{T}$ and use $(a, b, c)$ to denote the fan triple.

A set $S$ of leaves is a fan set of $\mathcal{T}$ if and only if $S \subseteq L(\mathcal{T})$ and $S$ is a fan set of $\mathcal{T}$ if in the induced tree $T_S$ all the leaves are children of the root. For example, in $T_2$ of Figure 1, both $\{ a, b, c \}$ and $\{ a, d, f \}$ are fan sets. For a pair $(a, b) \in \mathcal{T}$, we use $\text{lca}(a, b)$ to denote the LCA of $a, b$ in a tree $\mathcal{T} \in \mathcal{T}$ and use $\text{lca}(a)$ to denote the sequence $(\text{lca}(a, b), \text{lca}(a, b), \ldots, \text{lca}(a, b))$. The set of all rooted triplets common to all trees in $\mathcal{T}$ is denoted by $\mathcal{R}$ and the set of all fan triplets common to all trees in $\mathcal{T}$ is denoted by $\mathcal{F}$. A set is called a fan set of $\mathcal{T}$ if it is a fan set of every tree in $\mathcal{T}$. Since at least one tree has maximum degree $d$, it is easy to see that any fan set of $\mathcal{T}$ has size at most $d$.

3. Bryant's MAST algorithm

This section reviews Bryant's MAST algorithm. For brevity, only the computation for the size of a MAST of $\mathcal{T}$, denoted by $\text{mast}(\mathcal{T})$, is described. The following lemma provides the basis of Bryant's algorithm.

Lemma 3.1. [7] A tree $T$ is an agreement subtree of $\mathcal{T}$ if and only if $r(T) \subseteq \mathcal{R}$ and $f(T) \subseteq \mathcal{F}$, where $r(T)$ and $f(T)$ are, respectively, the set of rooted triplets and the set of fan triplets in $\mathcal{T}$.

For each pair $(a, b) \in \mathcal{T}$, define the following:
- $\mathcal{A}(a, b)$: the set of agreement subtrees of $\mathcal{T}$ in each of which $a, b$ are leaves and the LCA of $a, b$ is the root;
- $m(a, b)$: the size of the largest trees in $\mathcal{A}(a, b)$; and
- $\mathcal{M}(a, b)$: the set of largest trees in $\mathcal{A}(a, b)$.

Clearly, $\text{mast}(\mathcal{T})$ is the maximum value of $m(a, b)$ over all $(a, b) \in \mathcal{T}$. Thus, to compute $\text{mast}(\mathcal{T})$, it suffices to compute $m(a, b)$ for every $(a, b) \in \mathcal{T}$.

Bryant's algorithm computes all $m(a, b)$ by dynamic programming. It determines $m(a, b)$ before $m(a', b')$ if in $T$ the LCA of $a, b$ is a proper descendant of the LCA of $a', b'$. We proceed by discussing the computation for a fixed pair $(a, b) \in \mathcal{T}$. If $a = b$, we simply have $m(a, b) = 1$. Assume that $a \neq b$. Consider a tree $Q \in \mathcal{M}(a, b)$. Since the LCA of $a, b$ is the root of $Q$, we know that $a$ and $b$ are in different maximal subtrees of $Q$. Let $Q_a$ be the maximal subtree containing $a$, $Q_b$ be the maximal subtree containing $b$, and $Q_1, Q_2, \ldots, Q_k$ be the remaining maximal subtrees, if there are any. Note that $r \leq d - 2$, since at least one tree in $\mathcal{T}$ has maximum degree $d$. For $1 \leq j \leq r$, let $c_j$ be any leaf of $Q_j$.

For $1 \leq i \leq k$, let $a_i = \text{lca}(a, b)$. By the definition of $\mathcal{M}(a, b)$, $Q$ is a largest agreement subtree of $\{ T_i(a_i) : 1 \leq i \leq k \}$ under the condition that $a$ and $b$ should be contained in an agreement subtree. Since $\{ a, b, c_1, c_2, \ldots, c_k \}$ is a fan set of $Q$, according to Lemma 3.1, we know that in each $T_i$, $1 \leq i \leq k$, the leaves $a, b, c_1, c_2, \ldots, c_k$ are in different subtrees of $a_i$. As a result, the sizes of $Q_1, Q_2, Q_3, \ldots, Q$ can be discussed individually.

Consider the subtree $Q_1$ first. For $1 \leq i \leq k$, let $A_i$ be the subtree containing $a$ that is rooted at a child of $a_i$. Then, $Q_1$ is a largest agreement subtree of $\{ A_i : 1 \leq i \leq k \}$ under the condition that $a$ should be contained in an agreement subtree. Let $X(a) = \{ x : a, b \in x \}$. Note that $X(a)$ is the set of common labels of $A_1, A_2, \ldots, A_k$ and thus only the labels in $X(a)$ can be leaves of an agreement subtree of $\{ A_i : 1 \leq i \leq k \}$. Clearly, under the condition that $a$ should be contained in an agreement subtree, a tree $T$ is an agreement subtree of $\{ A_i : 1 \leq i \leq k \}$ if and only if $T \subseteq X(a)$ for some $x \in X(a)$. Therefore, the size of $Q_1$ can be computed as

$$m(a, b) = \max (m(a, x) : x \in X(a))$$

Similarly, the size of $Q_j$ can be computed as $m(b(a))$ and the size of each $Q_1, 1 \leq j \leq r$, can be computed as $m(c_j)$ (or $m(c_j)$). Consequently, given $c_1, c_2, \ldots, c_k$, the size of $Q$ can be computed as

$$m(a, b) + m(b(a)) + \sum_{j=1}^{k} m(c_j)$$

We proceed by deriving a recurrence for $m(a, b)$. Let $C(a, b) = \{ c : (a, b, c) \in \mathcal{F} \}$. If $C(a, b)$ is empty, all trees in $\mathcal{A}(a, b)$ are binary and thus we have

$$m(a, b) = m(a)(b) + m(b)(a)$$

Assume that $C(a, b)$ is not empty. Define $G(a, b)$ to be a graph in which for each $c \in (a, b)$ there is a vertex $c$ with weight $m(c(a))$ and for every pair $(u, v)$ of vertices, there is an edge between $u$ and $v$ if and only if $(a, u, v) \in \mathcal{F}$. A clique in a graph is a subset of the vertices such that every two distinct vertices in the clique are connected by an edge. By the definition of $G(a, b)$, a set $U$ is a clique in $G(a, b)$ if and only if $\{ a, b \} \cup U$ is a fan set of $\{ T_i(a_i) : 1 \leq i \leq k \}$. As a result, it can be concluded that the value of $(m(a, b), m(b)(a) + m(b)(a))$ if and only if $w$ is the maximum total weight of any clique in $G(a, b)$. Therefore,

$$m(a, b) = m(a)(b) + m(b)(a) + \sum_{c \in S} m(c(a))$$

where $S$ is a maximum weight clique in $G(a, b)$. Bryant's algorithm for computing each $m(a, b)$ is formally described as follows.

Procedure FINDMAST(a, b)
input: $(a, b) \in \mathcal{T}$
output: $m(a, b)$
begin
The complexity of FINDMAST is analyzed as follows. Lines 1-3 require $O(n)$ time. Lines 4-5 require $O(C(a, b) \times n) = O(n^2)$ time. The finding of a maximum weight clique in Line 6 is done by considering all subsets $\{c_1, c_2, \ldots, c_i\}$ of $C(a, b)$, where $0 \leq i \leq d - 2$. Since $|C(a, b)| \leq n$ and $d$ is a constant, the number of such subsets is at most $\sum_{i=0}^{d-2} \binom{n}{i} = O(n^d)$. Each subset can be checked in $O(1)$ time. Thus, Line 6 requires $O(n^d)$ time. Line 7 requires $O(d) = O(1)$ time. Therefore, the time complexity of FINDMAST is $O(n^2 + n^d)$.

To solve the MAST problem, FINDMAST is called for every $(a, b) \in L^2$. As a result, a total of $O(n^2 \times (n^2 + n^d)) = O(n^4 + n^{d+2})$ time is required. The $n^d$ term is contributed by Lines 4-5, which computes $m(a)$ for each $c \in C(a, b)$. This term can be reduced to $n^d$ by simply avoiding the work of recomputing the value of $m(a)$ for the same pair of $(c, a)$. Therefore, the total time spent on computing the values of all $m(a)$ is $O(n^4 + n^d)$. The construction of $R$ and $J$ takes $O(n^4)$ time [7]. Consequently, the following is obtained.

**Theorem 3.1.** [7] The MAST problem on a set of $k$ trees can be solved in $O(kn^4 + n^d)$ time, where $n$ is the size of the trees and at least one tree has maximum degree $d$.

### 4. Swenson et al.’s KAST algorithm

Swenson et al. solved the KAST problem through a modification of Bryant's MAST algorithm. Let $\mathcal{M}(\cdot, \cdot)$, $m(\cdot)$, $m(\cdot)$, $C(\cdot)$, $\cdot$, be defined as the same as in Section 3. For each pair $(a, b) \in L^2$, define

$$\mathcal{K}(a, b) = \bigcap_{x \in \mathcal{X}(a, b)} \mathcal{X}(a, b).$$

Then, the KAST of $T$ is the intersection of all $\mathcal{K}(a, b)$ such that $m(a, b)$ is a KAST($T$). Thus, the KAST problem can be solved by computing $\mathcal{K}(a, b)$ for every $(a, b) \in L^2$.

