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

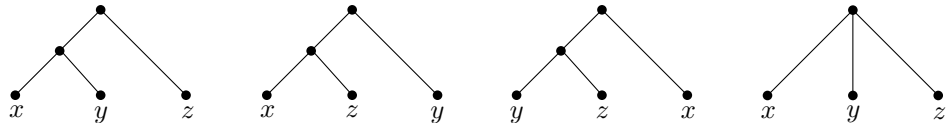
Background. Trees are data structures that are often used to represent relationships. For example in the field of Biology, a tree can be used to represent evolutionary relationships, with the leafs corresponding to species that exist today, and internal nodes to ancestor species that existed in the past. For a fixed set of n species, different data (e.g. DNA, morphological) or construction methods (e.g. Q* [3], neighbor joining [14]) 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 distance measures have been proposed in the past to compare two trees. A class of them includes distance measures that are based on how often certain features are different in the two trees. Common distance measures of this kind are the Robinson-Foulds distance [13],

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

25th Annual European Symposium on Algorithms (ESA 2017).
Editors: Kirk Pruhs and Christian Sohler; Article No. 1; pp. 1:1–1:29

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



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

Figure 1 Triplet topologies.

the triplet distance [7] and the quartet distance [8]. The Robinson-Foulds distance counts how many leaf bipartitions are different, where a bipartition in a given tree is generated by removing a single edge from the tree. The triplet distance is only defined for rooted trees, and counts how many leaf triples induce different topologies in the two trees. The counterpart of the triplet distance for unrooted trees, is the quartet distance, which counts how many leaf quadruples induce different topologies in the two trees.

Algorithms exist that can efficiently compute these distance measures. The Robinson-Foulds distance can be optimally computed in $O(n)$ time [6]. The triplet distance can be computed in $O(n \log n)$ time [4]. The quartet distance can be computed in $O(dn \log n)$ time [4], where d is the maximal degree of any node in the two input trees.

Note that the above bounds are in the RAM model. Previous work did not consider any other models, for example external memory models like the I/O model [1] and the cache oblivious model [9]. Typically when talking about algorithms for external memory models, one might (sometimes incorrectly) think of algorithms that have to deal with large amounts of data. So any practical improvement that comes from an algorithm that scales to external memory compared to an equivalent that does not, can only be noticed if the inputs are large. However, this is not necessarily the case for cache oblivious algorithms. A cache oblivious algorithm, if built and implemented correctly, can take advantage of the L1, L2 and L3 caches that exist in the vast majority of computers these days and give a significant performance boost even for small inputs.

A trivial modification of the algorithm in [6], can give a cache oblivious algorithm for computing the Robinson-Foulds distance that achieves the sorting bound by requiring $O(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{M})$ I/Os. For the triplet and quartet distance measures, no such trivial modifications exist.

In this paper we focus on the triplet distance computation and present the first non trivial algorithms for computing the triplet distance between two rooted trees, that for the first time for this problem, also scale to external memory.

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. The notation $xy|z$ is used to describe a triplet where 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. Note that the triplet $xy|z$ is the same as the triplet $yx|z$ because T is considered to be unordered. Similarly, notation xyz is used to describe a triplet for which every pair of leafs has the same lowest common ancestor. This triplet can only appear if we allow nodes with degree three or larger in T.

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 $D(T_1, T_2) + S(T_1, T_2) = \binom{n}{3}$.

