Cache Oblivious Algorithms for Computing the Triplet Distance Between Trees*†

Gerth Stølting Brodal[1] and Konstantinos Mampentzidis[2]

1 **Department of Computer Science, Aarhus University, Aarhus, Denmark**
gerth@cs.au.dk

2 **Department of Computer Science, Aarhus University, Aarhus, Denmark**
kmampent@cs.au.dk

Abstract

We study the problem of computing the triplet distance between two rooted unordered trees with n labeled leafs. Introduced by Dobson 1975, the triplet distance is the number of leaf triples that induce different topologies in the two trees. The current theoretically best algorithm is an $O(n \log n)$ time algorithm by Brodal *et al.* (SODA 2013). Recently Jansson *et al.* proposed a new algorithm that, while slower in theory, requiring $O(n \log^3 n)$ time, in practice it outperforms the theoretically faster $O(n \log n)$ algorithm. Both algorithms do not scale to external memory.

We present two cache oblivious algorithms that combine the best of both worlds. The first algorithm is for the case when the two input trees are binary trees and the second a generalized algorithm for two input trees of arbitrary degree. Analyzed in the RAM model, both algorithms require $O(n \log n)$ time, and in the cache oblivious model $O(\frac{n}{B} \log_2 \frac{n}{M})$ I/Os. Their relative simplicity and the fact that they scale to external memory makes them achieve the best practical performance. We note that these are the first algorithms that scale to external memory, both in theory and practice, for this problem.

1998 ACM Subject Classification G.2.2 Trees, G.2.1 Combinatorial Algorithms

Keywords and phrases Phylogenetic tree, tree comparison, triplet distance, cache oblivious algorithm

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.1

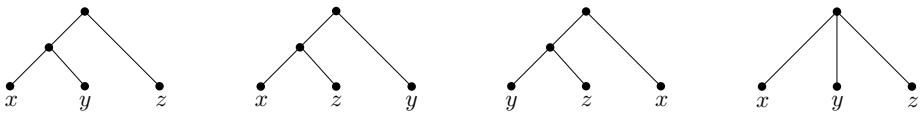
1 Introduction

Background. Trees are data structures that are often used to represent relationships. For example in the field of Biology, a tree can be used to represent evolutionary relationships, with the leafs corresponding to species that exist today, and internal nodes to ancestor species that existed in the past. For a fixed set of n species, different data or construction methods (e.g. Q* [2], neighbor joining [13]) can lead to trees that look structurally different. An interesting question that arises then is, given two trees T_1 and T_2 over n species, how different are they? An answer to this question could potentially be used to determine whether the difference is statistically significant or not, which in turn could help with evolutionary inferences. Several ways of comparing two trees have been proposed in the past, with different types of trees (e.g. rooted versus unrooted, binary versus arbitrary degree) having different distance measures (e.g. Robinson-Foulds distance [12], triplet distance [6], quartet distance [7]). In this paper we focus on the triplet distance computation, which is defined for rooted trees.

* Research supported by the Danish National Research Foundation, grant DNRF84, Center for Massive Data Algorithmics (MADALGO).

† An extended version of the paper is available on arXiv [4].

© Gerth Stølting Brodal and Konstantinos Mampentzidis;
licensed under Creative Commons License CC-BY
25th Annual European Symposium on Algorithms (ESA 2017).
Editors: Kirk Pruhs and Christian Sohler; Article No. 1; pp. 1:1–1:13
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



(a) $xy|z$ **(b)** $xz|y$ **(c)** $yz|x$ **(d)** xyz

Figure 1 Triplet topologies.

Problem Definition. For a given rooted unordered tree T where each leaf has a unique label, a *triplet* is defined by a set of three leaf labels x, y and z and their induced topology in T. The four possible topologies are illustrated in Figure 1. For two such trees T_1 and T_2 that are built on n identical leaf labels, the *triplet distance* $D(T_1, T_2)$ is the number of triplets that are different in T_1 and T_2. Let $S(T_1, T_2)$ be the number of *shared* triplets in the two trees, i.e. leaf triples with identical topologies in the two trees. We have the relationship that $D(T_1, T_2) + S(T_1, T_2) = \binom{n}{3}$.

Results. All related work can be found in [5, 1, 14, 3, 15, 9, 10, 11]. Previous and new results are shown in the table below. For the cache oblivious model [8], the papers [5, 1, 14, 3, 10, 11] do not provide an analysis, so here we provide an upper bound.

Year	Reference	Time	IOs	Space	Non-Binary Trees
1996	Critchlow *et al.* [5]	$O(n^2)$	$O(n^2)$	$O(n^2)$	no
2011	Bansal *et al.* [1]	$O(n^2)$	$O(n^2)$	$O(n^2)$	yes
2013	Brodal *et al.* [14]	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(n)$	no
2013	Brodal *et al.* [3]	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	yes
2015	Jansson *et al.* [10, 11]	$O(n \log^3 n)$	$O(n \log^3 n)$	$O(n \log n)$	yes
2017	new	$O(n \log n)$	$O(\frac{n}{B} \log_2 \frac{n}{M})$	$O(n)$	yes

The common main bottleneck with all previous approaches is that the data structures used rely intensively on $\Omega(n \log n)$ random memory accesses. This means that all algorithms are penalized by cache performance and thus do not scale to external memory. We address this limitation by proposing new algorithms for computing the triplet distance on binary and non-binary trees, that match the previous best $O(n \log n)$ time and $O(n)$ space bounds in the RAM model, but for the first time also scale to external memory. More specifically, in the cache oblivious model, the total number of I/Os required is $O(\frac{n}{B} \log_2 \frac{n}{M})$. The basic idea is to essentially replace the dependency of random access to data structures by scanning contracted versions of the input trees. A careful implementation of the algorithms is shown to achieve the best practical performance, thus essentially documenting that the theoretical results carry over to practice.