Consider the computation of $\mathcal{K}(a, b)$ for a fixed pair $(a, b) \in L^2$. For convenience, define

$$X(a, b) = \{x : x \in \mathcal{X}(a, b), m(a, x) = m(a, b)\}$$

$$\mathcal{K}(a, b) = \bigcap_{x \in X(a, b)} \mathcal{X}(a, b).$$

For example, consider the set $T = \{T_1, T_2\}$, where $T_1 = \{(((((a, w), x), p), y), z), (b, c)\}$ and $T_2 = \{(((a, w), z), q), (a, c)\}$. Then, $\mathcal{X}(a, b) = \{p, q, x, y, z, w, m(a, w) = 2, m(a, x) = m(a, y) = m(a, z) = 3, m(a, p) = m(a, q) = 4\}$. Hence, $m(a, b) = 4$ and $X(a, b) = \{p, q\}$. Since $M(a, p) = \{((a, w), x), p)\}$ and $M(a, q) = \{((a, w), y), q)\}$, we have $K(a, p) = \{a, w, x, p\}$ and $K(a, q) = \{a, w, y, q) \cap \{a, w, z, q\} = \{a, w, q\}$. Thus, $K(a, b) = K(a, p) \cap K(a, q) = \{a, w, q\}$.

If $a = b$, we have $\mathcal{K}(a, b) = \{a\}$. Assume that $a \neq b$. By definition, $\mathcal{K}(a, b)$ is the set of leaves common to all trees in $\mathcal{M}(a, b)$. Consider the case that $C(a, b)$ is empty. In this case, all trees in $\mathcal{M}(a, b)$ are binary trees. Under the condition that $a$ should be contained in an agreement subtree, the set of all maximum agreement subtrees of $T$ on the leaf set $\mathcal{X}(a, b)$ is

$$\mathcal{M}(a, b) = \{Q : Q \in \mathcal{M}(a, x), x \in X(a, b)\}.$$
6. $\Phi(a, b) \leftarrow \bigcap_{s \in \gamma(a, b)} (\bigcup_{c \in S} K_c(a))$
7. $K(a, b) \leftarrow K(a(b)) \cup K(b(a)) \cup \Phi(a, b)$
8. return $K(a, b)$

Assume that $R$, $F$, and all $m(a(b))$ and $m(a, b)$, where $(a, b) \in L^2$, have been computed by performing Bryant's algorithm. The time complexity of FINDKAST is analyzed as follows. Line 1 takes $O(1)$ time. Lines 2-5 compute the following sets: $K(a(b))$, $K(b(a))$, and $K(c(a))$ of each $c \in C(a, b)$. According to (2), each of these sets needs a sequence of $O(n)$ set intersection operations to compute. A set operation requires $O(n)$ time. Therefore, the running time of Lines 2-5 is $O((C(a, b) + 2) \times n \times n) = O(n^3)$. Lines 6 and 7 compute $K(a, b)$ according to (4) and (5). The computation of $\Phi(a, b)$ in Line 6 needs a sequence of $O(d \times |S(a, b)|) = O(n^{d-2})$ set union and set intersection operations. Therefore, Line 6 requires $O(n^{d-1})$ time. Line 7 takes $O(n)$ time. As a result, the overall running time of Lines 6-7 is $O(n^{d-1})$.

To solve the KAST problem, FINDKAST is called for every $(a, b) \in L^2$. For convenience, we say that the total running time of Lines 2-5 of FINDKAST over all $(a, b) \in L^2$ is spent on computing all $K(a(b))$ and the total running time of Lines 6-7 of FINDKAST over all $(a, b) \in L^2$ is spent on computing all $K(a, b)$. For a fixed pair $(a, b)$, Lines 2-5 require $O(n^3)$ time. However, the time spent on computing all $K(a(b))$ is $O(n^3)$, instead of $O(n^2)$, since we only need to compute $K(a(b))$ once for each distinct $(a, b)$. For a fixed pair $(a, b)$, Lines 6-7 requires $O(n^{d-1})$ time. Therefore, the time spent on computing all $K(a, b)$ is $O(n^{d-1})$. Bryant's algorithm for computing $R$, $F$, and all $m(a(b))$ and $m(a, b)$ requires $O(kn^3 + n^d)$ time. Consequently, we obtain the following.

**Theorem 4.1.** [45] The KAST problem on a set of $k$ trees can be solved in $O(kn^3 + n^d + n^{d+1})$ time, where $n$ is the size of the trees and at least one tree has maximum degree $d$.

An important case of the KAST problem is the case when the given trees are binary trees. For this case, the following is obtained.

**Corollary 4.1.** [45] The KAST problem on a set of $k$ binary trees can be solved in $O(kn^3 + n^4)$ time, where $n$ is the size of the trees.

### 5. An $O(kn^3 + n^4 + n^d)$-time KAST algorithm

A bottleneck of the algorithm in Section 4 is the computation of $\Phi(a, b)$ in Line 6 of FINDKAST. According to (5), computing each $\Phi(a, b)$ needs $O(n^2)$ set operations. In this section, we improve the upper bound of the KAST problem to $O(kn^3 + n^d + n^d)$ by showing that each $\Phi(a, b)$ can be computed by using amortized $O(1)$ set operations.

Consider the computation of $\Phi(a, b)$ for a fixed pair $(a, b) \in L^2$. As in Section 3, for $1 \leq i \leq k$, we define $a_i$ to be $lca(a, b)$. For a tree $T(a_i)$, $1 \leq i \leq k$, and a leaf $l$ of $T(a_i)$, we use $\beta(i, l)$ to denote the subtree of $\alpha_i$ that contains $l$. For example, in Figure 3, $\beta(1, x) = H_1$, $\beta(2, x) = I_2$, and $\beta(3, x) = J_3$. Let $G(a, b)$, $S(a, b)$, and $Q(S)$ be defined the same as in Section 4. Our first intent is to avoid redundant set operations for computing $\Phi(a, b)$ by utilizing the topological structure of $T_i(a_i)$. We start with the following two simple observations.

**Lemma 5.1.** Any clique in $G(a, b)$ contains at most one leaf of each subtree of $\alpha_i$.

**Proof.** Let $S \subseteq C(a, b)$ be a set containing two leaves, say $u$ and $v$, of a subtree of $\alpha_i$. In $G(a, b)$, there is an edge between $u$ and $v$ if and only if $(a, u, v) \in F$. Since $u$ and $v$ is a rooted triple of $T_i$, we have $(a, u, v) \notin F$. Thus, $S$ is not a clique in $G(a, b)$ and the lemma holds.

**Lemma 5.2.** Let $S$ be a clique in $S'(a, b)$. A leaf $l \in L$ can be contained in a tree $Q \in Q(S)$ only if there is leaf $s \in \{a, b\}$ such that $l$ and $s$ are in the same subtree of $\alpha_i$ for every $i = 1, 2, ..., k$.

**Proof.** According to the construction of the trees in $Q(S)$, $Q$ is made up by taking a tree from $M(a(b))$, a tree from $M(b(a))$, and a tree from $M(s(a))$ for each $s \in S$. The trees in $M(a(b))$ are agreement subtrees of $\{\beta(i, a) : 1 \leq i \leq k\}$, and thus only contains common to all $\beta(i, a)$, $1 \leq i \leq k$. Similarly, the trees in $M(b(a))$ contain only leaves common to all $\beta(i, b)$, $1 \leq i \leq k$, and for each $s \in S$ the trees in $M(s(a))$ contain only leaves common to all $\beta(i, s)$, $1 \leq i \leq k$. Therefore, a leaf $l \in L$ can be contained in a tree $Q \in Q(S)$ only if there exists a leaf $s \in \{a, b\} \cup S$ such that $l$ is in $\beta(i, s)$ for every $i = 1, 2, ..., k$. Thus, the lemma holds.

For convenience, if a clique $S$ in $G(a, b)$ contains a leaf of a subtree $H$ of $\alpha_i$, we say that $S$ uses $H$. For example, in Figure 3, $S = \{z, v\}$ is a clique in $G(a, b)$ and it uses $H_1$ and $H_3$. We have the following.

**Lemma 5.3.** Let $H$ be a subtree of $\alpha_i$ that contains neither $a$ nor $b$. If $H$ is not used by all cliques in $S'(a, b)$, then $\mathcal{K}(a, b)$ does not contain any leaf of $H$.

**Proof.** Assume that $H$ is not used by a clique $S \in S'(a, b)$. By definition, $\mathcal{K}(a, b)$ is the set of leaves common to all trees in $\mathcal{M}(a, b)$. This lemma can be proved by showing that there exists a tree $Q \in \mathcal{M}(a, b)$ that does not contain any leaf of $H$. Let $Q \in Q(S)$ be a tree. Note that any member of $Q(S)$ is a tree in $\mathcal{M}(a, b)$. Consider any leaf $l$ of $H$. Since $a$ and $b$ are not in $H$ and $H$ is not used by $S$, leaf $l$ is not in $\beta(i, s)$ for all $s \in \{a, b\} \cup S$. As a result, by Lemma 5.2, $Q$ does not contain any leaf of $H$. Thus, the lemma holds.