Previous and new results for computing the triplet distance are shown in the table below. Note that the papers [5, 2, 15, 4, 11] do not provide an analysis of the algorithms in the cache oblivious model, 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.* [2]	$O(n^2)$	$O(n^2)$	$O(n^2)$	yes
2013	Brodal *et al.* [15]	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(n)$	no
2013	Brodal *et al.* [4]	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	yes
2015	Jansson *et al.* [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

Related Work. The triplet distance was first suggested as a method of comparing the shapes of trees by Dobson in 1975 [7]. The first non-trivial algorithmic result dates back to 1996, when Critchlow *et al.* [5] proposed an $O(n^2)$ algorithm that however works only for binary trees. Bansal *et al.* [2] introduced an $O(n^2)$ algorithm that works for general (binary and non-binary) trees. Both of these algorithms use $O(n^2)$ space. Brodal *et al.* [15] introduced a new $O(n^2)$ algorithm using only $O(n)$ space for the case of binary trees that they showed how to optimize to reduce the time to $O(n \log^2 n)$. This algorithm was also implemented and shown to be the most efficient in practice. Soon after, Brodal *et al.* [4] managed to extend the $O(n \log^2 n)$ algorithm to work for general trees, and at the same time brought the time down to $O(n \log n)$ but now with the space increased to $O(n \log n)$. Note that the space for binary trees was still $O(n)$. The algorithms from [15] and [4] were implemented and added to the library tqDist [16]. Interestingly, it was shown in [10] that for binary trees the $O(n \log^2 n)$ algorithm had a better practical performance than the $O(n \log n)$ algorithm. Jansson *et al.* [11, 12] showed that an even slower theoretically algorithm requiring worst case $O(n \log^3 n)$ time and $O(n \log n)$ space could give the best practical performance, both for binary and non-binary trees. A detailed survey over previous results until 2013 can be found in [17].

Contribution. 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.

Outline of the Paper. In Section 2 we provide an overview of previous approaches. In Section 3 we describe the new algorithm for the case where T_1 and T_2 are binary trees. In Section 4 we extend it to work for general trees. In Section 5 we provide some details related to our implementation of the new algorithms. In Section 6 we include some experiments

illustrating the efficiency of the new algorithms compared to the previously known algorithms for this problem. Finally, in Section 7 we provide some concluding remarks.

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.* [2] exploit the shared leafs between subtrees and essentially 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 has an elegant dynamic programming solution.

The $O(n^2)$ Algorithm for Binary Trees in [15]. The algorithm for binary trees in [15] is the basis for all subsequent improvements [15, 4, 11], including ours as well, so we will describe it in more detail here. The dependency that was exploited is the same as in [2] 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 $O(n^2)$ the value $|s(u) \cap s(v)|$ must be computed in $O(1)$ 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.* [15] applied the *smaller half trick*, which specifies a depth first search order to visit the nodes u of T_1, so that each leaf in T_1 changes color at most $O(\log n)$ times. To count shared triplets efficiently without scanning T_2 completely for each node u in T_1, 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 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 [15] the HDT is a binary tree of height $O(\log n)$ and every update can be done in a leaf to root path traversal in the HDT, which in total gives $O(n \log^2 n)$ time. In [4] 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 $O(n \log n)$. Finally, in [11] as an HDT the so called *heavy-light tree decomposition* is used. Note that the only difference in all $O(n \operatorname{polylog} n)$ results that are available until now is the type of HDT used.

In terms of external memory efficiency, every $O(n \operatorname{polylog} n)$ 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.

