Computing the Quartet Distance between Evolutionary Trees in Time $O(n \log n)$*

Gerth Stølting Brodal[†,‡] Rolf Fagerberg[†] Christian N. S. Pedersen[†]

March 3, 2003

Abstract

Evolutionary trees describing the relationship for a set of species are central in evolutionary biology, and quantifying differences between evolutionary trees is therefore an important task. The quartet distance is a distance measure between trees previously proposed by Estabrook, McMorris and Meacham. The quartet distance between two unrooted evolutionary trees is the number of quartet topology differences between the two trees, where a quartet topology is the topological subtree induced by four species. In this paper, we present an algorithm for computing the quartet distance between two unrooted evolutionary trees of n species, where all internal nodes have degree three, in time $O(n \log n)$. The previous best algorithm for the problem uses time $O(n^2)$.

Keywords: Evolutionary trees, distance measures, quartet distance, hierarchical decompositions.

*A preliminary version of this paper appeared in the proceedings of 12th International Symposium on Algorithms and Computation [3].

†BRICS (Basic Research in Computer Science, www.brics.dk, funded by the Danish National Research Foundation), Department of Computer Science, University of Aarhus, Ny Munkegade, DK-8000 Århus C, Denmark. E-mail: {gerth,rolf,cstorm}@brics.dk. Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

‡Supported by the Carlsberg Foundation (contract number ANS-0257/20).

1 Introduction

The evolutionary relationship for a set of species can be described by an evolutionary tree, which is a rooted tree where the leaves correspond to the species, and the internal nodes correspond to speciation events, i.e. the points in time where the evolution has diverged in different directions. The direction of the evolution is described by the location of the root, which corresponds to the most recent common ancestor for all the species, and the duration of evolutionary periods is described by assigning lengths to the edges. The true evolutionary tree for a set of species is most often unknown; estimating it from obtainable information about the species, e.g. genomic data, is of great interest in evolutionary biology. The problem of computationally estimating aspects of the true evolutionary tree requires a model describing how to use the available information about the species in question. Given a model, the problem of estimating aspects of the true evolutionary tree is referred to as constructing the evolutionary tree in that model. Many models and construction methods are available, see [11, Chapter 17] for an overview.

An important aspect of the true evolutionary tree for a set of species is its undirected tree topology induced by ignoring the location of the root and the length of the edges. Many models and methods are concerned with estimating just this tree topology, usually under the further assumption that all internal nodes have degree three. We say that such models and methods are concerned with constructing the unrooted evolutionary tree of degree three for a set of species. For the remainder of this paper, an evolutionary tree will denote an unrooted evolutionary tree of degree three.

Different models and methods often yield different estimates of the evolutionary tree for the same set of species, and even the same model and method can yield different evolutionary trees for the same set of species when applied to different information about the species, e.g. different genes. To study such differences in a systematic manner, one has to be able to quantify differences between evolutionary trees using well-defined and efficient methods.

One approach used for comparing two evolutionary trees is to define a distance measure between two trees and compare the two trees by computing the distance between them. Many distance measures have been proposed—among these are the symmetric difference metric [16], the nearest-neighbor interchange metric [19], the subtree transfer distance [1], the Robinson and Foulds metric [17], and the quartet metric [9]. Each distance measure has different properties and reflects different aspects of biology, e.g. the subtree transfer distance is related to the number of recombination events between the two sets of species. Bryant et al. in [6] argue that the quartet metric has several attractive properties and does not suffer from drawbacks of other distance measures, such as measures based on transformation operations (e.g. the subtree transfer distance) not distinguishing between transformations that affect a large number of leaves and transformations that affect a small number of leaves.

In this paper, we study the quartet metric. For an evolutionary tree for a set of n species, the *quartet topology* of four species is the topological subtree induced by these species. In general, there are four possible quartet topologies, as shown in Figure 1. However, if we assume that all internal nodes have degree three, then the right-most quartet topology cannot occur. It is well-known that the complete set of quartet topologies is unique for a given tree and that the tree can be recovered from its set of quartet topologies in polynomial time [7]. If the tree has degree three, then, as observed in [13], it can be recovered from its set of quartet topologies in time $O(n \log n)$ using methods for constructing an evolutionary tree in the experiment model [4, 10, 12, 13, 15].

Given two evolutionary trees on the same set of n species, the *quartet distance* between them is the number of sets of four species for which the quartet topologies differ in the two trees. Since

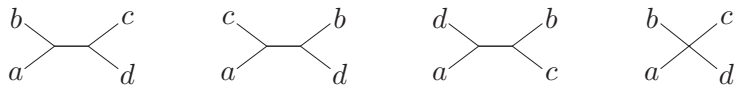

Figure 1: The four possible quartet topologies of species a, b, c, and d.

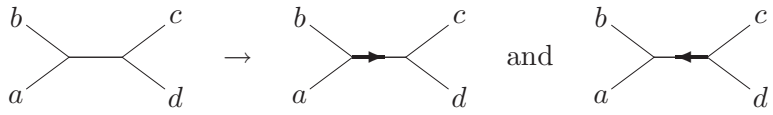

Figure 2: The two orientations of a quartet topology.

there are $\binom{n}{4}$ sets of four species, the quartet distance can be calculated in time $O(n^4)$ by examining the sets one by one. Steel and Penny in [18] presented an algorithm for computing the quartet distance in time $O(n^3)$. Bryant et al. in [6] presented an improved algorithm which computes the quartet distance in time $O(n^2)$.

In this paper, we present an algorithm which computes the quartet distance in time $O(n \log n)$. Our solution is based on a data structure related to the data structure for dynamic expression trees [8], the "extended smaller-half trick" [5], and the on-the-fly compression of the data structure to facilitate the use of the extended smaller-half trick.

The rest of the paper is organized as follows. In Section 2, we introduce quartets and our strategy for computing the quartet distance between two unrooted evolutionary trees. In Section 3, we describe and analyze a hierarchical decomposition of unrooted trees which is an essential part of the data structure used by our algorithm. In Section 4, we present the details of our data structure. In Section 5, we describe an algorithm with running time $O(n \log^2 n)$, which will serve as a basis for our final algorithm. In Section 6, we present our final algorithm with running time $O(n \log n)$.

2 Terminology

As mentioned, we in this paper by an *evolutionary tree* mean an unrooted tree where all nodes are either leaves (i.e. have degree one) or have degree three, and where the leaves are uniquely labeled by the elements of a set S of species. Let n denote the size of S. For an evolutionary tree T, the *quartet topology* of a set $\{a, b, c, d\} \subseteq S$ of four species is the topological subtree of T induced by these species. In general, the possible quartet topologies for species a, b, c, d are the four shown in Figure 1. Of these, the right-most does not occur in our setting, due to the assumption about all internal nodes having degree tree. Hence, the quartet topology is a pairing of the four species into two pairs, defined by letting a and b be a pair if among the three paths in T from a to respectively b, c, and d, the path to b is the first to separate from the others—more formally, the quartet topology is a two-set of two-sets $\{\{a, b\}, \{c, d\}\}$.

Given two evolutionary trees T_1 and T_2 on the same set S of species, the *quartet distance* between the two trees is the number of four-sets $\{a, b, c, d\} \subseteq S$, for which the quartet topologies in T_1 and T_2 differ. As there are $\binom{n}{4}$ different four-sets in S, the quartet distance can also be calculated as $\binom{n}{4}$ minus the number of four-sets for which the quartet topologies in T_1 and T_2 are identical. In this paper, we show how to find this number in time $O(n \log n)$.

To facilitate the counting of identical quartet topologies in the two trees, we view the quartet topology of a four-set $\{a, b, c, d\}$ as two *oriented* quartet topologies given by the two possible

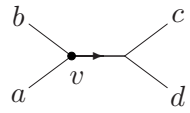

Figure 3: A generic quartet $ab \cdot cd$.