For each subtree $H$ of $\alpha_i$, let $C(a, b, H)$ be the set of leaves in $H$ that are contained in some clique in $S'(a, b)$. That is, $C(a, b, H) = \bigcup_{s \in \gamma(a, b)} (\mathcal{L}(H) \cap S)$.

For convenience, we say that some leaves of $L$ are coherent with respect to $(a, b)$ if they are in the same subtree of $\alpha_i$ for every $i = 1, 2, ..., k$. For example, in Figure 3, $x$ and $y$ are
coherent; but $x, y,$ and $z$ are not, since in $T_2$, they are not contained in the same subtree of $a_z$. According to Lemma 5.3, in the computation of $K(a, b)$, we can ignore any subtree of $a_z$ that contains neither $a$ nor $b$ and is not used by all cliques in $S' (a, b)$. The following lemma allows us to identify more subtrees that can be ignored.

**Lemma 5.4.** Let $H$ be a subtree of $a_1$ that is used by all cliques in $S'(a, b)$. If $C(a, b, H)$ contains two leaves that are not coherent with respect to $(a, b)$, then the set $K(a, b)$ does not contain any leaf of $H$.

**Proof.** Assume that there are two leaves, say $c$ and $c'$, in $C(a, b, H)$ that are not coherent with respect to $(a, b)$. In the following, we prove this lemma by showing that there are two trees $Q, Q' \in M(a, b)$ such that $L(Q) \cap L(Q')$ does not contain any leaf in $H$.

By the definition of $C(a, b, H)$, there are two cliques, say $S$ and $S'$, in $S'(a, b)$ such that $S$ contains $c$ and $S'$ contains $c'$. Note that $S \neq S'$, since by Lemma 5.1 each of $S$ and $S'$ contains at most one leaf of $H$. Let $Q$ be a tree in $Q'(S)$ and $Q'$ be a tree in $Q'(S')$. Consider a leaf $l$ in $H$. By Lemma 5.1, $c$ is the unique leaf of $S$ that is in $H$. Thus, $c$ is the only leaf of $S$ such that $c$ and $l$ are in the same subtree of $a_1$. Consequently, according to Lemma 5.2, $l$ can be contained in $Q'$ only if $l'$ and $l'$ are coherent with respect to $(a, b)$. Similarly, $l$ can be contained in $Q'$ only if $l'$ and $l'$ are coherent with respect to $(a, b)$. Since $c$ and $c'$ are not coherent with respect to $(a, b)$, we know that $l$ cannot be a leaf of both $Q$ and $Q'$. The lemma holds.

Let $Valid(a, b) = \{H: H$ is a subtree of $a_1$ used by all cliques in $S'(a, b)$ and all leaves in $C(a, b, H)$ are coherent$\}$. According to (4), $K(a, b)$ is the union of three disjoint sets $K(a_1b, b_1), K(b_1a), and \Phi(a, b)$. The leaves of $K(a_1b)$ and $K(b_1a)$ are contributed, respectively, by the subtree of $a_1$ that contains $a$ and the subtree of $a_1$ that contains $b$; and the leaves of $\Phi(a, b)$ are contributed by the subtrees of $a_1$ that contain neither $a$ nor $b$. By Lemmas 5.3 and 5.4, if a subtree $H$ of $a_1$ contains neither $a$ nor $b$, $K(a, b)$ contains a leaf of $H$ only if it is in $Valid(a, b)$. Thus, it can be concluded that only the subtrees in $Valid(a, b)$ can give leaves to $\Phi(a, b)$. Therefore, the right side of equation (5) can be simplified as follows:

\[
\bigcap_{S \subseteq \text{Valid}(a, b)} \bigcup_{c \in S} \text{K}(c(a)) = \bigcap_{S \subseteq \text{Valid}(a, b)} \bigcup_{c \in S} \text{K}(c(a)) \bigcup \{1 \in S \mid \forall c \in \text{Valid}(a, b) \}
\]

Consequently, (5) can be written as

\[
\Phi(a, b) = \bigcup_{H \in \text{Valid}(a, b)} \Phi_1(C(a, b, H)) \bigcup \{1 \in S \mid \forall c \in \text{Valid}(a, b) \}
\]

(6)

where for any $U \subseteq L$,

\[
\Phi(U) = \bigcap_{c \in U} \text{K}(c(a)).
\]

(7)

Clearly, according to (6) and (7), the number of set operations required for computing each $\Phi(a, b)$ is reduced to $O(n)$. Consequently, the total number of set operations required to compute $\Phi(a, b)$ for all $(a, b) \in L$ is $O(n^2)$. In the following, by further utilizing the topological structure of each input tree, we show that the total number of required set operations can be reduced to $O(n^2)$. More specifically, we show that for each $a \in L$, $O(n)$ set operations are sufficient for computing $\Phi(a, b)$ for all $b \in L$.

Consider a fixed $a \in L$. According to (6), to compute $\Phi(a, b)$ for all $b \in L$, it suffices to compute $\Phi_1(C(a, b, H))$ for all $b \in L$ and all $H \in \text{Valid}(a, b)$. If we compute $\Phi(a, b)$ for all $b \in L$ individually according to (4) and (5), it may happen that $\Phi_1(U)$ is computed many times for the same set $U$. For instance, consider the artificial example in Figure 4, in which $k = 3$ and $T_1(a_1), T_2(a_2)$, and $T_3(a_3)$ are isomorphic. In this example, it is easy to see that for each $b \in L(B)$, we have $H \in \text{Valid}(a, b)$ and $C(a, b, H) = L(H)$; and if we compute all $\Phi(a, b)$ individually according to (4) and (5), $\Phi_1(U(L(H)))$ will be computed $|L(B)|$ times. Define $\Gamma_a$ to be the following collection of sets:

\[
\{C(a, b, H) : b \in L, H \in \text{Valid}(a, b)\}.
\]

Then, to compute $\Phi(a, b)$ for all $b \in L$, it suffices to compute $\Phi_1(U(L(H)))$ for all $b \in \Gamma_a$. Our intent is to avoid re-computing the content of $\Phi_1(U)$ for the same set $U$ and to show that $O(n)$ set operations are sufficient for computing $\Phi_1(U)$ for all $b \in \Gamma_a$.

![Figure 4](image-url)

For convenience, we say that the subtrees in $\text{Valid}(a, b)$ are valid for $\Phi(a, b)$. We need the following lemma.

**Lemma 5.5.** Let $H$ be a subtree of $a_1$ that is valid for $\Phi(a, b)$. The value of $m(c(a))$ is the same for every $c \in C(a, b, H)$.

**Proof.** We prove this lemma by contradiction. Suppose that there are two leaves, $c', c'' \in C(a, b, H)$ such that $m(c(a)) < m(c'(a))$. By the definition of $C(a, b, H)$, there is a clique $S' \in S'(a, b)$ which contains $c$. Since $H$ is valid for $C(a, b, H)$, we know that $c$ and $c'$ are coherent with respect to $(a, b)$. Thus, for each $x \in C(a, b)$, we have $(a, c, x) \in F$ only if $(a, c', x) \in F$. As a result, $c$ and $c'$ have the same neighborhood in the graph $G(a, b)$. Therefore, $S - \{c\} \cup \{c'\}$ is clique. Since $m(c(a)) < m(c'(a))$, this clique, $S - \{c\} \cup \{c'\}$, has larger total weight than $S$, contradicting that $S$ is a maximum weight clique in $G(a, b)$. Therefore, the lemma holds.

**Lemma 5.6.** Let $a, b, b'$ be three leaves. If $lca(a, b) \neq lca(a, b')$, we have $C(a, b) \cap C(b, b') = \emptyset$.

**Proof.** Assume that $lca(a, b) \neq lca(a, b')$. Since $lca(a, b) \neq lca(a, b')$, there is a tree $T_i$ such that $lca(a, b) \neq lca(a, b')$, where $1 \leq i \leq k$. Since $lca(a, b) \neq lca(a, b')$ and there is an ancestor-descendant relationship between them, there is no leaf $l$ such that both $(a, b, l)$ and $(a, b', l)$ are fan triples in $T_i$. By definition, if $C(a, b) \cap C(b, b') \neq \emptyset$, there exists a leaf $l$ such that both $(a, b, l)$ and $(a, b', l)$ are fan triples in $T_i$, for every $i = 1, 2, ..., k$. Therefore, it can be concluded that $C(a, b) \cap C(b, b') = \emptyset$. Thus, the lemma holds.