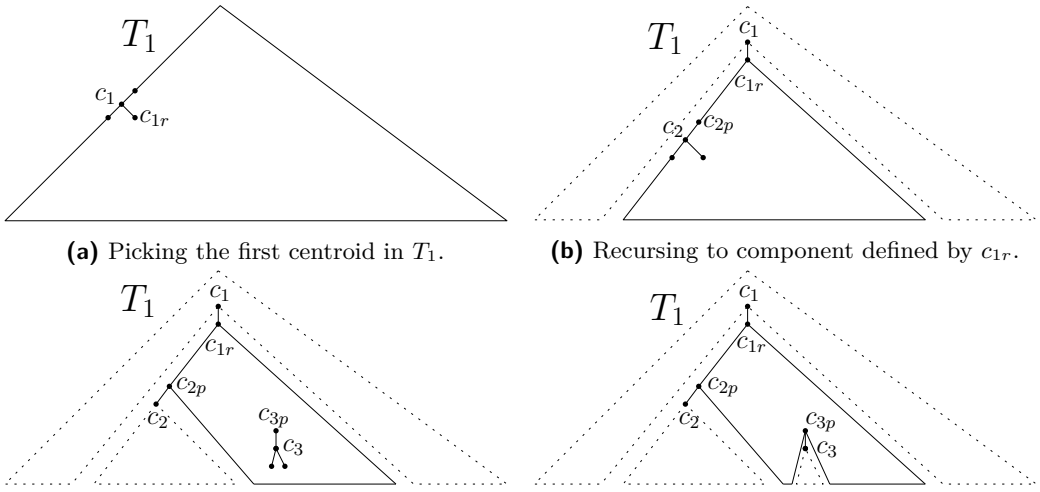
3 The New Algorithm for Binary Trees

Overview. We will use the $O(n^2)$ 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 scanning becomes the basic primitive in the algorithm. To make our algorithm I/O efficient, all that remains to be done is 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.

We do this 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. The *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 generate a level of $CD(T)$ in $O(|T|)$ time. 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.



(a) Picking the first centroid in T_1.
(b) Recursing to component defined by c_{1r}.
(c) Recursing to component defined by c_{2p}.
(d) Recursing to component defined by c_{3p}.

Figure 2 Generating a component in $CD(T_1)$ that has two edges from below. The black polygon is the component.

▶ **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). An example with a component that has two edges from below can be seen in Figure 2. It is trivial to see that by following the same pattern of generating components we can have a component with an arbitrary number of edges from below. 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. A $MCD(T)$ is built recursively as follows: If a component C has no edge from below, we select the centroid c of C 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 presented 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 $MCD(T)$.*

Proof. In $MCD(T)$ if a component C does not have an edge from below then the centroid of C is used as a splitting node, thus generating three components C_l, C_r and C_p such that $|C_l| \leq \frac{|C|}{2}$, $|C_r| \leq \frac{|C|}{2}$ and $|C_p| \leq \frac{|C|}{2}$. Otherwise, C has one edge (x, y) from below, with x being the node that is part of C. Let c be the centroid of C. We have to consider the following two cases: if c happens to be the lowest common ancestor of c and x, then our algorithm will split C according to the actual centroid, so we will have that $|C_l| \leq \frac{|C|}{2}$, $|C_r| \leq \frac{|C|}{2}$ and $|C_p| \leq \frac{|C|}{2}$. Otherwise, the splitting node will produce components C_l, C_r, and C_p such that $|C_l| + |C_p| \leq \frac{|C|}{2}$ and $|C_r| \geq \frac{|C|}{2}$. From the first inequality we have that $|C_l| \leq \frac{|C|}{2}$ and $|C_p| \leq \frac{|C|}{2}$. Notice however that C_r is going to be a component corresponding

to a complete subtree of T, so it will have no edges from below. This means that in the next recursion level when working with C_r the actual centroid of C_r will be chosen as a splitting node, then in the next level the three components produced from C_r will be such that their sizes are at most half the size of C. From the analysis given so far, it becomes clear that when we have a component of size $|C|$ with one edge from below, then we will need at most 2 levels in $MCD(T)$ before producing components all of which will have a guaranteed size at most $\frac{|C|}{2}$. Hence we proved the given statement. ◀

It follows,

▶ **Theorem 3.** *For any rooted binary tree T with n leafs, building $MCD(T)$ takes $O(n \log n)$ 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 $O(|\Lambda|)$ 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), 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 level i, which correspond to a set of disjoint connected components in T_1, we can reuse the contractions of level $i-1$. This means that we have to spend $O(n)$ time to generate the contractions of level i, so to generate all contractions we need $O(n \log n)$ time. Note that by explicitly storing all contractions, we will also need to use $O(n \log n)$ space. For our problem, we traverse $MCD(T_1)$ in a depth first search 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. Lemma 4 states that the size of this stack is always $O(n)$.