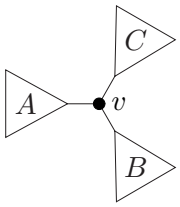

Figure 4: Subtrees incident to an internal node v.

orientations of the "middle edge" of the topology. Figure 2 shows the two oriented quartet topologies arising from one non-oriented quartet topology. More formally, an oriented quartet topology is an ordered pair of two-sets $(\{a, b\}, \{c, d\})$. Clearly, the number of identical oriented quartet topologies between the trees T_1 and T_2 is twice the number of identical non-oriented quartet topologies. The goal of our algorithm is to count identical oriented quartet topologies. For brevity, we in the rest of this paper let the word *quartet* denote an oriented quartet topology of a four-set, and use the notation $ab \cdot cd$ for $(\{a, b\}, \{c, d\})$. There are $2 \cdot 3 \cdot \binom{n}{4}$ possible quartets of S, of which any evolutionary tree T contains a subset Q_T of size $2 \cdot \binom{n}{4}$.

We *associate* the quartets in Q_{T_1} with internal nodes in T_1 as follows: Consider the generic quartet $ab \cdot cd$ in Figure 3. There is a unique node v in T_1 where the paths from a and b to c (and d) meet. We associate the quartet with the node v. This partitions Q_{T_1} into $n - 2$ disjoint sets, as there are $n - 2$ internal nodes in a tree of n leaves, when all internal nodes have degree three. For an internal node v in T_1, we by Q_v denote the set of quartets associated with v.

For an internal node v in T_1, we by the subtrees *incident* to v mean the three subtrees which arise if v and its three incident edges are removed from T_1. These are shown in Fig 4, denoted by A, B, and C. The number of quartets associated with v is given by the expression

$$\binom{|A|}{2} \cdot |B| \cdot |C| + \binom{|B|}{2} \cdot |A| \cdot |C| + \binom{|C|}{2} \cdot |A| \cdot |B| \,,$$

where $|T|$ denotes the number of leaves in subtree T. The three terms of the expression are the number of quartets where c and d (in Figure 3) are in respectively the subtree A, B, and C (in Figure 4).

Our strategy for computing the quartet distance between T_1 and T_2 is for each internal node v in T_1 to count how many of the quartets associated with v which are also quartets of T_2. The sum over all internal nodes in T_1 of these counts then gives the required number of identical quartets in T_1 and T_2. In other words, we use the fact that $|Q_{T_1} \cap Q_{T_2}| = \sum_{v \in T_1} |Q_v \cap Q_{T_2}|$.

To implement the above strategy, we construct an algorithm which colors the elements of S using the three colors $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$. We relate the coloring and the quartets to each other by the following two definitions: For an internal node v in T_1, we say that the elements of S are *colored*

according to v if the labels of the leaves of one of the three subtrees incident to v all have color $\mathcal{A}$, the labels of the leaves of another of the subtrees all have color $\mathcal{B}$, and the labels of the leaves of the remaining subtree all have color $\mathcal{C}$. For a coloring of the elements in S and a quartet $ab \cdot cd$, we say that the quartet is *compatible* with the coloring if a and b have different colors, and c and d both have the remaining color. From these definitions the lemma below is immediate.

Lemma 1 *When S is colored according to a choice of v in T_1, then the set of possible quartets of S that are compatible with the coloring is exactly the set Q_v of quartets associated with v.*

From Lemma 1, it follows that if the coloring of S is according to a choice of v in T_1, then the quartets in T_2 compatible with the coloring are exactly the quartets associated with v which are in both T_1 and T_2.

We maintain the coloring via a data structure described in Section 4. The central feature of the data structure is that it can in constant time return the number of quartets in T_2 compatible with the current coloring. The data structure also allows the color of k elements to be changed in time $O(k + k \log \frac{n}{k})$, given k pointers to the elements. For each node v in T_1 the algorithm will ensure a coloring according to v and then query the data structure to find the number of quartets associated with v that also are quartets of T_2.

3 Hierarchical Decomposition

An essential part of the data structure in Section 4 is a hierarchical decomposition of the evolutionary tree T_2. In the following, we describe how to obtain a well balanced hierarchical decomposition of any unrooted tree T with nodes of degree at most three. Our decomposition is related to the decompositions used for solving the parallel and dynamic expression tree evaluation problems [2, 8], but in our setting the underlying tree is unrooted.

The hierarchical decomposition is based on the notion of components. A *component* C in T is a connected subset of nodes in T. An *external edge* of C is an edge in T connecting nodes in C and $T \setminus C$, i.e. an edge crossing the cut defined by C. The *degree* of C is its number of external edges. We allow only the following two types of components:

1. Components containing a single node of T.
2. Components of degree at most two.

We let each node of T (including leaves) constitute a component of type 1. Components of type 2 are formed as the union of two adjacent components C' and C'', where C' and C'' are said to be adjacent if there exists an edge (u, v) in T such that $u \in C'$ and $v \in C''$. We call such a union a *composition*. Each composition of two components corresponds to a unique edge in the tree T, namely the edge connecting the two components.

We allow only the four compositions depicted in Figure 5, where nodes represent contracted components and ovals represent compositions. Types (i), (iii), and (iv) are the cases where a component with degree one is composed with a component of degree three, two, and one respectively. Type (ii) is the case where two components with degree two are composed into a new component with degree two. Note that these compositions will only produce components of degree at most two.

A *hierarchical decomposition* of T is a set of components produced during some sequence of compositions, starting from an initial set containing one type 1 component for each node in T. If

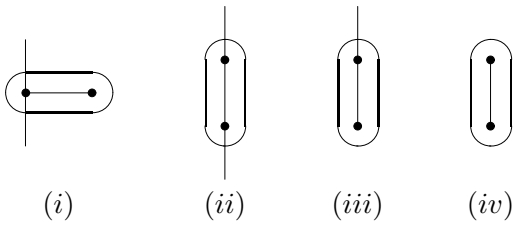

Figure 5: The four possible types of compositions of components.

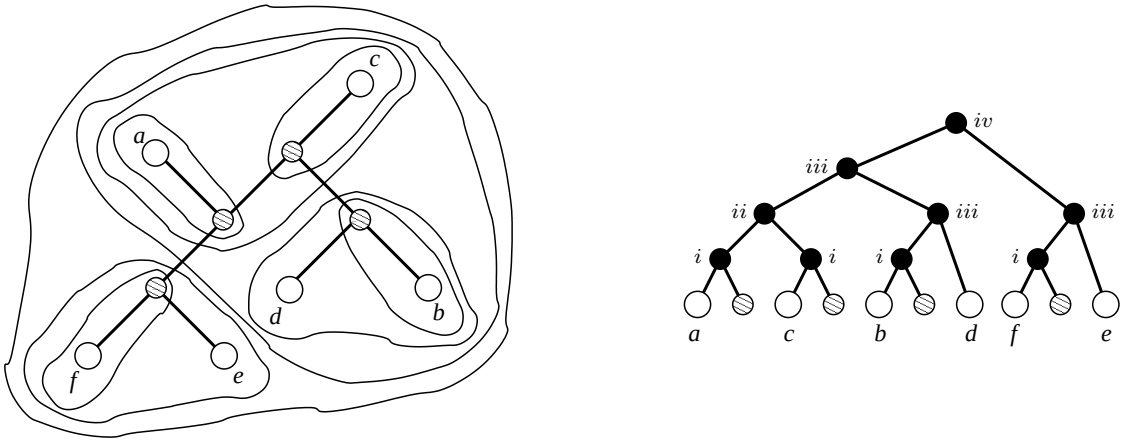

Figure 6: A hierarchical decomposition of a tree T with eight nodes and the corresponding hierarchical decomposition tree $H(T)$ with eight leaves. Each node in the hierarchical decomposition tree corresponds to a component in the hierarchical decomposition of the tree. The labels are the types of compositions used.

we contract components to single nodes after each composition, the compositions correspond to edge contractions. As new components have degree at most two, we in this contracted view always have a tree with nodes of degree at most three. We can therefore always apply a composition of type (i), (iii), or (iv), unless we already have a single component containing all of T.