**Lemma 5.7.** Let $a, b, b'$ be three leaves such that $lca(a, b) = lca(a, b')$. Let $H$ be a subtree of $a_1 = lca(a, b)$. If $H$ is valid for both $\Phi(a, b)$ and $\Phi(a, b')$, then either $C(a, b, H) \cap C(a, b', H) = \emptyset$ or $C(a, b, H) = C(a, b', H)$.
Lemma 5.6, we have

Proof. Note that since \( \text{lca}'(a, b) = \text{lca}'(a', b') \), we know that \( b' \) and \( a \) are in different subtrees of \( \alpha_i = \text{lca}(a, b) \) for every \( i = 1, 2, ..., k \); but \( b' \) and \( b \) may be in the same subtree or in different subtrees of \( \alpha_i \). Since \( H \) is valid for \( \Phi(a, b) \), all leaves in \( C(a, b, H) \) are in the same subtree of \( \alpha_i \) for every \( i = 1, 2, ..., k \). Similarly, all leaves in \( C(a', b'; H) \) are in the same subtree of \( \alpha_i \) for every \( i = 1, 2, ..., k \). Two cases are considered.

Case 1. There is a tree \( T_g \), \( 1 \leq g \leq k \), such that the leaves in \( C(a, b, H) \) and \( C(a', b', H) \) are in different subtrees of \( \alpha_g \). (See Figure 5(a) for an illustration, in which \( g = 2 \).)

Since the leaves in \( C(a, b, H) \) and the leaves in \( C(a, b', H) \) are contained, respectively, in two disjoint subtrees of \( \alpha_g \), we have \( C(a, b, H) \cap C(a, b', H) = \emptyset \).

Case 2. The leaves in \( C(a, b, H) \) and \( C(a', b', H) \) are in the same subtree of \( \alpha_i \) for every \( i = 1, 2, ..., k \). (See Figure 5(b)).

In the following, we prove that in this case \( C(a, b, H) = C(a, b', H) \). First, we show that \( C(a, b, H) \subseteq C(a, b', H) \).

Consider a leaf \( c \) in \( C(a, b, H) \). Since \( H \) is valid for \( \Phi(a, b) \), there is a clique \( S' \subseteq S(a', b') \) that contains \( c \). Let \( S' \) be any clique in \( S(a', b') \). Since \( H \) is valid for \( \Phi(a', b') \), \( S' \) contains a leaf \( c' \) in \( C(a', b', H) \). Note that \( c \) and \( c' \) are in the same subtree of \( \alpha_i \) for every \( i = 1, 2, ..., k \). Recall that if a set \( X \) is a clique in \( G(a, b) \) if and only if \( \{ b \} \cup \{ X \} \) is a fan of \( T \). Thus, \( \{ a, b \} \cup S \) is a fan of \( T \). Since \( S \) contains \( c \) and the two leaves \( c, c' \) are in the same subtree of \( \alpha_i \) for every \( i = 1, 2, ..., k \), \( \{ a, b \} \cup (S - \{ c \} \cup \{ c' \}) \) is also a fan of \( T \). That is, \( S - \{ c \} \cup \{ c' \} \) is a clique in \( G(a, b) \). Consequently, \( m(c') \leq m(c) \); otherwise this clique, \( S - \{ c \} \cup \{ c' \} \), has larger total weight than \( S \), contradicting to that \( S \) is a maximum weight clique in \( G(a, b) \). Similarly, from the fact that \( S' \) is a maximum weight clique in \( G(a', b') \), it can also be derived that \( S' - \{ c' \} \cup \{ c \} \) is a clique in \( G(a', b') \) and \( m(c') \leq m(c) \). As a result, it can be concluded that \( m(a) = m'(a) \) and \( S' - \{ c' \} \cup \{ c \} \) is a maximum weight clique in \( G(a', b') \). Therefore, \( c \) is contained in \( C(a', b', H) \). From the above discussion, we know that \( C(a, b, H) \subseteq C(a, b', H) \).

By combining these two statements, we have \( C(a, b, H) = C(a, b', H) \), which completes the proof of this lemma.

By combining Lemmas 5.6 and 5.7, the following is obtained.

**Lemma 5.8.** Let \( a, b, b' \) be three leaves. Let \( H \) be a subtree of \( \text{lca}(a, b) \) that is valid for \( \Phi(a, b) \) and \( H' \) be a subtree of \( \text{lca}(a, b') \) that is valid for \( \Phi(a, b') \). Then, either \( C(a, b, H) \cap C(a, b', H') = \emptyset \) or \( C(a, b, H) = C(a', b', H') = \emptyset \).

**Proof.** If \( \text{lca}(a, b) \neq \text{lca}(a, b') \), the lemma holds, since by Lemma 5.6, we have \( C(a, b, H) \cap C(a, b', H') \subseteq C(a, b) \cap C(a, b') = \emptyset \). Assume that \( \text{lca}(a, b) = \text{lca}(a, b') \). Two cases are considered: \( H \neq H' \) and \( H = H' \). If \( H \neq H' \), we know that \( H \) and \( H' \) are two different subtrees of \( \text{lca}(a, b) \), since \( \text{lca}(a, b) = \text{lca}(a, b') \). Thus, we have \( C(a, b, H) \cap C(a, b', H') = \emptyset \). If \( H = H' \), by Lemma 5.7, either \( C(a, b, H) \cap C(a, b', H') = \emptyset \) or \( C(a, b, H) = C(a', b', H') \), which completes the proof.

We proceed to show that for each \( a \in L \), \( O(n) \) set operations are sufficient for computing \( \Phi(U) \) for all \( U \subseteq \Gamma_a \). Recall that \( \Gamma_a = \{ C(a, b, H) : b \in L, H \in \text{Valid}(a, b) \} \). Let \( \text{size}(\Gamma_a) \) be the total size of the sets in \( \Gamma_a \). According to (7), for each \( U \subseteq \Gamma_a \), the number of set operations required for computing \( \Phi(U) \) is \( |U| - 1 \). Therefore, the number of set operations required for computing \( \Phi(U) \) for all \( U \subseteq \Gamma_a \) is less than \( \text{size}(\Gamma_a) \).

The following lemma gives an upper bound on \( \text{size}(\Gamma_a) \).

**Lemma 5.9.** For any \( a \in L \), \( \text{size}(\Gamma_a) \leq n - 1 \).

**Proof.** Clearly, \( a \not\in C(a, b, H) \) for any \( b \in L \) and \( H \in \text{Valid}(a, b) \). By Lemma 5.8, any two sets in \( \Gamma_a \) are disjoint. Therefore, \( \text{size}(\Gamma_a) \leq |L| - |a| \leq n - 1 \) and the lemma holds.

According to (4), (6), (7), and Lemma 5.9, a more efficient algorithm for computing \( \mathcal{K}(a, b) \) is given as follows.

**Procedure NEWKAST-1(a, b)**

**input:** \( (a, b) \in L^2 \)

**output:** \( \mathcal{K}(a, b) \)

**begin**
1. if \( a = b \) then return \( \{a\} \)
2. \( \mathcal{K}(a) = \bigcap_{x \in \text{Valid}(a, b)} \mathcal{K}(a, x) \)
3. \( \mathcal{K}(b) = \bigcap_{x \in \text{Valid}(a, b)} \mathcal{K}(b, x) \)
4. for each c \( \in C(a, b) \) do
   5. \( \mathcal{K}(c) = \bigcap_{x \in \text{Valid}(a, b)} \mathcal{K}(c, x) \)
6. find \( \text{Valid}(a, b) \) and compute \( C(a, b, H) \) for each subtree \( H \in \text{Valid}(a, b) \)
7. for each \( H \in \text{Valid}(a, b) \) do
   8. \( \Phi(H) \leftarrow \bigcap_{c \in \text{Valid}(a, b)} \mathcal{K}(c) \), where \( U = C(a, b, H) \)
   9. \( \Phi(H) \leftarrow \bigcup_{c \in \text{Valid}(a, b)} \Phi(H) \)
   10. \( \Phi(H) \leftarrow \mathcal{K}(a) \bigcup \Phi(H) \)
   11. return \( \mathcal{K}(a, b) \)

**end**

The detailed implementation of NEWKAST-1 is described as follows. Without loss of generality, assume that the maximum degree of \( T_a \) is \( d \). The following lemma gives the time complexity of Line 6.

**Lemma 5.10.** Given a pair \( (a, b) \in L^2 \), we can find \( \text{Valid}(a, b) \) and compute \( C(a, b, H) \) for each \( H \in \text{Valid}(a, b) \) in \( O(n^{d-2}) \) time.

**Proof.** If \( d = 2 \), since \( \text{Valid}(a, b) \) is empty, the lemma holds trivially. Assume that \( d \geq 3 \). We prove this lemma by presenting an algorithm. Initially, all subtrees of \( \alpha_i \) and all leaves in \( T_i(\alpha_i) \) are unmarked. This initialization takes \( O(n) \) time. First,
for each clique $S \in S'(a, b)$, we mark all the leaves in $S$ and mark all the subtrees of $\alpha$, that are not used by $S$. Since the degree of $\alpha$, and the size of a clique are both bounded by $d$, this step requires $O(d \times |S'(a, b)|) = O(n^{d^2})$ time. Next, for each subtree $H$ of $\alpha$, we compute $C(a, b, H)$ as the set of marked leaves in $H$. This step requires $O(|L(T(\alpha))|) = O(n)$ time.