▶ **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 $\sum_{i=1}^{k} |T_2(u_i)| = O(n)$.*

Proof. For the root u_1 we have that $T_2(u_1) = T_2$ so $|T_2(u_1)| \leq 2n$. From the proof of Lemma 2 we have that for every component of size x, we need at most two levels in $MCD(T_1)$ before producing components all of which will have a guaranteed size of at most $\frac{x}{2}$. This means that $\sum_{i=1}^{k} |T_2(u_i)| \leq 2n + 2n + \frac{2n}{2} + \frac{2n}{2} + \frac{2n}{4} + \frac{2n}{4} + ... + \frac{2n}{2^i} + \frac{2n}{2^i} + ... = 2 \sum_{j=0}^{\infty} \frac{2n}{2^j} \leq 8n = O(n)$. ◀

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 $O(n \log n)$. 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 $O(n^2)$ algorithm (Section 2) but now instead of scanning T_2 we scan $T_2(u)$.
- *C_u has one edge from below.*

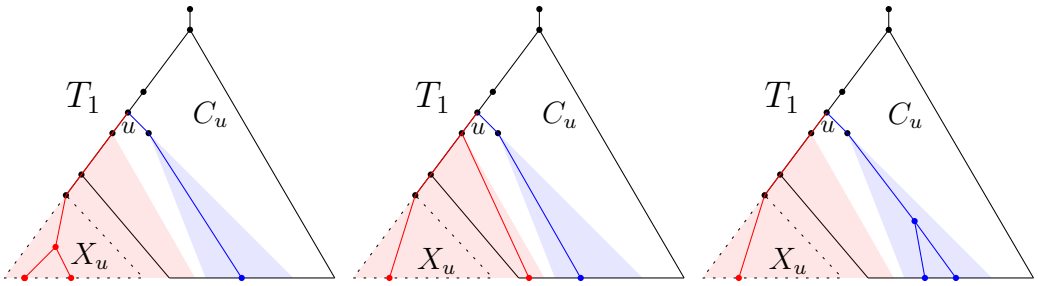


Figure 3 $MCD(T_1)$: Triplets that can be anchored in u with the leafs not being in the component C_u.

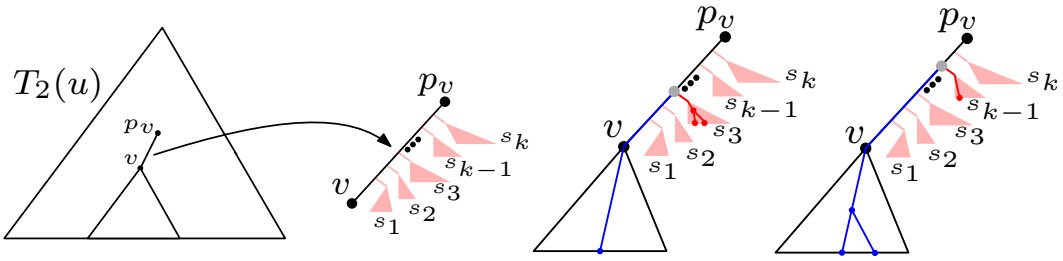


Figure 4 Contracted subtrees on edges in $T_2(u)$ and shared triplets rooted on contracted nodes.

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. Note that because in the preprocessing step T_1 was made to be 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. An illustration can be found in Figure 3. Acting in the exact same manner as in the previous case is not sufficient because we need to count these triplets 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 (see Figure 4). 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 was made to be left heavy). For every node v in $T_2(u)$ the counters that we have are the following:

- $v_{\texttt{red}}$: the total number of red leafs in the subtree of v (including those coming from X_u).
- $v_{\texttt{blue}}$: the total number of blue leafs in the subtree of v.
- v_{ts}: the total number of red leafs in $s_1, s_2, ..., s_k$.