2 Previous Approaches

A naive algorithm would enumerate over all $\binom{n}{3}$ sets of 3 labels and find for each set whether the induced topologies in T_1 and T_2 differ or not, giving an $O(n^3)$ algorithm. This naive approach does not exploit the fact that the triplets are not completely independent. For example the triplets $xy|z$ and $yx|u$ share the leafs x and y and the fact that the lowest common ancestor of x and y is at a lower depth than the lowest common ancestor of z with either x or y and the lowest common ancestor of u with either x or y. Dependencies like this can be exploited to count the number of shared triplets faster.

Critchlow *et al.* [5] exploit the depth of the leafs' ancestors to achieve the first improvement over the naive approach. Bansal *et al.* [1] exploit the shared leafs between subtrees and reduce the problem to computing the intersection size (number of shared leafs) of all pairs of subtrees, one from T_1 and one from T_2, which can be solved with dynamic programming.

The $\mathrm{O}(n^2)$ Algorithm for Binary Trees in [14]. The algorithm for binary trees in [14] is the basis for all subsequent improvements [14, 3, 10], including ours as well, so we will describe it in more detail here. The dependency that was exploited is the same as in [1], but the procedure for counting the shared triplets is completely different. More specifically, each triplet in T_1 and T_2, defined by the leafs i, j and k, is implicitly *anchored* in the lowest common ancestor of i, j and k. For a node u in T_1 and v in T_2, let $s(u)$ and $s(v)$ be the set of triplets that are anchored in u and v respectively. For the number of shared triplets $S(T_1, T_2)$ we then have that

$$S(T_1, T_2) = \sum_{u \in T_1} \sum_{v \in T_2} |s(u) \cap s(v)| \ .$$

For the algorithm to be $\mathrm{O}(n^2)$ the value $|s(u) \cap s(v)|$ must be computed in $\mathrm{O}(1)$ time. This is achieved by a leaf colouring procedure as follows: Fix a node u in T_1 and color the leafs in the left subtree of u *red*, the leafs in the right subtree of u *blue*, let every other leaf have no color and then transfer this coloring to the leafs in T_2, i.e. identically labelled leafs get the same color. To compute $|s(u) \cap s(v)|$ we do as follows: let l and r be the left and right children of v, and let $w_{\texttt{red}}$ and $w_{\texttt{blue}}$ be the number of red and blue leafs in a subtree rooted at a node w in T_2. We then have that

$$|s(u) \cap s(v)| = \binom{l_{\texttt{red}}}{2} r_{\texttt{blue}} + \binom{l_{\texttt{blue}}}{2} r_{\texttt{red}} + \binom{r_{\texttt{red}}}{2} l_{\texttt{blue}} + \binom{r_{\texttt{blue}}}{2} l_{\texttt{red}} \ . \tag{1}$$

Subquadratic Algorithms. To reduce the time, Brodal *et al.* [14] applied the *smaller half trick*, which specifies a depth first order to visit the nodes u of T_1, so that each leaf in T_1 changes color at most $\mathrm{O}(\log n)$ times. To count shared triplets efficiently without scanning T_2 completely for each node u in T_1, the tree T_2 is stored in a data structure denoted a *hierarchical decomposition tree* (*HDT*). This HDT maintains for the current visited node u in T_1, according to (1) the sum $\sum_{v \in T_2} |s(u) \cap s(v)|$, so that each color change in T_1 can be updated efficiently in T_2. In [14] the HDT is a binary tree of height $\mathrm{O}(\log n)$ and every update can be done in a leaf to root path traversal in the HDT, which in total gives $\mathrm{O}(n \log^2 n)$ time. In [3] the HDT is generalized to also handle non-binary trees, each query operates the same, and now due to a contraction scheme of the HDT the total time is reduced to $\mathrm{O}(n \log n)$. Finally, in [10] as an HDT the so called *heavy-light tree decomposition* is used. Note that the only difference in all $\mathrm{O}(n \operatorname{polylog} n)$ results that are available until now is the type of HDT used.

In terms of external memory efficiency, every $\mathrm{O}(n \operatorname{polylog} n)$ algorithm performs $\Theta(n \log n)$ updates to an HDT data structure, which means that for sufficiently large input trees every algorithm requires $\Omega(n \log n)$ I/Os.

3 The New Algorithm for Binary Trees

Overview. We will use the $\mathrm{O}(n^2)$ algorithm described in Section 2 as a basis. The main difference lies in the order that we visit the nodes of T_1 and how we process T_2 when we count. We propose a new order of visiting the nodes of T_1, which we find by applying a hierarchical decomposition on T_1. Every component in this decomposition corresponds to a

connected part of T_1 and a contracted version of T_2. In simple terms, if Λ is the set of leafs in a component of T_1, the contracted version of T_2 is a binary tree on Λ that preserves the topologies induced by Λ in T_2 and has size $O(|\Lambda|)$. To count shared triplets, every component of T_1 has a representative node u that we use to scan the corresponding contracted version of T_2 in order to find $\sum_{v \in T_2} |s(u) \cap s(v)|$. Unlike previous algorithms, we do not store T_2 in a data structure. We process T_2 by contracting and counting, both of which can be done by scanning. At the same time, even though we apply a hierarchical decomposition on T_1, the only reason why we do so, is so we can find the order in which to visit the nodes of T_1. This means that we do not need to store T_1 in a data structure either. Thus, we completely remove the need of data structures (and thereby random memory accesses) and scanning becomes the basic primitive in the algorithm. To make our algorithm I/O efficient, all that remains to be done is to use a proper layout to store the trees in memory, so that every time we scan a tree of size s we spend $O(s/B)$ I/Os.

Preprocessing. As a preprocessing step, first we make T_1 *left heavy*, by swapping children so that for every node u in T_1 the left subtree is larger than the right subtree, by a depth first traversal. Second, we change the leaf labels of T_1, which can also be done by a depth first traversal of T_1, so that the leafs are numbered 1 to n from left to right. This step takes $O(n)$ time in the RAM model. The second step is done to simplify the process of transferring the leaf colors between T_1 and T_2. The coloring of a subtree in T_1 will correspond to assigning the same color to a contiguous range of leaf labels. Determining the color of a leaf in T_2 will then require one `if-statement` to find in what range (red or blue) its label belongs to.