For hierarchical decompositions that include a component containing all of T, we may in a natural way view the decomposition as a rooted binary tree $H(T)$, which we call a *hierarchical decomposition tree* for T. Each node of $H(T)$ represents a component in the decomposition. Leaves of $H(T)$ represent the components of type 1 in a one-to-one fashion, and an internal node v of $H(T)$ represents a component of type 2 formed by the composition of the two components represented by the children of v. Figure 6 shows a hierarchical decomposition of a tree T and the corresponding hierarchical decomposition tree $H(T)$.

We will show how to construct hierarchical decomposition trees which are locally-balanced. A rooted binary tree with n nodes is *c-locally-balanced* if for all nodes v in the tree, the height of the subtree rooted at v is at most $c \cdot (1 + \log |v|)$, where $|v|$ is the number of leaves in the subtree rooted at v and height is the maximal number of edges on any root-to-leaf path. For locally-balanced binary trees, Lemma 2 below holds. This property is essential for the use of the extended smaller-half trick in the final algorithm in Section 6.

Lemma 2 *The union of k root-to-leaf paths in a c-locally-balanced rooted binary tree with n leaves contains at most $k(3+4c)+2ck\log\frac{n}{k}$ nodes.*

Proof. Let T be a c-locally-balanced binary tree with n leaves, and $N(n,k,c)$ be the maximal number of *edges* in the union of k root-to-leaf paths in such a tree. We first give an upper bound on the number of edges which lead to exactly one of the k leaves. The edges constitute a set of k paths $P_1,\ldots,P_k$, such that each path starts at some internal node and leads to exactly one of the k leaves. If (u_i,v_i) is the first edge in a path P_i, then P_i is the only path containing edges from the subtree rooted at v_i, and we have $|P_i| \leq 1+h(v_i) \leq 1+c+c\log|v_i|$, where $h(v_i)$ and $|v_i|$ are the height and size of the subtree rooted at v_i, and $|P_i|$ is the number of edges in P_i. Since the subtrees rooted at $v_1,\ldots,v_k$ are disjoint, we have $|v_1|+\cdots+|v_k| \leq n$, so by the convexity of the logarithm we get the following bound on the number of edges leading to exactly one leaf.

$$\sum_{i=1}^{k}|P_i| \leq \sum_{i=1}^{k}(1+c+c\log|v_i|) = k+ck+c\sum_{i=1}^{k}\log|v_i| \leq k+ck+ck\log\frac{n}{k}\ .$$

The edges leading to at least two of the k leaves constitute a subtree T' of T with at most $\lfloor k/2\rfloor$ leaves, since a leaf of T' is an internal node v of T where both the edges to the children of v lead to exactly one of the k leaves. Hence, T' is contained in the union of at most $\lfloor k/2\rfloor$ of the paths $P_1,\ldots,P_k$, and we get the recurrence

$$N(n,k,c) \leq k+ck+ck\log\frac{n}{k}+N(n,\lfloor k/2\rfloor,c)\ ,$$

with $N(n,1,c)=c+c\log n$. Using the fact that $2x+x\log\frac{n}{x}$ is increasing in x for $0<x\leq n$, we by induction get $N(n,k,c)\leq 2k+4ck+2ck\log\frac{n}{k}$. As a path with t edges contains $t+1$ nodes, the union of the k paths contains at most $k(3+4c)+2ck\log\frac{n}{k}$ nodes. □

We now describe how to construct locally-balanced hierarchical decomposition trees in linear time.

Lemma 3 *For any unrooted tree T with n nodes of degree at most three, a $(1/\log\frac{18}{17})$-locally-balanced hierarchical decomposition tree $H(T)$ can be computed in time $O(n)$.*

Proof. Given an unrooted tree with n nodes, we construct a hierarchical decomposition tree bottom-up in $O(\log n)$ rounds. Each round performs a number of compositions of types (i)–(iv), and hence produces a new set of components which forms a tree with nodes of degree at most three. This tree is the basis for the next round. Before the first round, each node in T is a component by itself. In each round, we by a traversal of the tree greedily select an arbitrary maximal set of non-overlapping compositions, using time linear in the size of the tree, i.e. in the number of remaining components. Since one of the compositions (i), (iii), and (iv) can always be applied if there are at least two components left, this algorithm will eventually terminate with a single component representing the entire tree.

Let v be a node in the constructed hierarchical decomposition tree, and let m be the number of nodes of T in the component C represented by v. We will argue that the height of the subtree rooted at v is $O(\log m)$ by arguing that the construction of C has been done in $O(\log m)$ rounds. If $m=1$, the height is zero. If $m=2$, there are three cases: If C has no external edges then type (iv) has been applied, if C has one external edge then type (iii) has been applied, and if C has two external edges then types (i) or (ii) has been applied. In either case, the height is one.

So assume $m \geq 3$. Let t denote the number of external edges of C, and let m_1, m_2, and m_3 denote the number of nodes contained in C that are of degree one, two, and three in T. For a tree with nodes of degree at most three, the number of degree three nodes is exactly two less than the number of degree one nodes. In particular, this holds for a tree consisting of the nodes and edges inside C, plus the t external edges each terminated by a degree one node. This implies the relation $m_3 = m_1 + t - 2$. As $t \leq 2$, we have $m_3 \leq m_1$.

There are $m-1$ edges inside C, as C constitute a tree. Of these, the only edges not corresponding to legal compositions are edges connecting a node of degree three with a node of degree two or three. There are at most $3m_3$ such edges. The number of possible compositions is therefore at least $m - 1 - 3m_3 \geq m - 1 - 3m_1$. In the case $m_1 < m/6$, this bound is larger than $3m/6 - 1$. As $m \geq 3$, this is at least $m/6$. Otherwise, we have the case $m_1 \geq m/6$. Since $m \geq 3$, there are always m_1 possible compositions of types (i) and (iii). In both cases, there are at least $m/6$ possible compositions on edges in C. Since each possible composition can be in conflict with at most two other possible compositions (cf. Figure 5), any maximal set of non-conflicting compositions chosen in the first round of the construction algorithm contains at least $m/18$ edges in C.

The analysis above also applies to subsequent rounds, except that it should use values m'_1, m'_2, m'_3, and m' denoting the current number of components within C of degree one, two, and three, and their sum, respectively. The construction of component C starts with m components corresponding to single nodes. After k rounds, at most $m(17/18)^k$ components remain. In particular, one component remains after at most $\lceil \log_{18/17} m \rceil$ steps, so the height of the subtree rooted at v is bounded by $\lceil \log_{18/17} m \rceil \leq (1/\log \frac{18}{17})(1 + \log m)$.

Finally, consider the time it takes to construct the hierarchical decomposition tree, i.e. the time it takes to construct the component represented by the root in the hierarchical decomposition tree. Let n be the number of nodes in this component. The construction of this component takes $\lceil \log_{18/17} n \rceil$ rounds, where each round takes time proportional to the number of components remaining. Initially, there are n components corresponding to single nodes. Since the number of components decreases geometrically in each round, the total time becomes $O(n)$. □

In the following lemma, we are not concerned with the balance of hierarchical decompositions, but rather with how they can be used to contract the nodes of a tree while leaving a designated set of leaves untouched.