- v_{ps}: the total number of pairs of red leafs from $s_1, s_2, ..., s_k$ 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 $p+x+1$, making T_1 left heavy requires $O(n/B)$ 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. Finding the splitting node of a component C_u requires $O(|C_u|/B)$ 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 to initialize a large enough 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

Unlike a binary tree, a general tree allows having internal nodes with an arbitrary number of children. By introducing four colors, we will show that we can count the shared triplets in the two trees. We start by describing the new $O(n^2)$ algorithm for general trees. We will then show how we can use the same ideas presented in the previous section to reduce the time to $O(n \log n)$.

4.1 $O(n^2)$ Algorithm for General Trees.

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 the edge (w, c). Let $s'(w, c)$ to be a set containing all triplets anchored in edge (w, c). For the number of triplets $S(T_1, T_2)$ we have

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

To count shared triplets efficiently we use the following coloring procedure: in T_1 color the leafs of every child subtree of u to the left of c red, the leafs of the child subtree of c blue, the leafs of every child subtree to the right of c green and give the color black to every other leaf. We then transfer this coloring to the leafs in T_2. For the resolved triplet $ij|k$, i corresponds to the red color, j corresponds to the blue color and k corresponds to the black color. For

the unresolved triplet ijk, i corresponds to the red color, j corresponds to the blue color and k corresponds to the green color.

Suppose that the node v in T_2 has k children. We will compute all shared triplets that are anchored in the k edges that point to the children of v in $O(k)$ time, giving us the $O(n^2)$ total time that we want. In v we have the following counters:

- $v_{\texttt{red}}$: total number of red leafs in the subtree of v.
- $v_{\texttt{blue}}$: total number of blue leafs in the subtree of v.
- $v_{\texttt{green}}$: total number of green leafs in the subtree of v.
- $\overline{v}_{\texttt{black}}$: total number of black leafs not in the subtree of v.

While scanning the k children edges of v from left to right, for a child c' that is the $m-th$ child of v, we want to maintain the following counters:

- $a_{\texttt{red}}$: total number of red leafs from the first $m-1$ children subtrees.
- $a_{\texttt{blue}}$: total number of blue leafs from the first $m-1$ children subtrees.
- $a_{\texttt{green}}$: total number of green leafs from the first $m-1$ children subtrees.
- $p_{\texttt{red,green}}$: total number of pairs of leafs from the first $m-1$ children subtrees, where one is red, the other is green and they both come from different children subtrees.
- $p_{\texttt{red,blue}}$: total number of pairs of leafs from the first $m-1$ children subtrees, where one is red, the other is blue and they both come from different children subtrees.
- $p_{\texttt{blue,green}}$: total number of pairs of leafs from the first $m-1$ children subtrees, where one is blue, the other is green and they both come from different children subtrees.
- $t_{\texttt{red,blue,green}}$: the total number of leaf triples from the first $m-1$ children subtrees, where one is red, one is blue, one is green, and all three leafs come from different children subtrees.

Before beginning the scanning process of the children edges of v, every variable is initialized to 0. Then for the child c' every variable can then be updated in $O(1)$ time as follows:

- $a_{\texttt{red}} = a_{\texttt{red}} + c'_{\texttt{red}}$
- $a_{\texttt{blue}} = a_{\texttt{blue}} + c'_{\texttt{blue}}$
- $a_{\texttt{green}} = a_{\texttt{green}} + c'_{\texttt{green}}$
- $p_{\texttt{red,green}} = p_{\texttt{red,green}} + a_{\texttt{green}} \cdot c'_{\texttt{red}} + a_{\texttt{red}} \cdot c'_{\texttt{green}}$
- $p_{\texttt{red,blue}} = p_{\texttt{red,blue}} + a_{\texttt{blue}} \cdot c'_{\texttt{red}} + a_{\texttt{red}} \cdot c'_{\texttt{blue}}$
- $p_{\texttt{blue,green}} = p_{\texttt{blue,green}} + a_{\texttt{green}} \cdot c'_{\texttt{blue}} + a_{\texttt{blue}} \cdot c'_{\texttt{green}}$
- $t_{\texttt{red,blue,green}} = t_{\texttt{red,blue,green}} + p_{\texttt{red,green}} \cdot c'_{\texttt{blue}} + p_{\texttt{red,blue}} \cdot c'_{\texttt{green}} + p_{\texttt{blue,green}} \cdot c'_{\texttt{red}}$