Centroid Decomposition. For a given rooted binary tree T we let $|T|$ denote the number of nodes in T (internal nodes and leafs). For a node u in T we let l and r be the left and right children of u, and p the parent. Removing u from T partitions T into three (possibly empty) *connected components* T_l, T_r and T_p containing l, r and p, respectively. A *centroid* is a node u in T such that $\max\{|T_l|, |T_r|, |T_p|\} \leq |T|/2$. A centroid always exists and can be found by starting from the root of T and iteratively visiting the child with a largest subtree, eventually we will reach a centroid. Finding the size of every subtree and identifying u takes $O(|T|)$ time in the RAM model. By recursively finding centroids in each of the three components, we will in the end get a ternary tree of centroids, which is called the *centroid decomposition* of T, denoted $CD(T)$. We can generate a level of $CD(T)$ in $O(|T|)$ time, given the decomposition of T into components by the previous level. Since we have to generate at most $1 + \log_2(|T|)$ levels, the total time required to build $CD(T)$ is $O(|T| \log |T|)$, hence we get Lemma 1.

▶ **Lemma 1.** *For any rooted binary tree T with n leafs, building $CD(T)$ takes $O(n \log n)$ time in the RAM model.*

A component in a centroid decomposition $CD(T)$, might have many edges crossing its boundaries (connecting nodes inside and outside the component). The below *modified centroid decomposition*, denoted $MCD(T)$, generates components with at most two edges crossing the boundary, one going towards the root and one down to exactly one subtree.

Modified Centroid Decomposition. An $MCD(T)$ is built recursively as follows: If a component C has no edge from below, we select the centroid c of C as a splitting node as described above. Otherwise, let (x, y) be the edge that crosses the boundary from below, where x is in C and let c be centroid of C. As a splitting node choose the lowest common

ancestor of x and c. By induction every component has at most one edge from below and one edge from above. A useful property of $MCD(T)$ is captured by the following lemma:

▶ **Lemma 2.** *For any rooted binary tree T with n leafs, we have that $h(MCD(T)) \leq 2+2\log_2 n$, where $h(MCD(T))$ denotes the height of $MCD(T)$.*

Since each level of $MCD(T)$ can be constructed in $\mathrm{O}(n)$ time, we have

▶ **Theorem 3.** *For any rooted binary tree T with n leafs, building $MCD(T)$ takes $\mathrm{O}(n \log n)$ time in the RAM model.*

To return to our original problem, we visit the nodes of T_1, given by the depth first traversal of the ternary tree $MCD(T_1)$, where the children of every node u in $MCD(T_1)$ are visited from left to right. For every such node u we process T_2 in two phases, the *contraction* phase and the *counting* phase.

Contraction. Let $L(T_2)$ denote the set of leafs in T_2 and $\Lambda \subseteq L(T_2)$. In the contraction phase, T_2 is compressed into a binary tree of size $\mathrm{O}(|\Lambda|)$ whose leaf set is Λ. The contraction is done in a way so that all the topologies induced by Λ in T_2 are preserved in the compressed binary tree. This is achieved by the following three sequential steps: prune all leafs of T_2 that are not in Λ, repeatedly prune all internal nodes of T_2 with no children and repeatedly contract unary internal nodes, i.e. nodes having exactly one child.

Let u be a node of $MCD(T_1)$ and C_u the corresponding component of T_1. For every such node u we have a contracted version of T_2, from now on referred to as $T_2(u)$, where $L(T_2(u)) = L(C_u)$. The goal is to augment $T_2(u)$ with counters (see counting phase below), so that we can find $\sum_{v \in T_2} |s(u) \cap s(v)|$ by scanning $T_2(u)$. One can imagine $MCD(T_1)$ as being a tree where each node u is augmented with $T_2(u)$. To generate all contractions of T_2 for level i of $MCD(T_1)$, which correspond to a set of disjoint connected components in T_1, we can reuse the contractions of T_2 at level $i-1$ in $MCD(T_1)$. This means that we have to spend $\mathrm{O}(n)$ time to generate the contractions of level i, so to generate all contractions of T_2 we need $\mathrm{O}(n \log n)$ time. Note that by explicitly storing all contractions, we will also need to use $\mathrm{O}(n \log n)$ space. For our problem, we traverse $MCD(T_1)$ in a depth first manner, so we only have to store a stack of contractions corresponding to the stack of nodes of $MCD(T_1)$ that we have to remember during our traversal. Since the components at every second level of $MCD(T_1)$ have at most half the size of the components two levels above, Lemma 4 states that the size of this stack is always $\mathrm{O}(n)$.

▶ **Lemma 4.** *Let T_1 and T_2 be two rooted binary trees with n leafs and $u_1, u_2, ..., u_k$ a root to leaf path of $MCD(T_1)$. For the corresponding contracted versions $T_2(u_1), T_2(u_2), ..., T_2(u_k)$ we have that $\sum_{i=1}^{k} |T_2(u_i)| = \mathrm{O}(n)$.*

Counting. In the counting phase, we find $\sum_{v \in T_2} |s(u) \cap s(v)|$ by scanning $T_2(u)$ instead of T_2. This makes the total time of the algorithm in the RAM model $\mathrm{O}(n \log n)$. We consider the following two cases:

- *C_u has no edges from below.*
  In this case C_u corresponds to a complete subtree of T_1. We act exactly like in the $\mathrm{O}(n^2)$ algorithm (Section 2) but now instead of scanning T_2 we scan $T_2(u)$.
- *C_u has one edge from below.*
  In this case C_u does not correspond to a complete subtree of T_1, since the edge from below C_u, will point to a subtree X_u, that is located outside of C_u (for an illustration of