Lemma 4 *Let T be an unrooted tree with n nodes of degree at most three, and let $k \geq 0$ leaves be marked as* non-contractible. *In $O(n)$ time a hierarchical decomposition of T into at most $4k + 1$ components can be computed such that each marked leaf is a component by itself.*

Proof. We construct the decomposition by repeatedly applying valid compositions (cf. Figure 5) on an initial set of components consisting of the n nodes of T. Since each valid composition corresponds to an edge of T, this algorithm takes $O(n)$ time if we maintain a queue of edges corresponding to valid compositions. Consider a situation where the algorithm stops, i.e. where there are no more edges corresponding to valid compositions. We must argue that there are at most $4k + 1$ components. Let n_1, n_2, and n_3 be the number of components of degree one, two, and three respectively. If $n_3 = 0$, the tree is a path and we have $k \leq n_1 \leq 2$. The number of edges where we cannot apply a decomposition is at most k, so the number of components is at most $k + 1 \leq 4k + 1$. If $n_3 \geq 1$, we argue as follows. If $n_1 > k$, then at least one leaf is contractible, and a composition of type (i) or (iii) can by applied. So the only components of degree one are the k marked leaves, and we have $n_1 = k$ and $n_3 = n_1 - 2 = k - 2$. The only edges not corresponding to valid compositions are edges incident to a marked leaf, or edges incident to a component of degree three (cf. the proof

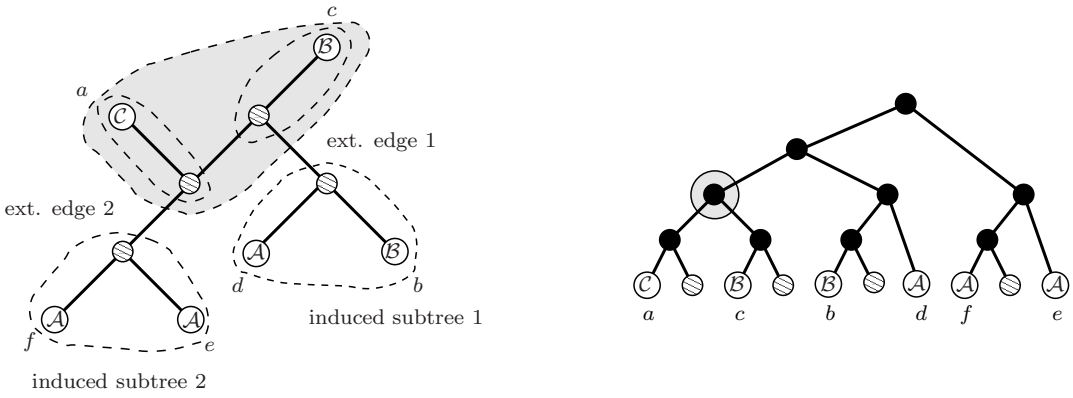

Figure 7: A component in the hierarchical decomposition with two external edges. The component corresponds to the marked node in the hierarchical decomposition tree to the right. This node is decorated with information $(0, 1, 1)$ and $F(\boldsymbol{a}_1, \boldsymbol{b}_1, \boldsymbol{c}_1, \boldsymbol{a}_2, \boldsymbol{b}_2, \boldsymbol{c}_2)$, where $\boldsymbol{a}_i$, $\boldsymbol{b}_i$, and $\boldsymbol{c}_i$ denote the number of elements in leaves from the subtree induced by external edge i which are colored $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$, respectively. In the figure, $(\boldsymbol{a}_1, \boldsymbol{b}_1, \boldsymbol{c}_1, \boldsymbol{a}_2, \boldsymbol{b}_2, \boldsymbol{c}_2) = (1, 1, 0, 2, 0, 0)$. F states, as a function of the variables $\boldsymbol{a}_i$, $\boldsymbol{b}_i$, and $\boldsymbol{c}_i$, the number of the quartets which are both associated with nodes in the component and are compatible with the given coloring. For the highlighted component these are the quartets $ab{\cdot}ef$, $ac{\cdot}ef$, $ae{\cdot}bc$, and $af{\cdot}bc$. In total there are four quartets that are both associated with nodes in the component and are compatible with the given coloring, i.e. $F(1, 1, 0, 2, 0, 0) = 4$.

of Lemma 3), i.e. at most $k + 3n_3 = 4k - 6$ edges do not represent valid compositions. As the components form a tree, the number of components is at most $4k - 5 \leq 4k + 1$. □

4 Counting Quartets in Components

Let T be an evolutionary tree and $H(T)$ be a hierarchical decomposition tree for T. We now describe how to decorate the nodes of $H(T)$ with information such that the number of quartets of T which are compatible with a given coloring of S can be returned in constant time. Furthermore, for a given coloring, the decoration can be generated in $O(n)$ time, and if k elements of S change color, the decoration can be updated in time $O(k + k \log \frac{n}{k})$.

For each node of $H(T)$, we store a tuple (a, b, c) of integers and a function F. Recall that a node in $H(T)$ represents a component in T. The integers a, b, and c of the tuple are the number of elements at the leaves of T contained in this component which are colored $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$, respectively. A component has k external edges for k between zero and three (the case of zero external edges occurs only at the root of $H(T)$). The function F has three variables for each of the external edges of the component. For a component with $k \geq 1$ external edges, we number these edges arbitrarily from 1 to k and denote the three variables corresponding to edge i by $\boldsymbol{a}_i$, $\boldsymbol{b}_i$, and $\boldsymbol{c}_i$. If an external edge was removed from T, two subtrees of T would arise, where one does not contain the component in question. We call this subtree the subtree *induced* by the external edge. The variables $\boldsymbol{a}_i$, $\boldsymbol{b}_i$, and $\boldsymbol{c}_i$ denote the number of elements in leaves from the subtree induced by edge i which are colored $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$, respectively. Finally, F states, as a function of the variables $\boldsymbol{a}_i$, $\boldsymbol{b}_i$, and $\boldsymbol{c}_i$, for $1 \leq i \leq k$, the number of the quartets which are both associated (as defined in Section 2) with nodes in the component *and* are compatible with the given coloring. It will turn

out that F is actually a polynomial of total degree at most four (the total degree of a monomial is the sum of the powers of its variables, and the total degree of a polynomial is the maximum of this over its monomials—for example, the total degree of $x^3y^3 + x^4z$ is six). Figure 7 gives an example of the described decoration.

The root of $H(T)$ represents a component which comprises the entire tree T. This component has no external nodes, so the function F stored there is a constant. Hence, the number of quartets of T which are compatible with a given coloring of S is part of the information stored at the root of $H(T)$.

Lemma 5 *The tree $H(T)$ can be decorated with the information described above in time $O(n)$.*

Proof. The information is computed in a bottom up fashion during a traversal of $H(T)$. We first describe how the information for leaves in $H(T)$ is generated, i.e. for nodes representing single node components. Recall that a node in T is either a leaf and has degree one, or is an internal node and has degree three.

For a component consisting of a single leaf with an element colored $\mathcal{A}$, $\mathcal{B}$, or $\mathcal{C}$, the tuple is $(1,0,0)$, $(0,1,0)$, and $(0,0,1)$, respectively. The function F is identically zero, as quartets are only associated with internal nodes of T, not with leaves of T.