To find the set $Valid(a, b)$, we need to determine whether each subtree of $\alpha$ is valid for $(a, b)$. All marked subtrees are not valid, since they are not used by all the cliques in $S'(a, b)$. Consider an unmarked subtree $H$. By definition, $H$ is valid if all leaves in $C(a, b, H)$ are coherent with respect to $(a, b)$. Clearly, two leaves $c, c'$ in $H$ are coherent with respect to $(a, b)$ if and only if $cc' \in \mathcal{R}$. Therefore, we determine whether $H$ is valid as follows: pick an arbitrary leaf $c \in C(a, b, H)$ and then check whether $cc' \in \mathcal{R}$ for all $c' \in C(a, b, H) - \{c\}$. The above checking requires $O(|C(a, b, H)|)$ time. Consequently, the set $Valid(a, b)$ can be found in $O(\sum_{H} |C(a, b, H)|) = O(|L(T(\alpha))|) = O(n)$ time.

The above algorithm requires $O(n^{d^2})$ time. Therefore, the lemma holds.

We proceed to describe the implementation of Lines 7 and 8. By Lemma 5.9, $O(n)$ set operations are sufficient for computing $\Phi_\alpha(U)$ for all $U \in \mathcal{U}_\alpha$. To avoid re-computation, for each leaf $a \in \mathcal{L}$, we maintain an array, TABLE$_a$, of $n$ entries to save the results of computed $\Phi_\alpha(U)$ and implement the finding of $\Phi_\alpha(U)$ in Line 8 as follows: first check to see whether $\Phi_\alpha(U)$ has been already computed for $U = C(a, b, H)$; if so, use the result stored in TABLE$_a$; if not, compute $\Phi_\alpha(U)$ and then store the result in TABLE$_a$. We need to establish a relationship between the positions of TABLE$_a$ and the sets in $\mathcal{U}_\alpha$. By Lemma 5.8, any two sets in $\mathcal{U}_\alpha$ are disjoint. Thus, each set in $\mathcal{U}_\alpha$ can be uniquely identified by any of its elements. We let each set in $\mathcal{U}_\alpha$ be represented by its smallest element. More specifically, when $\Phi_\alpha(U)$ has been computed for a set $U$, the result is stored in TABLE$_a$[$\min(U)$], where $\min(U)$ is the smallest element in $U$. Accordingly, the corresponding position of a given set $U$ in TABLE$_a$ can be found in $O(|U|) = O(n)$ time.

For a fixed $a \in \mathcal{L}$, Lines 7-8 of NEWKAST-1 over all $b \in \mathcal{L}$ require $O(n)$ set operations. Therefore, the required set operations of Lines 7-8 over all $(a, b) \in \mathcal{L}^2$ is $O(n^2)$. Consequently, the total time of Lines 7-8 over all $(a, b) \in \mathcal{L}^2$ is $O(n^2)$.

By Lemma 5.10, the total time of Line 6 over all $(a, b) \in \mathcal{L}^2$ is $O(n^2)$. As discussed in Section 4, the total time of Lines 1-5 and Line 10 over all $(a, b) \in \mathcal{L}^2$ is $O(n^2)$. Therefore, we obtain the following:

**Theorem 5.1.** The KAST problem on a set of $k$ trees can be solved in $O(kn^3 + n^3 + n^4)$ time, where $n$ is the size of the trees and at least one tree has maximum degree $d$.

**6. An $O(kn^3 + n^4)$-time KAST algorithm**

In this section, we show how to further improve the upper bound of the KAST problem to $O(kn^3 + n^4)$.

To solve the KAST problem, the algorithm in Section 5 computes $\mathcal{K}(a, b)$ for all $(a, b) \in \mathcal{L}^2$. At the time of writing, we are not aware of a more efficient way to compute all $\mathcal{K}(a, b)$ explicitly. In the time complexity of Theorem 5.1, the $n^4$ term comes from Lines 2-5 of NEWKAST-1, which computes $\mathcal{K}(a, b)$ for all $(a, b) \in \mathcal{L}^2$. Recall that this straightforward method for computing $\mathcal{K}(a, b)$ potentially requires a linear number of set intersections on subtrees $\mathcal{K}(a, x)$ such that $m(a, x) = m(a|b)$ and $x \in \mathcal{X}(a|b)$. Thus, for each pair in $\mathcal{L}^2$ we must take potentially $\Omega(n^2)$ time to perform all set intersections.

This section reduces the total number of set intersections necessary by working with what we call refinements of $\mathcal{K}(a, b)$ and $\mathcal{K}(a|b)$, denoted by $\mathcal{K}_a(a, b)$ and $\mathcal{K}_b(a|b)$, rather than $\mathcal{K}(a, b)$ and $\mathcal{K}(a|b)$ themselves. A refinement of a set is a subset that preserves all KAST leaves. We compute $\mathcal{K}_a(a, b)$ and $\mathcal{K}_b(a|b)$ only for the pairs $(a, b)$ that are relevant to the computation of the KAST of $\mathcal{T}$. The idea is to group computations for a fixed leaf $a$; all relevant pairs $(a, b)$ have equivalent $\mathcal{K}_b(a|b)$ if they have the same value for $m(a, b)$ and their subsolutions are all computed from the same intersection of sets $\mathcal{K}_a(a, x)$ such that $m(a, x) = m(a|b)$ and $(a, b)$ are relevant. The implication is that $\mathcal{K}_a(a, x)$ appears only in a single set intersection for a fixed $a$. This results in an algorithm that uses a linear number of total set operations to compute refinements of $\mathcal{K}(a|b)$ for a fixed $a$, rather than $\Omega(n^2)$. Consequently, the $n^4$ term in the time complexity of Theorem 5.1 is removed.

Since $\mathcal{K}_a(a, b)$ preserves all KAST leaves of $\mathcal{K}(a, b)$, the KAST of $\mathcal{T}$ is the intersection of all $\mathcal{K}_a(a, b)$ such that $m(a, b) = mast(T)$.

Section 6.1 shows how to identify all relevant pairs $(a, b)$. Section 6.2 describes the computation of $\mathcal{K}_a(a, b)$ and $\mathcal{K}_b(a|b)$ for each relevant pair.

### 6.1 Finding relevant pairs

Let $(a, b)$ be a pair in $\mathcal{L}^2$. We say that $(a, b)$ is relevant if there exists a MAST of $\mathcal{T}$ that contains a tree in $\mathcal{M}(a, b)$ as a subtree, and is irrelevant otherwise. That is, $(a, b)$ is relevant if $\mathcal{M}(a, b)$ contains a sub-solution to the problem of finding the MASTs of $\mathcal{T}$. Suppose that there is a MAST of $\mathcal{T}$ that contains a tree $X$ in $\mathcal{M}(a, b)$ as a subtree. If we modify the MAST by replacing $X$ with any other tree in $\mathcal{M}(a, b)$, it is easy to conclude from Lemma 3.1 that the resulting tree is still a MAST. Therefore, if $(a, b)$ is relevant, every tree in $\mathcal{M}(a, b)$ is a subtree of a MAST of $\mathcal{T}$.

As discussed in Section 3, given a clique $S \in S'(a, b)$, a tree in $\mathcal{M}(a, b)$ can be constructed by taking a tree in $\mathcal{M}(a|b)$, a tree in $\mathcal{M}(b|a)$, and a tree in $\mathcal{M}(c|a)$ for each $c \in S$. From this, and the definition of $\mathcal{M}(a|b)$ in (3), it can be concluded that if $(a, b)$ is a relevant pair, all pairs in the following set are relevant:

$$\Pi(a, b) = \{(a, x) : x \in \mathcal{X}(a|b)\} \cup \{(b, x) : x \in \mathcal{X}(b|a)\} \cup (\bigcup_{x \in \mathcal{X}(a|b)} (\bigcup_{c \in S} \{(c, x) : x \in \mathcal{X}(c|a)\}))$$

As a result, $m(a|b)$ and $m(a, b)$ have been computed for every $(a, b) \in \mathcal{L}^2$ by applying Bryant's dynamic programming algorithm, all relevant pairs can be recognized by tracing back the steps that led to each maximum table entry. More specifically, all relevant pairs can be recognized, in a top-down manner, according to the following rules:

(i) if $m(a, b) = mast(T)$, $(a, b)$ is relevant (since all trees in $\mathcal{M}(a, b)$ are MASTs); and

(ii) if $(a, b)$ is relevant, all pairs in $\Pi(a, b)$ are relevant.

Assume that $\mathcal{R}$, $\mathcal{F}$, and all $m(a|b)$ and $m(a, b)$, where $(a, b) \in \mathcal{L}^2$, have been computed. A procedure that finds all relevant pairs is given as follows.
Procedure FINDRELEVANT
/* find the set of relevant pairs */
begin
1. RELEVANT ← ∅ /* initialization */
2. for each (a, b) ∈ E2 do /* initialization */
3. compute X'(a|b) and mark X'(a|b) unexplored
4. for each (a, b) ∈ E2 do /* find relevant pairs recursively */
5. if m(a, b) = mast(T) then ADDRELEVANT(a, b)
6. return RELEVANT
end
Procedure ADDRELEVANT(a, b)
// add (a, b) and the pairs in Π(a, b) into RELEVANT
begin
1. RELEVANT ← RELEVANT ∪ {(a, b)}
2. if X'(a|b) is unexplored then EXPLORE(X'(a|b))
3. if X'(b|a) is unexplored then EXPLORE(X'(b|a))
4. for each c ∈ ΠS∩S(b, a) do
5. if X'(c|a) is unexplored then EXPLORE(X'(c|a))
end
Procedure EXPLORE(X'(p|q))
// the pairs in X'(p|q) into RELEVANT
begin
1. mark X'(p|q) as explored /* avoid re-exploring */
2. for each x ∈ X'(p|q) do
3. if (p, x) \in RELEVANT then ADDRELEVANT(p, x)
end

Lemma 6.1. FINDRELEVANT requires O(n^3 + n^3) time.
Proof. Each call to ADDRELEVANT takes O(\sum_{(a, b)} E2) time, excluding the time spent on the calls to EXPLORE. Each call to EXPLORE takes O(X'(p|q)) time, excluding the time spent on the calls to ADDRELEVANT. The number of calls to ADDRELEVANT is bounded by the number of relevant pairs, which is at most n^2. The number of calls to EXPLORE is bounded by the number of explored sets X'(p|q), which is also bounded n^2. Therefore, the total time of FINDRELEVANT is O(n^3 + n^3) and the lemma holds.