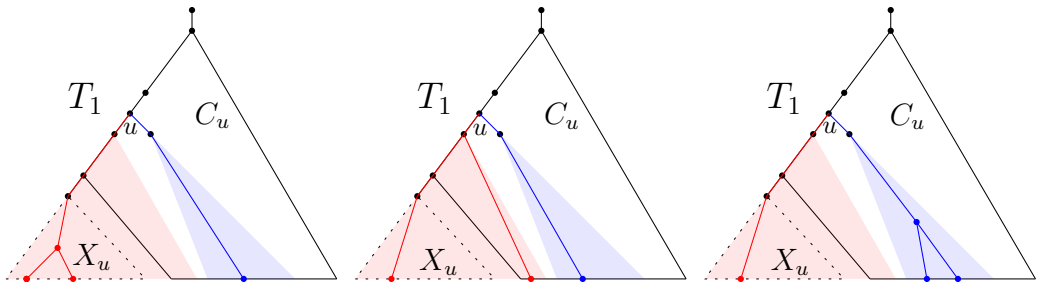


Figure 2 $MCD(T_1)$: Triplets that can be anchored in u with the leafs not being in the component C_u.

this case see Figure 2). Note that because T_1 is left heavy, X_u is always rooted in a node on the left most path from u. The leafs in X_u are important because they can be used to form triplets that are anchored in u. Acting in the same manner as in the previous case is not sufficient because we need to count the triplets involving X_u as well.

To address this problem, every edge (p_v, v) in $T_2(u)$ between a node v and its parent p_v, is augmented with some counters about the leafs from X_u that were contracted away in T_2. For every such edge (p_v, v), let $s_1, s_2, ..., s_k$ be the contracted subtrees rooted on the edge. Every such subtree contains either leafs with no color or leafs from X_u that have the color red (the color can not be blue because T_1 is left heavy). For every node v in $T_2(u)$ the counters that we have are the following:

- $v_{\texttt{red}}$: total number of red leafs in the subtree of v (including those coming from X_u).
- $v_{\texttt{blue}}$: total number of blue leafs in the subtree of v.
- v_{ts}: total number of red leafs in $s_1, s_2, ..., s_k$.
- v_{ps}: total number of pairs of red leafs in $s_1, s_2, ..., s_k$ such that each pair comes from the same contracted subtree, i.e. $\sum_{i=1}^{k} \binom{r_i}{2}$ where r_i is the number of red leafs in s_i.

The number of shared triplets that are anchored in a non-contracted node v of $T_2(v)$ can be found like in the $O(n^2)$ algorithm using the counters $v_{\texttt{red}}$ and $v_{\texttt{blue}}$ in (1). As for the number of shared triplets that are anchored in a contracted node on edge (p_v, v), this value is exactly $\binom{v.\texttt{blue}}{2} \cdot v_{ts} + v_{\texttt{blue}} \cdot v_{ps}$.

Scaling to External Memory. If we store T_1 in an array of size $2n-1$ by using a preorder layout, i.e. if a node v is stored in position p, the left child of v is stored in position $p+1$ and if x is the size of the left subtree of v the right child of v is stored in position $p+x+1$, we can make T_1 left heavy in two depth first traversals using $O(n/B)$ I/Os. The preprocessing step that changes the labels of the leafs in T_1 and T_2 can be done in $O(\frac{n}{B}\log_2 \frac{n}{M})$ I/Os with a cache oblivious sorting routine, e.g. using merge sort. By scanning the left most path that starts from the root of a component C_u, we can find the splitting node of C_u in $O(|C_u|/B)$ I/Os, so in total the number of I/Os spent processing T_1 becomes $O(\frac{n}{B}\log_2 \frac{n}{M})$.

We use the proof of Lemma 4 (see [4]) to initialize an array that can fit the contractions that we need to remember while traversing $MCD(T_1)$. This array is used as a stack that we use to push and pop the contractions of T_2. Each contraction of T_2 is stored in memory using a post order layout, i.e. if a node v is stored in position p and y is the size of the right subtree of v, the left child of v is stored in position $v-y-1$ and the right child of v is stored in position $v-1$. By using a stack, counting and contracting $T_2(u)$ requires $O(|T_2(u)|/B)$ I/Os, so the total number of I/Os spent processing T_2 becomes $O(\frac{n}{B}\log_2 \frac{n}{M})$ as well.

Overall, our algorithm requires $O(\frac{n}{B}\log_2 \frac{n}{M})$ I/Os in the cache oblivious model.

4 The New Algorithm for General Trees

In this algorithm we anchor the triplets of T_1 and T_2 in edges. Let t be a triplet with leafs i, j and k that is either a resolved triplet $ij|k$ or an unresolved triplet ijk, where i is to the left of j and for the triplet ijk, k is also to the right of j. Let w be the lowest common ancestor of i and j and (w, c) the edge from w to the child c whose subtree contains j. We anchor t in edge (w, c). Define $s'(w, c)$ to be the number of triplets anchored in edge (w, c).

Preprocessing. In the preprocessing step of the algorithm, we start by transforming T_1 into a binary tree, denoted $b(T_1)$. Let w be a node of T_1 that has exactly k children, where $k > 2$. The k edges that connect w to its children in T_1 are replaced in $b(T_1)$ by a so called *orange binary tree*. The root of this binary tree is w and the leafs are the k children of w in T_1. Every internal node (except the root) and edge is colored orange, hence the given name. We assume that node w and its k children in T_1, in $b(T_1)$ have the color *black*. This binary tree is built in a way so that every orange node is on the left most path that starts from w, and its left most leaf stores the heaviest child of w in T_1, thus making $b(T_1)$ left heavy. The order in which the other children of w in T_1 are stored in the remaining leafs does not matter, however for the notation below to be mathematically correct, we assume that after constructing $b(T_1)$, the left to right order of the children of w in T_1 is implicitly updated, so that it matches the left to right order in which they appear in the leafs of the orange binary tree below w in $b(T_1)$.