For a component consisting of a single degree three node u, the tuple is $(0,0,0)$, as no leaves of T are contained in the component. The function F should count the number of quartets which are both compatible with the coloring and associated with u in T. A quartet $ab \cdot cd$ fulfills this requirement precisely when c and d are contained in one of the three subtrees induced by the external edges of the component, and they have the same color, and a and b each are in one of the remaining two induced subtrees and each have one of the remaining two colors. For the case that c and d are in the subtree induced by edge number one and have color $\mathcal{A}$, the number of quartets fulfilling this is

$$\binom{\boldsymbol{a}_1}{2} \cdot (\boldsymbol{b}_2\boldsymbol{c}_3 + \boldsymbol{b}_3\boldsymbol{c}_2)\,.$$

Summing over all $3 \cdot 3 = 9$ choices of the induced subtree and color for c and d, we get:

$$\begin{aligned} F(&\boldsymbol{a}_1, \boldsymbol{b}_1, \boldsymbol{c}_1, \boldsymbol{a}_2, \boldsymbol{b}_2, \boldsymbol{c}_2, \boldsymbol{a}_3, \boldsymbol{b}_3, \boldsymbol{c}_3) \\ &= \binom{\boldsymbol{a}_1}{2} \cdot (\boldsymbol{b}_2\boldsymbol{c}_3 + \boldsymbol{b}_3\boldsymbol{c}_2) + \binom{\boldsymbol{a}_2}{2} \cdot (\boldsymbol{b}_1\boldsymbol{c}_3 + \boldsymbol{b}_3\boldsymbol{c}_1) + \binom{\boldsymbol{a}_3}{2} \cdot (\boldsymbol{b}_2\boldsymbol{c}_1 + \boldsymbol{b}_1\boldsymbol{c}_2) \\ &+ \binom{\boldsymbol{b}_1}{2} \cdot (\boldsymbol{a}_2\boldsymbol{c}_3 + \boldsymbol{a}_3\boldsymbol{c}_2) + \binom{\boldsymbol{b}_2}{2} \cdot (\boldsymbol{a}_1\boldsymbol{c}_3 + \boldsymbol{a}_3\boldsymbol{c}_1) + \binom{\boldsymbol{b}_3}{2} \cdot (\boldsymbol{a}_2\boldsymbol{c}_1 + \boldsymbol{a}_1\boldsymbol{c}_2) \\ &+ \binom{\boldsymbol{c}_1}{2} \cdot (\boldsymbol{b}_2\boldsymbol{a}_3 + \boldsymbol{b}_3\boldsymbol{a}_2) + \binom{\boldsymbol{c}_2}{2} \cdot (\boldsymbol{b}_1\boldsymbol{a}_3 + \boldsymbol{b}_3\boldsymbol{a}_1) + \binom{\boldsymbol{c}_3}{2} \cdot (\boldsymbol{b}_2\boldsymbol{a}_1 + \boldsymbol{b}_1\boldsymbol{a}_2) \end{aligned}$$

We now turn to the generation of the information stored in the internal nodes of $H(T)$. Consider the composition of two components C' and C''. Let (a', b', c') and F', and (a'', b'', c'') and F'' be the information stored at the nodes representing the components C' and C''. The information stored at the node representing the composition C of C' and C'' is $(a' + a'', b' + b'', c' + c'')$ and F, where F depends on the type of composition. If the component composition is of type (ii), we consider the case where the numbering of external edges of components is such that the first external edge of C' and C'' is the edge connecting C' and C'', and the second external edge of C' is the first external edge of C, and the second external edge of C'' is the second external edge of C. The remaining

cases of numbering of external edges are obtained by appropriate changes of the arguments to F' and F''.

$$\begin{aligned} & F(\boldsymbol{a}_1, \boldsymbol{b}_1, \boldsymbol{c}_1, \boldsymbol{a}_2, \boldsymbol{b}_2, \boldsymbol{c}_2) \\ & \quad = \quad F'(\boldsymbol{a}_2 + a'', \boldsymbol{b}_2 + b'', \boldsymbol{c}_2 + c'', \boldsymbol{a}_1, \boldsymbol{b}_1, \boldsymbol{c}_1) \\ & \quad + \quad F''(\boldsymbol{a}_1 + a', \boldsymbol{b}_1 + b', \boldsymbol{c}_1 + c', \boldsymbol{a}_2, \boldsymbol{b}_2, \boldsymbol{c}_2) \end{aligned}$$

Component compositions of type (iii) and (iv) are identical to type (ii), except that the definition of F is simpler. For type (iii) we have (assuming that C'' is the component of degree one)

$$F(\boldsymbol{a}_1, \boldsymbol{b}_1, \boldsymbol{c}_1) \;=\; F'(a'', b'', c'', \boldsymbol{a}_1, \boldsymbol{b}_1, \boldsymbol{c}_1) \;+\; F''(\boldsymbol{a}_1 + a', \boldsymbol{b}_1 + b', \boldsymbol{c}_1 + c') \;\;,$$

and for type (iv) we have

$$F \;=\; F'(a'', b'', c'') \;+\; F''(a', b', c') \;\;.$$

Note that for type (iv) compositions, F is a constant. Finally, we for type (i) compositions get the following expression for F, assuming C' has degree one and the first and second external edges of C are the second and third external edges of C'', respectively.

$$\begin{aligned} & F(\boldsymbol{a}_1, \boldsymbol{b}_1, \boldsymbol{c}_1, \boldsymbol{a}_2, \boldsymbol{b}_2, \boldsymbol{c}_2) \\ & \quad = \quad F'(\boldsymbol{a}_1 + \boldsymbol{a}_2 + a'', \boldsymbol{b}_1 + \boldsymbol{b}_2 + b'', \boldsymbol{c}_1 + \boldsymbol{c}_2 + c'') \\ & \quad + \quad F''(a', b', c', \boldsymbol{a}_1, \boldsymbol{b}_1, \boldsymbol{c}_1, \boldsymbol{a}_2, \boldsymbol{b}_2, \boldsymbol{c}_2) \end{aligned}$$

Note that the function F for a component consisting of a single node is a polynomial with at most nine variables and total degree at most four. By structural induction on the definition of the F functions, the same is seen to hold for all components. Polynomials with at most nine variables and total degree at most four can be stored in constant space by storing the coefficients, and they can be manipulated in constant time, e.g. when adding or composing two polynomials. Actually, it can be shown that except for components of degree three, the polynomials have at most six variables and total degree at most three[1]. As all components of degree three have an F of a fixed form, the space required to store the polynomials is less than implied by the bound stated above.

We conclude that for a component C which is the composition of two components C' and C'', the information to be stored at C can be computed in constant time, provided that the information stored at C' and C'' is known. It follows that $H(T)$ can be decorated in time $O(n)$. □

Lemma 6 *The decoration of $H(T)$ can be updated in $O(k + k \log \frac{n}{k})$ time when the color of k elements in S changes.*

Proof. From the proof of Lemma 5 we know that the decoration of a node in $H(T)$ only depends on the decoration of the children of the node in $H(T)$, i.e. the only decorations that need to be updated in $H(T)$ while changing the color of an element in S are the decorations of the ancestors of the leaf in $H(T)$ corresponding to the element. The decoration of a node takes constant time to compute knowing the decoration of the children. Since $H(T)$ is a $(1/\log \frac{18}{17})$-locally balanced tree, we from Lemma 2 have that at most $O(k + k \log \frac{n}{k})$ nodes should be updated. We first mark the

[1]For instance, the exact format of components with two external edges with variables (a_1, b_1, c_1) and (a_2, b_2, c_2) can by structural induction be shown to be $F(a_1, b_1, c_1, a_2, b_2, c_2) = (k_0 + k_1 a_1 + k_2 b_1 + k_3 c_1 + k_4 a_2 + k_5 b_2 + k_6 c_2 + k_7 a_1^2 + k_8 b_1^2 + k_9 c_1^2 + k_{10} a_2^2 + k_{11} b_2^2 + k_{12} c_2^2 + k_{13} a_1 a_2 + k_{14} b_1 b_2 + k_{15} c_1 c_2 + k_{16} a_1 b_2 + k_{17} a_1 c_2 + k_{18} b_1 a_2 + k_{19} b_1 c_2 + k_{20} c_1 a_2 + k_{21} c_1 b_2 + k_{22} a_1^2 b_2 + k_{23} a_1^2 c_2 + k_{24} b_1^2 a_2 + k_{25} b_1^2 c_2 + k_{26} c_1^2 a_2 + k_{27} c_1^2 b_2 + k_{28} a_1 b_2^2 + k_{29} a_1 c_2^2 + k_{30} b_1 a_2^2 + k_{31} b_1 c_2^2 + k_{32} c_1 a_2^2 + k_{33} c_1 b_2^2)/2$, where $k_0, k_1, \ldots, k_{33}$ are integer coefficients.

```
Procedure Count(v)
    if v is a leaf then
        color v by the color C
        return 0
    else
        ColorLeaves(Small(v), B)
        x = NodeCount(v)
        ColorLeaves(Small(v), C)
        y = Count(Large(v))
        ColorLeaves(Small(v), A)
        z = Count(Small(v))
        return x + y + z
```