For ease of discussion, we call Φ(a, b) and Φ(b, a), respectively, the Φ-set and the conditional Φ-set of (a, b). According to (2), (4), and (5), Φ(a, b) is determined from the Φ-sets of the pairs in Π(a, b). Therefore, if (a, b) is relevant, all the Φ-sets required for computing Φ(a, b) are also of relevant pairs. As a result, it can be concluded that a set Φ(a, b) needs to be computed only if (a, b) is relevant. In the remainder of this section, we shall see that a set Φ(a, b) also needs to be computed only if (a, b) is relevant.

Lemma 6.2. Let a, b, c be three leaves such that (a, b, c) ∈ F and c is a leaf of a tree in Φ(a, b). If (a, b) is relevant, then (a, c) is relevant.
Proof. Assume that (a, b) is relevant. Let X be a tree in Φ(a, b) that contains the leaf c. Since (a, b) is relevant, there is a MAST, say Z, that contains X as a subtree. Since (a, b, c) ∈ F, tree X is an agreement subtree in which a, c are leaves and the root is the LCA of a, c. Thus, X ∈ Φ(a, c). All trees in Φ(a, c) are agreement subtrees of \{T(a): αi = lca(a, c), 1 ≤ i ≤ k\}. Therefore, after modifying Z by replacing X with any tree in Φ(a, c), the resulting tree is still an agreement subtree. Consequently, X is a largest tree in Φ(a, c); otherwise, Z is not a MAST. That is, X = Φ(a, c). Since X is a subtree of a MAST, (a, c) is relevant and the lemma holds.

According to (4) and (5), the following conditional Φ-sets are required for the computation of the Φ-set of a relevant pair (a, b):
Φ(a|b), Φ(b|a), and Φ(c|a) of each c ∈ ΠS∩S(b, a) S.
The sets Φ(a|b) and Φ(b|a) are of relevant pairs (a, b) and (b, a). Note that (a, b) is a relevant pair since M(a, b) = Φ(a, b). Recall that given a set S ∈ S(a, b), a tree in Φ(a, b) can be constructed by adding a tree in Φ(a, b), a tree in Φ(b|a), and a tree in Φ(c|a) for each c ∈ S. Therefore, each c ∈ ΠS∩S(b, a) S is a leaf of a tree in Φ(a, b). In addition, each c ∈ ΠS∩S(b, a) S is a leaf in C(a, b). As a result, it can be concluded from Lemma 6.2 that (c, a) is relevant for each c ∈ ΠS∩S(b, a) S. Based upon the above discussion, we know that Φ(a|b) also needs to be computed only if (a, b) is relevant.

6.2 Computing refinements
In this section, we shall show that Φ(a, b), which is a refinement of Φ(a, b), can be computed for every relevant pair (a, b) in O(\sum_{(a, b)} E2). Recall that the computation of Φ(a, b) is based on computing Φ(a(b)) = \bigcap_{\forall X^*} (Φ(a, x), where X^* = \{x \in Φ(a|b), m(a, x) = m(a|b)\}). Similarly, for each relevant pair (a, b), the computation of Φ(a, b) is based on an analogue of Φ(a|b), called Φ(a|c). The key observation will be described by Lemma 6.7, which says that, for any two relevant pairs (a, b) and (a, b'), Φ(a|b) and Φ(a|b') are equal if m(a, b) = m(a, b'). Thus, for a fixed a we group computations of Φ(a|b) according to the value of m(a|b) so that we only need to compute Φ(a|b) once for all b with the same m(a|b). As with Φ(a|b), Φ(a|c) depends only on the relevant pairs (a, x) with the same m(a, x). Therefore, each subbottom Φ(a, x) for relevant pair (a, x) is implicated in a set operation for exactly one group. Thus, all Φ(a|b) can be computed in quadratic time for a fixed a, yielding an algorithm that computes Φ(a|b) for all relevant pairs in O(\sum_{(a, b)} E2).

A refinement of a set U ⊆ L is any set obtained by deleting from U none or some leaves that are not in KAST(T). For example, if KAST(T) = {a, c, d} and U = {a, b, d, f}, then [a, d], [a, b, c, d], [a, b, d], [a, d, f], and [a, b, d, f] are all refinements of U; but [a, b, f] is not, since it does not contain the KAST leaf d. Recall that KAST(T) is the intersection of all Φ(a, b) with m(a, b) = mast(T). Since a refinement of a Φ-set preserves all the KAST leaves in the Φ-set, in the above computation of the intersection, each Φ-set can be replaced by any of its refinement. Therefore, to solve the KAST problem, it suffices to find a refinement of Φ(a, b) for each relevant pair (a, b).

By the definition of a refinement, it is easy to obtain the following.
Observation 6.3. Let U1, U1', U2, U2', ..., Up, U_p' be sets such that for each i, 1 ≤ i ≤ p, U_i^' is a refinement of U_i. Then, \bigcup U_1^' \cup U_2^' \cup ... \cup U_p^' is a refinement of \bigcup U_1 \cup U_2 \cup ... \cup U_p; and U_1 \cap U_2 \cap ... \cap U_p^' is a refinement of \bigcap U_1 \cap U_2 \cap ... \cap U_p.

Lemma 6.4. Let (a, b) be a relevant pair and Q be any tree in Φ(a, b). Then, for each T_i ∈ T, a leaf l of T(a_i) is in KAST(T) only if Q contains l, where a_i = lca(a, b).
Proof. Let T be a tree in T and l be a leaf of T(l(a)). Since (a, b) is relevant, there is a MAST, say Z, that contains Q as a subtree. By definition, the leaf l is in KAST(T) only if it is contained in every MAST of T. Assume that l \notin L(Q). In the following, we prove this lemma by showing that Z does not contain l. Suppose by contradiction that Z contains l. From
Lemma 3.1, it is easy to conclude that any subset of an agreement set is also an agreement set. Thus, \( L(Q) \cup \{l \} \) is an agreement set and \( T_{\{l\}}(Q) \cup \{l \} \) is an agreement subtree. Since \( l \) is a leaf of \( T(Q) \) and \( Q \in A(a, b) \), we know that in \( T_{\{l\}}(Q) \cup \{l \} \), \( a \) and \( b \) are leaves and the LCA of \( a \) and \( b \) is the root. Therefore, \( T_{\{l\}}(Q) \cup \{l \} \) is in \( A(a, b) \), contradicting to that \( Q \) is a largest tree in \( A(a, b) \). Thus, the lemma holds.

For ease of description, we define a valid filter of a set \( U \subseteq L \) to be a set containing all the KAST leaves in \( U \). For example, if \( KAST(T) = \{a, c, d\} \) and \( U = \{a, b, d, f\} \), then \( \{a, d\} \), \( \{a, c, d\} \) are all valid filters of \( U \); but \( \{a, c, f\} \) is not, since \( \{a, c, f\} \) does not contain the KAST leaf \( d \). Note that a refinement of \( U \) is also a valid filter of \( U \), but the reverse may not be true. For example, \( \{a, c, d\} \) is a valid filter of \( U \), but it is not a refinement of \( U \), since it is not a subset of \( U \). A valid filter of a set \( U \) can be used to remove from \( U \) some leaf that are not in \( KAST(T) \). More specifically, if \( U' \) is a valid filter of \( U \), then \( U' \cap U \) is a refinement of \( U \).

**Theorem 5.1** Let \((a, b)\) be a relevent pair. Then, for each \( T \in T \), \( K(a, b) \) is a valid filter of any subset of \( L(T, \alpha) \), where \( \alpha = lca(a, b) \).

**Proof.** We first show that \( K(a, b) \) is a valid filter of \( L(T, \alpha) \). Let \( l \) be any leaf of \( T(\alpha) \). By Lemma 6.4, \( l \) is in \( KAST(T) \) only if \( T(\alpha) \) is in \( K(a, b) \). Therefore, \( K(a, b) \) is a valid filter of \( L(T, \alpha) \). Clearly, a valid filter of a set \( U \) is a valid filter of any subset of \( U \). Thus, the lemma holds.

For each relevent pair \((a, b)\), define a variant of \( X(\alpha) \):

\[ X(\alpha) = \{ x : (a, x) \text{ is relevant}, m(a, x) = m(b) \} \]

We have the following.