Let u be a node in $b(T_1)$ and c its right child. By construction, c must be a black node. If u is orange, then let $u_{\texttt{root}}$ be the root of the orange binary tree that u is part of. If u is black, then let $u_{\texttt{root}} = u$. Again by construction, $u_{\texttt{root}}$ must be the parent of c in T_1. For the edge (u, c) in $b(T_1)$, we define $s''(u, c)$ to be the set of triplets that are anchored in edge (u_{root}, c) of T_1. Note that for an edge (u', c') in $b(T_1)$ connecting u' with its left child c' we have $s''(u', c') = 0$.

For the number of shared triplets we then have:

$$S(T_1, T_2) = \sum_{(u,c)\in b(T_1)} \sum_{(v,c')\in T_2} |s''(u, c) \cap s'(v, c')| \ .$$

We can capture all triplets in T_1 by coloring $b(T_1)$ instead of T_1. For the nodes u and c where c is the right child of u, the leafs of $b(T_1)$ are colored according to edge (u, c) as follows: the leafs in the left subtree of u are colored red, the leafs in the right right subtree of u are colored blue. If u is an orange node, then the black leafs in the remaining subtrees of the orange binary tree that u is part of are colored green. All other leafs of $b(T_1)$ maintain their color black.

The reason behind transforming T_1 into the binary tree $b(T_1)$, is because now we can use exactly the same core ideas described in Section 3. The tree $b(T_1)$ is a binary tree, so we apply the same preprocessing step, except we do not make it left heavy because by construction it already is. However, we change the labels of the leafs in $b(T_1)$ and T_2, so that the leafs in $b(T_1)$ are numbered 1 to n from left to right.

Modified Centroid Decomposition. After the preprocessing step, we build $MCD(b(T_1))$ as described in Section 3. Then we traverse the nodes of $b(T_1)$, given by a depth first traversal of $MCD(b(T_1))$, where we visit the children of every node u in $MCD(b(T_1))$ from left to right.

Like in the binary algorithm, while traversing $MCD(b(T_1))$ we process T_2 in two phases, the contraction phase and the counting phase. The only difference after this point in the

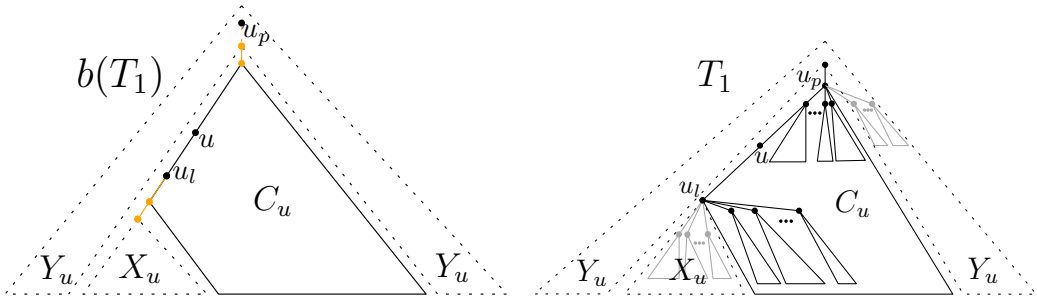


Figure 3 How a component in $b(T_1)$ translates to a component in T_1.

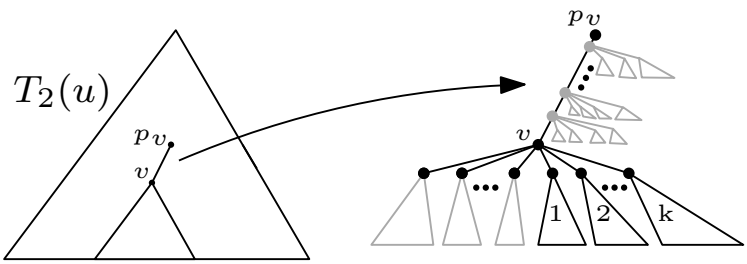


Figure 4 $T_2(u)$: Contracted children subtrees rooted on node v and contracted subtrees rooted on contracted nodes (gray color) in edge (p_v, v).

algorithm for general trees, is the counters that we have to maintain in the contracted versions of T_2, but otherwise, the same analysis from Section 3 holds.

Contraction. The contraction of T_2 with respect to a set of leafs $\Lambda \subseteq L(T_2)$, happens in the exact same way as described in Section 3, i.e. we start by pruning all leafs of T_2 that are not in Λ, then we prune all internal nodes of T_2 with no children, and finally, we contract the nodes that have exactly one child.

Let u be a node of $MCD(b(T_1))$ and C_u the corresponding component of $b(T_1)$. For every such node u we have a contracted version of T_2, denoted $T_2(u)$, where $L(T_2(u)) = L(C_u)$. Like in the binary algorithm, the goal is to augment $T_2(u)$ with counters, so that we can find $\sum_{(v,c')\in T_2} |s''(u,c) \cap s'(v,c')|$ by scanning $T_2(u)$ instead of T_2.

Because of the location where the triplets are anchored, in $T_2(u)$ every leaf that was contracted away, must have a color and be stored in some way. The color of each leaf depends on the type of the component that we have in $b(T_1)$ and the splitting node that is used for that component. For example, in Figure 3 the contracted leafs from X_u will have the red color because like in the binary algorithm $b(T_1)$ is left heavy. The contracted leafs from the children subtrees of u_p in T_1 can either have the color green or black. If u in $b(T_1)$ happens to be orange and part of the orange binary tree that u_p is the root of, then the color must be green, otherwise black. Finally, every leaf that is not in the subtree defined by u_p, and thus is in Y_u, must have the color black. The way we store this information is described in the counting phase below.

Counting. In Figure 4 we illustrate how a node v in $T_2(u)$ can look like. The contracted subtrees are illustrated with the dark gray color. Every such subtree contains some number of red, green and black leafs. The counters that we need to maintain should be so that if v has k children in $T_2(u)$, then we can count all shared triplets that are anchored in every child

edge (including those of the contracted children subtrees) of v in $\mathrm{O}(k)$ time. At the same time, in $\mathrm{O}(1)$ time we should be able to count all shared triplets that are anchored in every child edge of every contracted node that lies on edge (p_v, v). In this way, the counting phase will require $\mathrm{O}(|T_2(u)|)$ time, hence we will get the same bounds like in the binary algorithm.