Figure 8: The basic algorithm.

nodes to be updated bottom-up from each leaf until we find the first already marked node. The decorations of the marked nodes are then updated bottom-up by a traversal of the marked nodes, simultaneously with removing the marks again. In total, we spend time proportional to the number of nodes to be updated. □

5 The Basic Algorithm

In this section, we give an algorithm with running time $O(n \log^2 n)$. The algorithm starts by rooting T_1 at an arbitrary leaf. It then calculates the size $|v|$ of each node v in T_1 during a postorder traversal starting at the root, where $|v|$ denotes the number of leaves below v, and stores this information in the nodes. It also colors all elements of S by the color $\mathcal{A}$, except for the root which is colored $\mathcal{C}$, and builds $H(T_2)$ with decoration based on this coloring. The algorithm then recursively calculates the sum described in Section 2 of counts for all internal nodes of T_1, starting at the single child of the root of T_1. To achieve the claimed complexity, the algorithm at a node v will make a recursive call first on its larger child, then on its smaller child, and finally add the count for v to the sum calculated so far.

In Figure 8 the algorithm is described in pseudo-code as a recursive procedure `Count`(v). A call to `Count`(v) returns the sum of the counts for v and the internal nodes of T_1 below v. Initially, it is called with v set to the single child of the root of T_1. The routines `Small`(v) and `Large`(v) return the child of v having smallest and largest size respectively. The routine `NodeCount`(v) is a call to the data structure of Section 4, which returns the count for the node v by looking at the current information at the root of $H(T_2)$. The routine `ColorLeaves`$(v, \mathcal{X})$ colors by the color $\mathcal{X}$ all elements in the data structure which are labels of leaves below v in T_1. This is done by a traversal of the subtree in T_1 rooted at v. By maintaining bi-directional pointers between elements of S in the data structure and the leaves in T_1 and T_2 which they label, this can be done in the time stated in Lemma 6 with k equal to $|v|$. See Figure 9 for an illustration of the data structures used by the

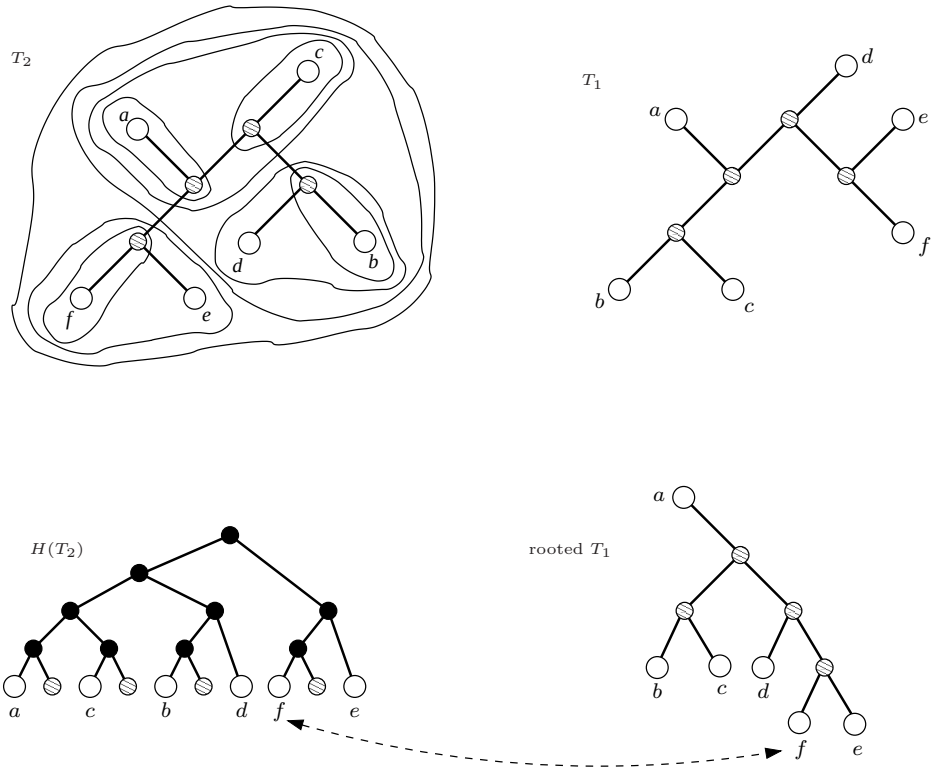

Figure 9: The data structures used by the basic algorithm are the hierarchical decomposition tree $H(T_2)$ decorated with information as described in Section 4, and the tree T_1 rooted at an arbitrary leaf. Pointers are maintained between elements of S in $H(T_2)$ and nodes of degree one in T_1.

basic algorithm.

Theorem 1 *Let T_1 and T_2 be two unrooted evolutionary trees on the same set S of species, and let all internal nodes in the trees have degree three. Then the quartet distance between T_1 and T_2 can be found in time $O(n \log^2 n)$.*

Proof. By induction on time, it follows that the algorithm above maintains the invariants:

1. At the *beginning* of the execution of an instance of `Count`(v), all elements in S which are labels of leaves *below* v in T_1 are colored $\mathcal{A}$, and all other elements in S are colored $\mathcal{C}$.

2. At the *end* of the execution of an instance of `Count`(v), all elements in S are colored $\mathcal{C}$.

The invariants imply that when a call to `NodeCount`(v) takes place, labels of leaves in the subtree of `Small`(v) are labeled by the color $\mathcal{B}$, labels of leaves in the subtree of `Large`(v) are labeled by the color $\mathcal{A}$, and the remaining elements are labeled by the color $\mathcal{C}$. In other words, the elements of S are colored according to v. From the discussion in Section 2, we have that the quartet distance equals $\binom{n}{4}$ minus half the value computed by the algorithm above.

For complexity, note that the work incurred by an instance of `Count`(v), not counting recursive calls made during this instance, is $O(\log n \cdot |\texttt{Small(v)}|)$, by the logarithmic height of $H(T_2)$. Let this work be accounted for by charging each leaf below `Small`(v) in T_1 (or v itself, if it is a leaf)

an amount of $O(\log n)$ work. For a given leaf, this charging can only happen at nodes v on the path from the leaf to the root where the path goes from $\texttt{Small}(v)$ to v. As the size of v is at least twice as large as the size of $\texttt{Small}(v)$, this can only happen $\log n$ times. Hence, each leaf is at most charged $O(\log^2 n)$ work in total, and the result follows. □

6 The Improved Algorithm

In the analysis of our basic algorithm in the previous section, we made use of the fact that if each node v in a binary tree with n leaves supplies a term $c \cdot |\texttt{Small}(v)|$, then the sum over all nodes in the tree is $O(cn \log n)$. In the literature, this is often referred to as the "smaller-half trick". We used it with $c = \log n$.

In this section, we improve the above algorithm to an algorithm with running time $O(n \log n)$. The improvement comes from changes in the algorithm which will allow us to use an "extended smaller-half trick". This stronger result is hinted at in [14, Exercise 35] and formulated in Lemma 7 below. As usual, a full binary tree is a tree where each internal node has two children.