**Lemma 6.6** For any relevent pair \((a, b)\), \( X(\alpha) \subseteq X(\alpha') \).

**Proof.** Let \( l \) be any leaf in \( X(\alpha') \). Recall that \( X(\alpha') = \{ x : x \in X(\alpha), m(a, x) = m(b) \} \). Since \( x \in X(\alpha') \), we have \( m(a, x) = m(b) \). In addition, since \( (a, x) \) is relevant and \( (a, x) \in X(\alpha') \), we know that \( (a, x) \) is relevant. Therefore, it can be concluded that each \( x \in X(\alpha') \) is contained in \( X(\alpha) \). Thus, the lemma holds.

\[ T_1 \]
\[ \alpha_1 = \alpha' \]
\[ a \]
\[ z \]
\[ b \]
\[ X(\alpha) \]
\[ \alpha_1' \]
\[ L(T_1(\alpha_1)) \]
\[ L(T_1(\alpha_1')) \]

Case 1. \( \alpha_1 = \alpha_1' \)

**Figure 6.** A leaf \( z \in X(\alpha) - X(\alpha') \) such that \( azb \) is not a rooted triple in \( T_1 \).

Before presenting an algorithm that computes a refinement for the \( K \)-set of each relevent pair, we describe the idea behind our approach. Consider the computation of \( K(a, b) \) for a relevent pair \((a, b)\). By definition, \( K(a, b) = \bigcap_{x \in X(\alpha)} K(a, x) \) to take the place of \( K(a, b) \). Let \( Z = X(\alpha) - X(\alpha') \). Since \( X(\alpha) \subseteq X(\alpha) \), this set, \( \bigcap_{x \in X(\alpha)} K(a, x) \), can be considered as the set obtained from \( K(a, b) \) by removing leaves not in \( K(a, z) \) for every \( z \in Z \). Consider a leaf \( z \in Z \). Since \( m(a, z) = m(b) \), we know that \( azb \) is \( R \); otherwise \( z \notin X(\alpha') \). That is, \( azb \) is not a rooted triple in at least one of the trees in \( T \). Without loss of generality, assume that \( azb \) is not a rooted triple in \( T_1 \). Since \( azb \) is not a rooted triple in \( T_1 \), we know that \( a = lca(a, b) \) is a descendant of \( a' = lca(a, z) \). There are two cases: \( a = a' \), and \( a \) is a proper descendant of \( a' \) (See Figure 6.) In either case, we have \( L(T_1(\alpha')) \subseteq L(T_1(\alpha)) \). Since \( K(a, b) \) is a subset of \( L(T_1(\alpha)) \) and thus is a subset of \( L(T_1(\alpha')) \). Hence, by Lemma 6.5, \( K(a, z) \) is a valid filter of \( K(a, b) \). Therefore, \( \bigcap_{x \in X(\alpha)} K(a, x) \) preserves all KAST leaves in \( K(a, b) \) and thus is a refinement of \( K(a, b) \).

Based upon the above idea, by replacing \( X(\alpha) \) with \( X(\alpha') \) in the definition of \( K(a, b) \), we define \( K(a, b) \) as follows:

\[ K(a, b) = K(a, b) \cup K(b, a) \cup \bigcap_{x \in X(\alpha)} K(a, x) \]

where \( K(a, b) = \bigcap_{x \in X(\alpha)} K(a, x) \).

Our algorithm uses \( K(a, b) \) to replace \( K(a, b) \). As mentioned, essentially, our idea is to compute \( \bigcap_{x \in X(\alpha)} K(a, x) \) to take the place of \( K(a, b) \). We remark that according to the definition in (9), \( K(a, b) \) may not be the same as \( \bigcap_{x \in X(\alpha)} K(a, x) \).

Before showing that \( K(a, b) \) is a refinement of \( K(a, b) \), we describe the advantage of replacing \( X(\alpha) \) with \( X(\alpha') \) in the definition of \( K(a, b) \).

**Lemma 6.7** Let \( a, b \), and \( b' \) be leaves such that \( (a, b) \) and \( (a, b') \) are both relevent, and \( m(a, b) = m(b, b') \). Then, \( K(a, b) = K(a, b') \) and the lemma holds.

**Proof.** Since \( m(a, b) = m(b, b') \), according to the definition of \( X_1(\cdot, \cdot) \), we have \( X_1(a, b) = X_1(a, b') \). Thus, \( K(a, b) = K(a, b') \) and the lemma holds.

In the time complexity of Theorem 5.1, the \( n^4 \) term comes from Lines 2-5 of NEWKAST-1, which computes \( K(a, b) \) for all \( (a, b) \in L^2 \). Lemma 6.7 indicates that for a fixed \( a \in L \) and size \( t < \text{mast}(T) \), \( K(a, b) \) only needs to be computed once for all \( b \) with \( m(a, b) = t \). Later, we will show that this property allows \( K(a, b) \) to be computed for all relevent pairs \((a, b)\) in \( O(n^2) \) time.

We proceed by showing that, for each relevent pair \((a, b)\), \( K(a, b) \) is a refinement of \( K(a, b) \). The proof is done by induction on the size \( m(a, b) \). The following lemma is needed.

**Lemma 6.8** Let \((a, b)\) be a relevent pair. Suppose that \( K(a, x) \) is a refinement of \( K(a, x) \) for each \( x \in X(\alpha) \). Then, \( K(a, b) \) is a refinement of \( K(a, b) \).

**Proof.** Recall that, previous to Lemma 6.7, we explained that \( \bigcap_{x \in X(\alpha)} K(a, x) \) is a refinement of \( K(a, b) \). The proof of this lemma is similar. By Lemma 6.6, \( X(\alpha) \subseteq X(\alpha') \). Thus, we can write \( K(a, b) = U_1 \cap U_2 \), where

\[ U_1 = \bigcap_{x \in X(\alpha)} K(a, x) \] and \( U_2 = \bigcap_{x \in X(\alpha)} K(a, x) \)

Since \( K(a, x) \) is a refinement of \( K(a, x) \) for each \( x \in X(\alpha) \), by Observation 6.3, \( \bigcap_{x \in X(\alpha)} K(a, x) \) is a refinement of \( \bigcap_{x \in X(\alpha)} K(a, x) \). That is, \( U_1 \) is a refinement of \( K(a, b) \).

In the following, we show that \( U_2 \) is a valid filter of \( K(a, b) \).

Consider a fixed \( z \in X(\alpha) \). From \( m(a, z) = m(b, z) \), we know that \( azb \) is not a rooted triple in at least one of the trees in \( T \). Without loss of generality,
assume that \( ax | b \) is not a rooted triple in \( T_r \). Since \( ax | b \) is not a rooted triple in \( T_r \), we know that \( a \alpha = \text{lca}_a(a, z) \) is a descendant of \( a' \alpha = \text{lca}_a(a, z) \). (See Figure 6.) Since \( K(a(b)) \subseteq L(T_r(a)) \subseteq L(T_r(\alpha)) \), by Lemma 6.5, \( K(a, z) \) is a valid filter of \( K(a(b)) \). In addition, since \( K(a, z) \) is a refinement of \( K(a, z) \), it is easy to see that \( K(a, z) \) is also a valid filter of \( K(a)(b) \). Clearly, the intersection of valid filters is also a valid filter. Therefore, \( U \) is a valid filter of \( K(a)(b) \).

In summary, \( U_1 \) is a refinement of \( K(a)(b) \) and \( U_2 \) is a valid filter of \( K(a)(b) \). From this, it is easy to conclude that \( U_1 \cap U_2 \) is a refinement of \( K(a)(b) \), which completes the proof of this lemma.

Our new algorithm is designed based upon the following.

**Theorem 6.1.** For any relevant pair \((a, b), K_i(a, b)\) is a refinement of \( K_i(a, b) \).

**Proof.** We prove the theorem by induction on the size \( m(a, b) \).

First, consider the case \( m(a, b) = 1 \). In this case, we have \( a = b \); otherwise, \( |m(a, b)| \geq |\{a, b\}| \geq 2 \). Since \( a = b \), by definition, we have \( K_i(a, b) = K_i(a, b) = \{a\} \) and thus the theorem holds.

Consider the case \( m(a, b) > 1 \). Suppose, by induction, that this theorem is true for all \( K_i(a', b') \) such that \( a', b' \) is relevant and \( m(a', b') < m(a, b) \); we will show that this theorem holds for \( K_i(a, b) \) as well. Recall that

\[
K_i(a, b) = K_i(a(b)) \cup K_i(a(b)) \\
\left(\bigcap_{s \subseteq S^*} (a, b) \right), \text{ and } K_i(a, b) = K_i(a(b)) \cup K_i(a(b)) \\
\left(\bigcap_{s \subseteq S^*} (a, b) \right).
\]

According to Observation 6.3, to prove that \( K_i(a, b) \) is a refinement of \( K_i(a, b) \), it suffices to show the following:

(i) \( K_i(a(b)) \) is a refinement of \( K_i(a(b)) \);

(ii) \( K_i(a(b)) \) is a refinement of \( K_i(a(b)) \); and

(iii) \( K_i(c(a)) \) is a refinement of \( K_i(c(a)) \) for any \( c \in S \) and \( S \subseteq S^*(a, b) \).