In v we have the following counters:

- $v_{\mathtt{i}}$: number of leafs with color i (including the contracted leafs) in the subtree of v, where $i \in \{\mathtt{red,\ blue,\ green}\}$.
- $\overline{v}_{\mathtt{black}}$: number of black leafs (including the contracted leafs) not in the subtree of v.

We divide the rest of the counters into two categories. The first category corresponds to the leafs in the contracted children subtrees of v and each counter will be stored in a variable of the form $v_{A.x}$. The second category corresponds to the leafs in the contracted subtrees in edge (p_v, v) and each counter will be stored in a variable of the form $v_{B.x}$.

For the first category A we have the following counters:

- $v_{A.\mathtt{i}}$: total number of leafs with color i in the contracted children subtrees of v, where $i \in \{\mathtt{red,\ green,\ black}\}$.
- $v_{A.\mathtt{red,green}}$: total number of pairs of leafs where one is red, the other is green and one leaf comes from one contracted child subtree of v and the other leaf comes from a different contracted child subtree of v.

While scanning the k children edges of v from left to right, for the child c' that is the mth child of v, we also maintain the following:

- $a_{\mathtt{i}}$: total number of leafs with color i from the first $m-1$ children subtrees, including the contracted children subtrees, where $i \in \{\mathtt{red,\ blue,\ green}\}$.
- $p_{\mathtt{i,j}}$: total number of pairs of leafs from the first $m-1$ children subtrees, including the contracted children subtrees, where one has color i, the other has color j and they both come from different subtrees (one might be contracted and the other non-contracted). We have that $(i,j) \in \{(\mathtt{red}, \mathtt{green}), (\mathtt{red}, \mathtt{blue}), (\mathtt{blue}, \mathtt{green})\}$.
- $t_{\mathtt{red,blue,green}}$: total number of leaf triples from the first $m-1$ children subtrees, including the contracted children subtrees, where one is red, one is blue and one is green, and all three leafs come from different subtrees (some might be contracted, some might be non-contracted).

Every variable is initialized and updated in the following order:

- $(a_{\mathtt{red}}, a_{\mathtt{blue}}, a_{\mathtt{green}}) = (v_{A.\mathtt{red}}, 0, v_{A.\mathtt{green}})$
- $p_{\mathtt{red,green}} = v_{A.\mathtt{red,green}}$
- $p_{\mathtt{red,blue}} = p_{\mathtt{blue,green}} = t_{\mathtt{red,blue,green}} = 0$
- $a_i = a_i + c'_i$, where $i \in \{\mathtt{red,\ blue,\ green}\}$.
- $p_{i,j} = p_{i,j} + a_i \cdot c'_j + a_j \cdot c'_i$, where $(i,j) \in \{(\mathtt{red}, \mathtt{green}), (\mathtt{red}, \mathtt{blue}), (\mathtt{blue}, \mathtt{green})\}$
- $t_{\mathtt{red,blue,green}} = t_{\mathtt{red,blue,green}} + p_{\mathtt{red,green}} \cdot c'_{\mathtt{blue}} + p_{\mathtt{red,blue}} \cdot c'_{\mathtt{green}} + p_{\mathtt{blue,green}} \cdot c'_{\mathtt{red}}$

After finishing scanning the k children edges of v, we can compute the shared triplets that are anchored in every child edge of v (including the children edges pointing to contracted subtrees) as follows: for the total number of shared resolved triplets, denoted $\mathtt{tot}_{\mathtt{A.res}}$, we have that $\mathtt{tot}_{\mathtt{A.res}} = p_{\mathtt{red,blue}} \cdot \overline{v}_{\mathtt{black}}$ and for the total number of shared unresolved triplets, denoted $\mathtt{tot}_{\mathtt{A.unres}}$, we have that $\mathtt{tot}_{\mathtt{unres}} = t_{\mathtt{red,blue,green}}$.

The second category B of counters will help us count triplets involving leafs (contracted and non-contracted) from the subtree of v and leafs from the contracted subtrees rooted on edge (p_v, v). We maintain the following:

- $v_{B.i}$: total number of leafs with color i in all contracted subtrees rooted on edge (p_v, v), where $i \in \{\texttt{red, green, black}\}$.
- $v_{B.\texttt{red,green}}$: total number of pairs of leafs where one is red and the other is green such that one leaf comes from a contracted child subtree of a contracted node v' and the other leaf comes from a different contracted child subtree of the same contracted node v'.
- $v_{B.\texttt{red,black}}$: total number of pairs of leafs where one is red and the other is black such that the red leaf comes from a contracted child subtree of a contracted node v' and the black leaf comes from a contracted child subtree of a contracted node v''. For v' and v'' we have that v'' is closer to v_p than v'.

For the total number of shared unresolved triplets, denoted $\texttt{tot}_{B.\texttt{unres}}$, that are anchored in the children edges of every contracted node that exists in edge (v_p, v), we have that $\texttt{tot}_{B.\texttt{unres}} = v_{\texttt{blue}} \cdot v_{B.\texttt{red,green}}$. For the total number of shared resolved triplets, denoted $\texttt{tot}_{B.\texttt{res}}$, that are anchored in the children edges of every contracted node that exists in edge (v_p, v), we have that $\texttt{tot}_{B.\texttt{res}} = v_{\texttt{blue}} \cdot v_{B.\texttt{red,black}} + v_{\texttt{blue}} \cdot v_{B.\texttt{red}} \cdot (\overline{v}_{\texttt{black}} - v_{B.\texttt{black}})$.

5 Experiments

The implementation of both algorithms was made using the C++ programming language. The source code can be found in `https://github.com/kmampent/CacheTD`.

The Setup The experiments were performed on a machine with 8GB RAM, Intel(R) Core(TM) i5-3470 CPU @ 3.20GHz, 32K L1 cache, 256K L2 cache and 6144K L3 cache. The operating system was Ubuntu 16.04.2 LTS. The compiler used was g++ 5.4 and cmake 3.5.1.