Lemma 7 *Let T be a full binary tree with n leaves. If $c_v = |\texttt{Small}(v)| \log(|v|/|\texttt{Small}(v)|)$ for every internal node v, and $c_v = 0$ for every leaf v, then*

$$\sum_{v \in T} c_v \leq n \log n \,.$$

Proof. The proof is by induction on the size of T. If $|T| = 1$, the lemma holds vacuously. Now assume inductively that the upper bound holds for all trees with at most $n-1$ leaves. Consider a tree with n leaves where the number of leaves in the subtrees rooted at the two children of the root are k and $n-k$ where $0 < k \leq n/2$. According to the induction hypothesis the sum over all nodes in these two subtrees is bounded by respectively $k \log k$ and $(n-k)\log(n-k)$. The entire sum is thus bounded by:

$$\begin{aligned} k\log(n/k) + k\log k + (n-k)\log(n-k) &= k\log n + (n-k)\log(n-k) \\ &< k\log n + (n-k)\log n \\ &= n\log n\,, \end{aligned}$$

which proves the lemma. □

The improvement of the basic algorithm is based on the following observation: In a recursive call to $\texttt{Count}(v)$, only the leaves below v in T_1 can be colored with a color different from $\mathcal{C}$, i.e. if $|v|$ is small then most components in $H(T_2)$ only contain nodes colored $\mathcal{C}$. By contracting such components in T_2 into single nodes, we by Lemma 4 can obtain a contracted version of T_2 with at most $4|v|+1$ nodes. We use this observation to construct an improved algorithm which works with contracted versions of T_2. By contracting T_2 whenever a constant fraction of the leaves has been colored $\mathcal{C}$, namely when $|T_2| > 5|v|$, we can guarantee that $|T_2| = O(|v|)$ when recursively invoking our improved algorithm, instead of $|T_2| = n$ as in the basic algorithm. By Lemma 6, this implies that updating the colors of $\texttt{Small}(v)$ by the three `ColorLeaves` operations takes time $O(|\texttt{Small}(v)| \cdot \log(|v|/|\texttt{Small}(v)|))$. Assuming that this dominates the work incurred by an instance of our improved algorithm, a total running time of $O(n \log n)$ is implied by Lemma 7.

```
Procedure FastCount(v, T)
    local var T'
    if v is a leaf then
        color v by the color C
        return 0
    else
        ColorLeaves(Small(v), B, T)
        x = NodeCount(v)
        T' = Contract(B, Extract(Small(v), T))
        ColorLeaves(Small(v), C, T)
        if |T| > 5 |Large(v)| then
            T = Contract(A, T)
        y = FastCount(Large(v), T)
        ColorLeaves(Small(v), A, T')
        z = FastCount(Small(v), T')
        return x + y + z
```

Figure 10: The extended algorithm.

To avoid the problem of reversing the contractions of T_2 made along one path in the recursion tree when another path is later taken, we make copies of T_2 during the recursion. An input parameter to the recursive procedure is therefore a compressed copy T of T_2, along with an associated hierarchical decomposition tree $H(T)$. In the initial call, we have $T = T_2$.

In Figure 10, our improved algorithm is described as a recursive procedure `FastCount`(v, T). In the pseudo-code, T refers to the tree T as well as its associated data structure. A similar remark applies to the copy T'. The differences between `FastCount`(v, T) and `Count`(v) are the two applications of the routine `Contract` and the single application of the routine `Extract`.

The routine `Contract`$(\mathcal{X}, U)$ applies the algorithm described in the proof of Lemma 4 to the tree U, with the term non-contractible taken to mean the leaves in U colored $\mathcal{X}$. It uses the decomposition resulting from this algorithm as a new tree by taking the components as the new nodes, and the edges between the components as the new edges. The (a, b, c) and F information of the components is inherited by the new nodes. Finally it builds the data structure for this contracted tree using Lemma 3. By Lemma 3 and Lemma 4, the running time of `Contract`$(\mathcal{X}, U)$ is $O(|U|)$.

The routine `Extract`(`Small`(v), T) uses the data structure of T to extract a copy of T at the point in the algorithm where all leaves below `Small`(v) are colored $\mathcal{B}$, all leaves in `Large`(v) are colored $\mathcal{A}$, and the remaining leaves are colored $\mathcal{C}$. In the copy, all leaves below `Small`(v) remain colored $\mathcal{B}$ and all other leaves are colored $\mathcal{C}$, i.e. the color of leaves below `Large`(v) has been changed from $\mathcal{A}$ to $\mathcal{C}$. We give the details of `Extract` below. The routine `Contract` is applied to the copy, and the resulting contracted copy T' of T is used for the subsequent recursive call on `Small`(v).

Just as in the basic algorithm, to perform `ColorLeaves` we need pointers between leaves in T_1 and elements of S in $H(T)$. However, in the extended algorithm of Figure 10, the first two calls

to `ColorLeaves` work on T, while the last call works on T'. As part of the construction of T', we therefore make a list where each element points to a leaf below $\texttt{Small}(v)$ in T_1 and to the corresponding element of S in T'. Just before the third call to `ColorLeaves`, we traverse this list and update the actual pointers in leaves below $\texttt{Small}(v)$ in T_1 to point to nodes of T' instead of T. For brevity, this is not shown in Figure 10.

We now give the details of $\texttt{Extract}(\texttt{Small}(v), T)$. As described in Section 3, the data structure of T is the binary tree structure $H(T)$. To extract the copy from T, we mark all internal nodes in $H(T)$ on paths from the leaves of color $\mathcal{B}$ to the root by marking bottom-up from each leaf until we find the first already marked node. We then traverse the marked part of the data structure, and identify the subtrees that would arise if the marked nodes were removed. In $H(T)$, internal nodes correspond to edges in T, and subtrees correspond to components of T of degree one, two, or three. Hence, the subtrees that would remain after removal of the marked nodes are components of T of degree one, two, or three. For each of these components, we create a new node for the extracted tree with the same degree as the component in T it corresponds to. These steps are illustrated as part of Figure 11, which gives an example of the construction of T'.

To be able to consider all leaves in the components as being colored $\mathcal{C}$, we extend the definition of the data structure in Section 4 such that each component also stores a function $F_{\mathcal{C}}$ defined equivalently to the function F, except that it assumes all leaves in the component to be colored $\mathcal{C}$. If a component before the extraction stores a tuple (a, b, c) and functions F and $F_{\mathcal{C}}$, then the corresponding node in the extracted tree stores the tuple $(0, 0, a + b + c)$ and the functions $F_{\mathcal{C}}$ and $F_{\mathcal{C}}$. We connect two new nodes v and u by an edge if an outgoing edge of the component corresponding to v is the same edge as an outgoing edge of the component corresponding to u. Note that the edges in any T will be a subset of the edges in the original evolutionary tree T_2 (as `Extract` and `Contract` maintain this invariant). If the $2n - 3$ edges of the original T_2 are labeled with the integers $1, \ldots, 2n - 3$, we can therefore connect the nodes in time proportional to the number of nodes by using the labels of outgoing edges as indexes into an array and connecting nodes ending up in the same entry.

In total, by Lemma 2, the extraction takes time $O(|\texttt{Small}(v)| \cdot \log(|T|/|\texttt{Small}(v)|))$. When we perform the operation $\texttt{Extract}(\texttt{Small}(v), T)$ on an instance of $\texttt{FastCount}(v, T)$, we have enforced $|T| = O(|v|)$, and therefore $\texttt{Extract}(\texttt{Small}(v), T)$ is performed within the same time bound $O(|\texttt{Small}(v)| \cdot \log(|v|/|\texttt{Small}(v)|))$ as the three `ColorLeaves` operations. The extracted tree has $O(|\texttt{Small}(v)| \cdot \log(|v|/|\texttt{Small}(v)|))$ nodes. By applying the linear time routine `Contract` to the extracted tree, we get an equivalent tree T' of size at most $4|\texttt{Small}(v)| + 1$.