In the following, only the proof of (ii) is given. The proofs of (ii) and (iii) are similar. Consider an \( x \in X_\ast(a,b) \). Note that according to the definition of \( X_\ast(a,b) \), \( (a, x) \) is relevant and \( m(a, x) = m(b, x) \). According to (1), we have \( m(a, x) < m(a, b) \) and thus \( m(a, x) = m(b, x) \). Consequently, by the induction hypothesis, \( K_i(a, x) \) is a refinement of \( K_i(a, x) \) for any \( x \in X_\ast(a,b) \). As a result, by Lemma 6.8, \( K_i(a)(b) \) is a refinement of \( K_i(a)(b) \), which completes the proof of this theorem.

Due to the similarity between the recurrences of \( K_i(a, b) \) and \( K_i(a, b) \), \( K_i(a, b) \) and \( K_i(a, b) \) can be computed by slightly modified NEWKAST-1 as follows.

**Procedure NEWKAST-2(a, b)***

**input:** \((a, b) \in \mathbb{E}^2\) is a relevant pair

**output:** \(K_i(a, b)\)

**begin**

1. if \( a = b \) then return \( (a) \)

2. \( K_i(a(b)) \leftarrow \bigcap_{x \subseteq X_\ast(a,b)} K_i(a, x) \) /* (a, b) is relevant

3. \( K_i(a(b)) \leftarrow \bigcap_{x \subseteq X_\ast(a,b)} K_i(b, x) \) /* (b, a) is relevant

4. for each \( c \in (a, b) \) do

5. if \((c, a)\) is relevant then \( K_i(c(a)) \leftarrow \bigcap_{x \subseteq X_\ast(c,a)} K_i(c, x) \)

6. find \( \text{Valid}(a, b) \) and compute \( C(a, b, H) \) for each subtree \( H \in \text{Valid}(a, b) \)

7. for each \( H \in \text{Valid}(a, b) \) do

8. \( \Phi(a) \leftarrow \bigcap_{x \subseteq X_\ast(a,b)} K_i(a(c), \text{ where } U = C(a(b, H)) \)

9. \( \Phi(a, b) \leftarrow \bigcup_{H \in \text{Valid}(a, b)} \Phi(a, b, H) \)

**end**

To solve the KAST problem, procedure NEWKAST-2 is called for every relevant pair \((a, b)\), in non-decreasing order of \( m(a, b) \). The overall time complexity is analyzed as follows. As shown in Section 5, Lines 1 and 6-10 of NEWKAST-2 can be implemented such that their execution requires \( O(n^2) \) time over all relevant pairs \((a, b)\). In the following, we show that Lines 2-5 of NEWKAST-2 can be implemented such that their execution takes \( O(n^2) \) time over all relevant pairs \((a, b)\).

The purpose of Lines 2-5 is to compute \( K_i(a)(b) \) for all relevant pairs \((a, b)\). According to Lemma 6.7, for any two relevant pairs \((a, b)\) and \((a', b')\), \( K_i(a)(b) \) and \( K_i(a')(b') \) are the same if \( m(a, b) = m(a', b') \). For each \( a \in L \) and positive integer \( t < \text{mast}(T) \), define

\[ K_0^{(a)} = \bigcap_{x \subseteq X_\ast(a, x)} \text{relevant}, m(a, x) = t \] \( K_i(a, x) \).

Then, \( K_i(a)(b) = K_i^{(m(a)(b))} \) for each relevant pair \((a, b)\), Thus, our problem is to compute \( K_0^{(a)} \) for all \( a \in L \) and all \( t \in \{1, \text{mast}(T)\} \).

Consider the computation of \( K_0^{(a)} \) for a fixed \( a \in L \) and all \( t \in \{1, \text{mast}(T)\} \). Let \( \lambda(t) \) be the number of relevant pairs \((a, x)\) with \( m(a, x) = t \). For each \( t \in \{1, \text{mast}(T)\} \), the computation of \( K_0^{(a)} \) requires \( \lambda(t) - 1 \) set operations. Since \( \sum \lambda(t) = n \), we know that \( O(n) \) set operations are sufficient for computing \( K_0^{(a)} \) for a fixed \( a \in L \) and all \( t \in \{1, \text{mast}(T)\} \). As a result, computing \( K_0^{(a)} \) for all \( a \in L \) and all \( t \in \{1, \text{mast}(T)\} \) requires \( O(n^2) \) set operations. Consequently, the execution time of Lines 2-5 of NEWKAST-2 over all relevant pairs \((a, b)\) is \( O(n^2) \).

In summary, we obtain the following.

**Theorem 6.2.** The KAST problem on a set of \( k \) trees can be solved in \( O(kn^2) \) time, where \( n \) is the size of the trees and at least one tree has maximum degree \( d \).

For binary trees, the following is obtained.

**Corollary 6.1.** The KAST problem on a set of \( k \) binary trees can be solved in \( O(kn^3) \) time, where \( n \) is the size of the trees.

7. Experiments

We tested our C implementation of Bryant’s MAST algorithm (FINDMAST), the original KAST algorithm (FINDKAST), and our new KAST algorithm (NEWKAST-2). The Swenson et al. [45] paper already conducted a thorough study on the utility of the KAST, therefore, the focus of our new experiments was on running time. We timed our implementations on both simulated data and on real data where we compared independently inferred binary trees. Our results on rRNA alignments show that comparison of even the largest trees can be performed on a modern laptop in one work day. Our simulated datasets show an average 43 fold speedup on trees with 2000 taxa. The low KAST values between different Maximum Likelihood trees highlight the importance of the KAST for reporting high-confidence subtrees.

7.1 Running Time on RNA Alignments

Hand-curated RNA alignments at the Comparative RNA Web (CRW) site [10], a Living Tree Project (LTP) alignment
NEWKAST-2 is always 2.7 times slower than FINDMAST. e.g. when there are few rooted trees (binary birth-death tree).

NEWKAST-2 is clear: for FINDKAST, the dataset with 5,088 leaves takes more than an hour and many gigabytes of RAM, while taking hours and a week of computation (on a server with hundreds of gigabytes of RAM). All other datasets failed to terminate after 2 weeks. In contrast, NEWKAST-2 uses only 15-20 minutes and about 1 gigabyte of RAM. Alignments with up to 13,000 taxa were downloaded and maximum likelihood (ML) phylogenies were reconstructed using RAxML [42].

Sets of most likely RAxML trees were reconstructed and then compared for similarity using our algorithms.

Alignments from [4], were used to test the advantages and the limits of our algorithm. Alignments with up to 13,000 taxa were downloaded and maximum likelihood (ML) phylogenies were reconstructed using RAxML [42].

Both FINDKAST and NEWKAST-2 have FINDMAST at their core, therefore we report times as a proportion of the FINDMast compute time. The advantage of our new algorithm is clear when there are few rooted trees (e.g. k = 2). NEWKAST-2 is always 2.7 times slower than FINDMAST while FINDKAST is already 10 times slower at n = 100 and more than 100 times slower at n = 1600. (Figure 7(a).)

At n = 2000, NEWKAST-2 takes under seven minutes while FINDKAST takes about five hours.

When the number of trees is larger (e.g. k = 100), NEWKAST-2 is three times faster (taking 2.1 hours) than FINDKAST (taking 6.1 hours) on trees with 2000 leaves (Figure 7(b)).

Although the computation of the unrooted KAST has yet to be directly studied in the literature, we can compute it by running the rooted version n times, each time rooting the input at a leaf that is assumed to be part of the solution. For the unrooted case, our implementations of NEWKAST-2 and FINDKAST scale similarly to the rooted case, where a tree on 900 leaves takes over 71 hours for FINDKAST but under four hours for NEWKAST-2. (Figure 7(c).)

### 8. Concluding remarks

The MAST algorithms in [7], [19] work for any k and d. For some special cases, more efficient solutions exist. For example, there is an O(n lg n) time algorithm for k = d = 2 [13]; there is an O(n lg n) time algorithm for k = 2 and constant d [28]; and there is an O(n^1.5) time algorithm for k = 2 and arbitrary d [28]. Clearly, a leaf is a KAST leaf of T if and only if its removal from each tree of T reduces the size of MASTs by one. That is, whether a leaf l ∈ L is a KAST leaf can be determined by performing a MAST algorithm to check whether mast(T) < mast(T \ l). Here T is the set obtained by removing l from each tree in T. As a result, the set of KAST leaves can be identified by simply running a MAST algorithm n times. By applying this simple approach to the algorithms in [13], [28], the KAST problem is solved in O(n^2 lg n) time for k = d = 2, in O(n^2 lg n) time for k = 2 and constant d, and
in $O(n^{1.5})$ time for $k = 2$ and arbitrary $d$. These simple results are better than the time complexity in Theorem 6.2. One direction for further study is to design more efficient KAST algorithms for these special cases. To beat our KAST algorithm for arbitrary $k$ and $d$, however, further innovation may have to be made for the general MAST problem.

**ACKNOWLEDGMENTS**

This research is supported by the National Science Council of the Republic of China under grant MOST-106-2221-E-007-022-MY3.

**REFERENCES**