Generating Random Trees. We use two different models for generating input trees. The first model is called the *random model*. A tree T with n leafs in this model is generated as follows:

- Create a binary tree T' with n leafs as follows: start with a binary tree T' with two leafs. Iteratively pick a leaf l uniformly at random. Make l an internal node by appending a left child node and a right child node to l, thus increasing the number of leafs in T' by exactly 1.
- With probability p contract every internal node u of T', i.e make the children of u be the children of u's parent and remove u.

The second model is called the *skewed* model. In this model, we can control more directly the shape of the input trees. A tree T with n leafs in this model is generated as follows:

- Create a binary tree T' with n leafs as follows: let $0 \leq \alpha \leq 1$ be a parameter, u some internal node in T, l and r the left and right children of u, and $T(u)$, $T(l)$ and $T(r)$ the subtrees rooted on u, l and r respectively. Create T' so that for every internal node u we have $\frac{|T(l)|}{|T(u)|} \approx \alpha$, i.e. $|T_l| = \max(1, \min(\lfloor a \cdot n \rfloor, n-1))$ and $|T_r| = 1 - |T_l|$, where $|T_l|$ and $|T_r|$ are the number of leafs in $T(l)$ and $T(r)$ respectively.
- With probability p contract every internal node u of T' like in the random model.

In both models, after creating T, we shuffle the leaf labels by using `std::shuffle`[1] together with `std::default_random_engine`[2].

[1] `http://www.cplusplus.com/reference/algorithm/shuffle/`
[2] `http://www.cplusplus.com/reference/random/default_random_engine/`

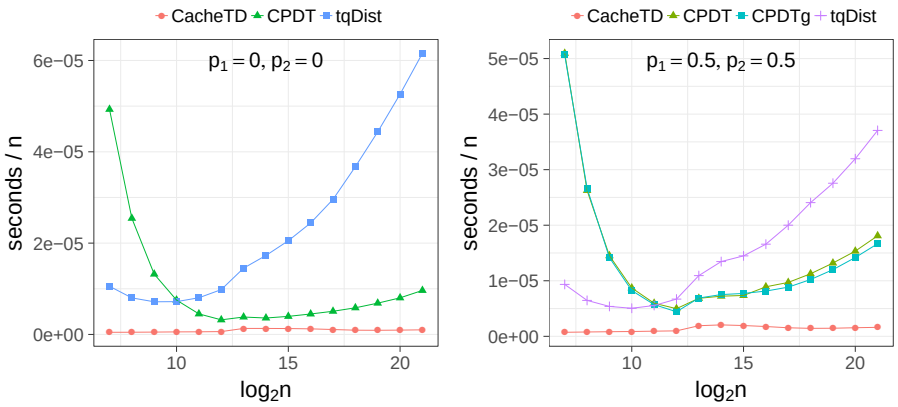


Figure 5 Time performance in the random model.

Implementations Tested. Let p_1 and p_2 denote the contraction probability of T_1 and T_2 respectively. When $p_1 = p_2 = 0$, the trees T_1 and T_2 are binary trees, so in our experiments we use the algorithm from Section 3. In all other cases, the algorithm from Section 4 is used. Note that the algorithm from Section 4 can handle binary trees just fine, however there is an extra overhead (factor 1.8 slower) compared to the algorithm from Section 3 that comes due to the additional counters that we have to maintain for the contractions of T_2.

We compared our implementation with previous implementations of [10] and [14, 3] available at `http://sunflower.kuicr.kyoto-u.ac.jp/~jj/Software/` and `http://users-cs.au.dk/cstorm/software/tqdist/` respectively. The implementation of the $O(n \log^3 n)$ algorithm in [10] has two versions, one that uses `unordered_map`[3], which we refer to as `CPDT`, and another that uses sparsehash[4], which we refer to as `CPDTg`. For binary input trees the hash maps are not used, thus `CPDT` and `CPDTg` are the same. The tqdist library [15], which we will refer to as `tqDist`, has an implementation of the binary $O(n \log^2 n)$ algorithm from [14] and the general $O(n \log n)$ algorithm from [3]. If the two input trees are binary the $O(n \log^2 n)$ algorithm is used. We will refer to our new algorithm as `CacheTD`.

Statistics. We measured the execution time of the algorithms with the `clock_gettime` function in C++. Due to the different parser implementations, we do not consider the time taken to parse the input trees. We used the PAPI library[5] for statistics related to L1, L2 and L3 cache accesses and misses. Finally, we count the space of the algorithms by considering the *Maximum resident set size* returned by `/usr/bin/time -v`.

Results. The experiments are divided into two parts. In the first part, we look at how the algorithms perform when the memory requirements do not exceed the available main memory (8G RAM). In the second part, we look at how they perform when the memory requirements exceed the available main memory (by limiting the available RAM to the operating system to be 1GB), thus forcing the operating system to use the swap space, which in turn can for large enough input trees yield the very expensive disk I/Os.

[3] `http://en.cppreference.com/w/cpp/container/unordered_map`
[4] `https://github.com/sparsehash/sparsehash`
[5] `http://icl.utk.edu/papi/`

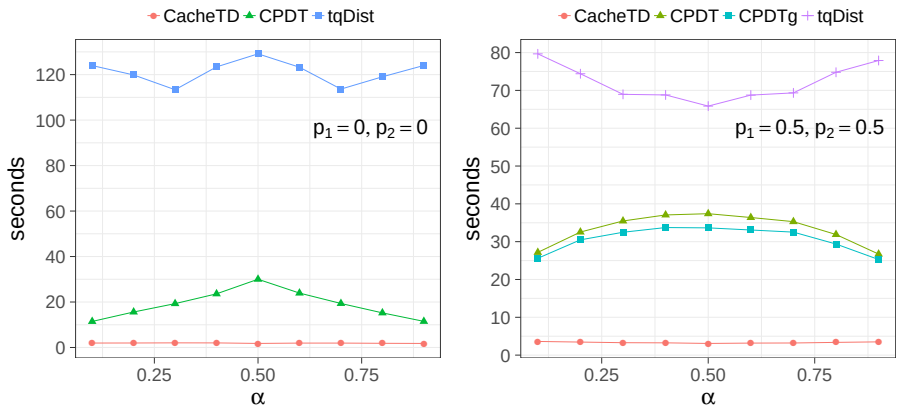


Figure 6 How the alpha parameter affects running time ($n = 2^{21}$).