Theorem 2 *Let T_1 and T_2 be two unrooted evolutionary trees on the same set S of species, and let all internal nodes in the trees have degree three. The quartet distance between T_1 and T_2 can be found in time $O(n \log n)$ and space $O(n)$.*

Proof. The extended algorithm $\texttt{FastCount}(v, T)$ obeys the same invariants about the coloring as stated in the proof of Theorem 1. The correctness of $\texttt{FastCount}(v, T)$ thus follows from the correctness of $\texttt{Count}(v)$. For time complexity, we have already observed that the three `ColorLeaves` operations and the single `Extract` operation can be performed in time $O(|\texttt{Small}(v)| \cdot \log(|v|/|\texttt{Small}(v)|))$, which by Lemma 7 (with T_1 for T) amounts to time $O(n \log n)$ in total during the entire recursion of the topmost call to $\texttt{FastCount}(v, T_2)$.

We now consider the time spent contracting T. We perform the $T = \texttt{Contract}(\mathcal{A}, T)$ operation whenever $|T| > 5\,|\texttt{Large}(v)|$. Since all leaves in $\texttt{Large}(v)$ are colored $\mathcal{A}$ when we contract, the size of the contracted T is at most $4\,|\texttt{Large}(v)| + 1$ by Lemma 4. Hence, the size of T is reduced by a factor $4/5$. This implies that the sequence of contractions applied to any specific copy of the

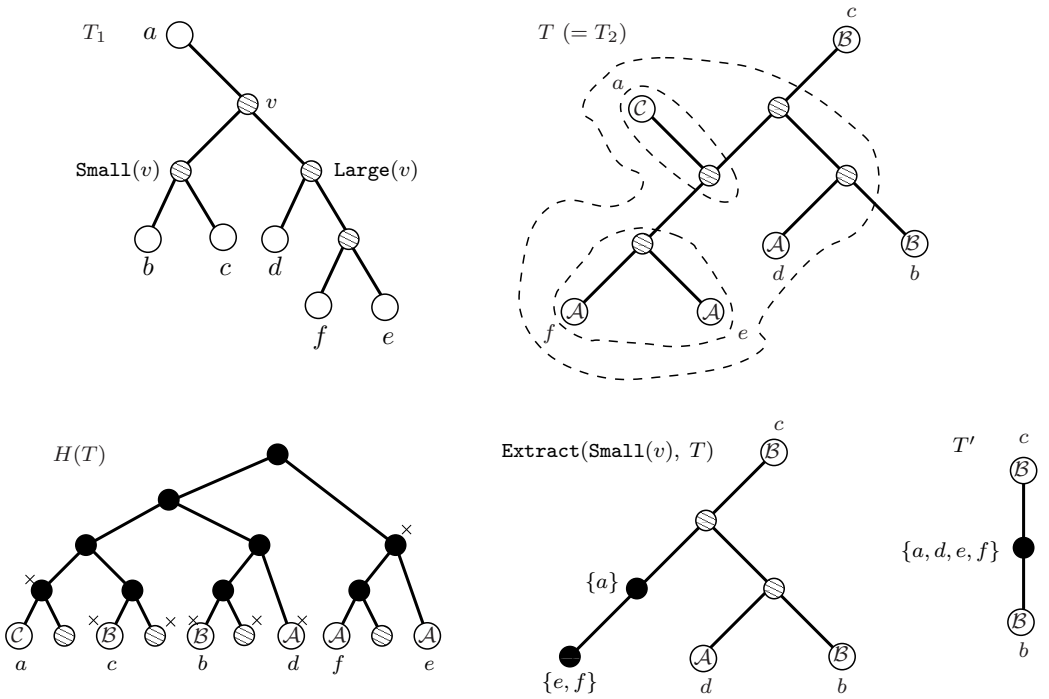

Figure 11: The construction of T' in the extended algorithm. The particular case shown is from the first call to `FastCount`, where T is identical to T_2. The nodes with crosses in $H(T)$ are the roots of subtrees identified during `Extract`(`Small`(v), T). The nodes labeled with sets of species represent contracted components of T, and these components are shown with dashed lines.

data structure results in a sequence of data structures of geometrically decreasing sizes. Since a contraction takes time $O(|T|)$, the total time spent on contracting a copy is linear in its initial size, i.e. it is dominated by the time for constructing the copy by the `Extract` routine.

For space complexity, we observe that the space consumption is $O(n)$ unless the copied trees consume too much space. However, we observe that on any path in the recursion, i.e. path in T_1, the sizes of the T' trees created at each node v in T_1 in the recursion has size $O(|\texttt{Small}(v)|)$ which is bounded by the size of the subtree in T_1 rooted at the child of v not in the recursion path (either $\texttt{Small}(v)$ or $\texttt{Large}(v)$). This implies that the extracted trees consume space $O(n)$ in total. □

Acknowledgment

We thank Mike Steel and an anonymous referee for detailed comments on the presentation.

References

[1] B. L. Allen and M. Steel. Subtree transfer operations and their induced metrics on evolutionary trees. *Annals of Combinatorics*, 5:1–13, 2001.

[2] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.

[3] G. S. Brodal, R. Fagerberg, and C. N. S. Pedersen. Computing the quartet distance between evolutionary trees on time $O(n \log^2 n)$. In *Proceedings of the 12th International Symposium on Algorithms and Computation (ISAAC)*, volume 2223 of *Lecture Notes in Computer Science*, pages 731–742. Springer-Verlag, 2001.

[4] G. S. Brodal, R. Fagerberg, C. N. S. Pedersen, and A. Östlin. The complexity of constructing evolutionary trees using experiments. In *Proceedings of the 28th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 2076 of *Lecture Notes in Computer Science*, pages 140–151. Springer-Verlag, 2001.

[5] G. S. Brodal and C. N. S. Pedersen. Finding maximal quasiperiodicities in strings. In *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 1848 of *Lecture Notes in Computer Science*, pages 397–411. Springer-Verlag, 2000.

[6] D. Bryant, J. Tsang, P. E. Kearney, and M. Li. Computing the quartet distance between evolutionary trees. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 285–286, 2000.

[7] P. Buneman. The recovery of trees from measures of dissimilairty. *Mathematics in Archeological and Historial Sciences*, pages 387–395, 1971.

[8] R. F. Cohen and R. Tamassia. Dynamic expression trees. *Algorithmica*, 13(3):245–265, 1995.

[9] G. Estabrook, F. McMorris, and C. Meacham. Comparison of undirected phylogenetic trees based on subtrees of four evolutionary units. *Systematic Zoology*, 34(2):193–200, 1985.

[10] M. Farach, S. Kannan, and T. J. Warnow. A robust model for finding optimal evolutionary trees. *Algorithmica*, 13(1/2):155–179, 1995.

[11] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[12] S. K. Kannan, E. L. Lawler, and T. J. Warnow. Determining the evolutionary tree using experiments. *Journal of Algorithms*, 21(1):26–50, July 1996.

[13] A. Lingas, H. Olsson, and A. Östlin. Efficient merging, construction, and maintenance of evolutionary trees. In *Proceedings of the 26th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1644 of *Lecture Notes in Computer Science*, pages 544–553. Springer-Verlag, 1999.

[14] K. Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*. Springer-Verlag, 1984.

[15] J. Pearl and M. Tarski. Structuring causal trees. *Journal of Complexity*, 2:60–77, 1986.

[16] D. F. Robinson and L. R. Foulds. Comparison of weighted labelled trees. In *Combinatorial mathematics, VI (Proceedings of the 6th Australian Conference, University of New England, Armidale, 1978)*, Lecture Notes in Mathematics, pages 119–126. Springer-Verlag, Berlin, 1979.

[17] D. F. Robinson and L. R. Foulds. Comparison of phylogenetic trees. *Mathematical Biosciences*, 53(1-2):131–147, 1981.

[18] M. Steel and D. Penny. Distribution of tree comparison metrics—some new results. *Systematic Biology*, 42(2):126–141, 1993.

[19] M. S. Waterman and T. F. Smith. On the similarity of dendrograms. *Journal of Theoretical Biology*, 73:789–800, 1978.