Table 1 Time performance when limiting the available RAM to be 1GB. For both tables we have $\alpha = 0.5$. For the left table we have $p_1 = p_2 = 0$ and for the right table $p_1 = p_2 = 0.5$.

n	CPDT/CPDTg	tqDist	CacheTD
2^{17}	0m:01s	0m:08s	0m:01s
2^{18}	0m:02s	3m:10s	0m:01s
2^{19}	0m:05s	2h:16m	0m:01s
2^{20}	0m:34s	-	0m:01s
2^{21}	7h:09m	-	0m:03s
2^{22}	-	-	0m:35s
2^{23}	-	-	10m:09s
2^{24}	-	-	43m:52s

n	CPDT	CPDTg	tqDist	CacheTD
2^{17}	0m:01s	0m:01s	0m:03s	0m:01s
2^{18}	0m:03s	0m:03s	1m:18s	0m:01s
2^{19}	0m:10s	0m:07s	19m:02s	0m:01s
2^{20}	1h:58m	6h:32m	>10h	0m:02s
2^{21}	-	-	-	0m:56s
2^{22}	-	-	-	4m:11s
2^{23}	-	-	-	24m:44s
2^{24}	-	-	-	2h:13m

RAM experiments. For the random model, in Figure 5 we illustrate a time comparison of all implementations for trees of up to 2^{21} leafs ($\sim$ 2 million) with varying contraction probabilities. Every data point is the average of 10 runs. In all cases `CacheTD` achieves the best time performance. The space and L1/L2/L3 cache performance of `CacheTD` is the best as well (see [4]). For the skewed model, in Figure 6 we plot the alpha parameter against the execution time of the algorithms, when $n = 2^{21}$. The alpha parameter has the least effect on `CacheTD`, with the maximum running time being only a factor of 1.1 larger than the minimum. From Section 2, `CPDT` and `CPDTg` use the heavy light decomposition for T_2. When α approaches 0 or 1, the number of heavy paths that will be updated because of a leaf color change decreases, thus the total number of operations of the algorithm decreases as well (see [4]).

I/O experiments. The results are included in Table 1. Each cell contains the execution time (including the waiting time due to disk I/Os). For this experiment we used the `time` function of Ubuntu and thus also considered the time taken to parse the input trees. Each cell contains the result of 1 run and for input trees with 2^{23} and 2^{24} leafs we used the 128 bit implementation of the new algorithms to avoid numeric overflows. The exact running times vary from run to run, but the general outcome is the same: unlike `CacheTD`, the performance of `CPDT`, `CPDTg` and `tqDist` deteriorates significantly the moment they start performing disk I/Os. More elaborate I/O experiments can be found in the arXiv version of the paper [4].

References

1. M.S. Bansal, J. Dong, and D. Fernández-Baca. Comparing and aggregating partially resolved trees. *Theoretical Computer Science*, 412(48):6634 – 6652, 2011.
2. V. Berry and O. Gascuel. Inferring evolutionary trees with strong combinatorial evidence. *Theoretical Computer Science*, 240(2):271–298, 2000.
3. G.S. Brodal, R. Fagerberg, C.N.S. Pedersen, T. Mailund, and A. Sand. Efficient algorithms for computing the triplet and quartet distance between trees of arbitrary degree. *24th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1814–1832, 2013.
4. G.S Brodal and K. Mampentzidis. Cache oblivious algorithms for computing the triplet distance between trees. *Computing Research Repository*, abs/1706.10284, 2017.
5. D.E. Critchlow, D.K. Pearl, and C.L. Qian. The Triples Distance for Rooted Bifurcating Phylogenetic Trees. *Systematic Biology*, 45(3):323, 1996.
6. A.J. Dobson. Comparing the shapes of trees. *Combinatorial Mathematics III. Lecture Notes in Mathematics*, pages 95–100, 1975.
7. G.F. Estabrook, F.R. McMorris, and C.A. Meacham. Comparison of undirected phylogenetic trees based on subtrees of four evolutionary units. *Systematic Zoology*, 34(2):193–200, 1985.
8. M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *40th Annual IEEE Symposium on Foundations of Computer Science*, pages 285–297, 1999.
9. M.K. Holt, J. Johansen, and G.S. Brodal. On the scalability of computing triplet and quartet distances. *16th Workshop on Algorithm Engineering and Experiments*, pages 9–19, 2014.
10. J. Jansson and R. Rajaby. A more practical algorithm for the rooted triplet distance. *International Conference on Algorithms for Computational Biology*, pages 109–125, 2015.
11. J. Jansson and R. Rajaby. A More Practical Algorithm for the Rooted Triplet Distance. *Journal of Computational Biology*, 24(2):106–126, 2017.
12. D.F. Robinson and L.R. Foulds. Comparison of phylogenetic trees. *Mathematical Biosciences*, 53(1):131–147, 1981.
13. N. Saitou and M. Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4(4):406, 1987.
14. A. Sand, G.S. Brodal, R. Fagerberg, C.N.S. Pedersen, and T. Mailund. A practical $O(n \log^2 n)$ time algorithm for computing the triplet distance on binary trees. *BMC Bioinformatics*, 14(2):S18, 2013.
15. A. Sand, M.K. Holt, J. Johansen, G.S. Brodal, T. Mailund, and C.N.S. Pedersen. tqdist: A library for computing the quartet and triplet distances between binary or general trees. *Bioinformatics*, 30(14):2079, 2014.