After finishing scanning the k children edges of v, we can compute the shared triplets that are anchored in every child edge of v as follows: for the total number of shared resolved triplets, denoted $\texttt{tot}_{\texttt{res}}$, we have that $\texttt{tot}_{\texttt{res}} = p_{\texttt{red,blue}} \cdot \overline{v}_{\texttt{black}}$ and for the total number of shared unresolved triplets, denoted $\texttt{tot}_{\texttt{unres}}$, we have that $\texttt{tot}_{\texttt{unres}} = t_{\texttt{red,blue,green}}$. Now we are ready to describe the $O(n \log n)$ algorithm.

4.2 $O(n \log n)$ Algorithm for General Trees.

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. The k edges that connect w to its children will be replaced by a binary tree, the leafs of which will be the k children of w, and each internal node will have the color *orange*. Every such binary tree

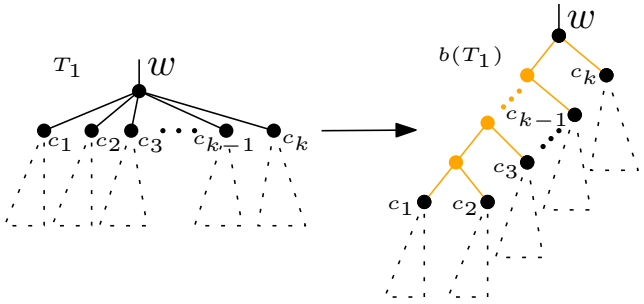


Figure 5 Transforming T_1 into $b(T_1)$.

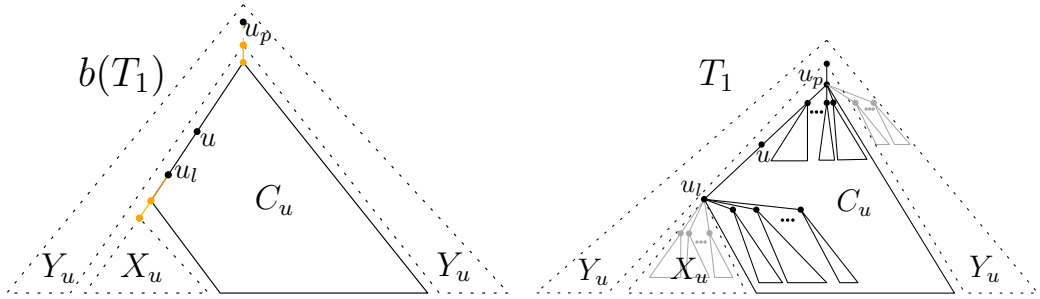


Figure 6 How a component in $b(T_1)$ translates to a component in T_1.

will be built in a way so that every orange node is on the left most path that starts from w, and the left most path is part of the heavy path starting from node w in $b(T_1)$, i.e. $b(T_1)$ is left heavy (see Figure 5). Note that the number of orange nodes added because of node w is exactly $k-2$ which is less than the number of w's children in T_1, so the total number of nodes in $b(T_1)$ is still $O(n)$.

We can generate the colorings of the $O(n^2)$ algorithm by using $b(T_1)$ instead of T_1. Let u be some internal node of $b(T_1)$ and c be the right child of u. The leafs of $b(T_1)$ are colored according to edge (u, c) as follows: the left subtree of u is colored red, the right subtree of u is colored blue. If u is an orange node, then the remaining leafs in the orange binary tree that u is part of are colored green. All other leafs of $b(T_1)$ are colored black.

Let $u_{\texttt{root}}$ be the root node of the orange tree that u is part of. If u is not part of an orange tree, then $u_{\texttt{root}}$ is the parent of u. The nodes $u_{\texttt{root}}$ and c exist in T_1 and $(u_{\texttt{root}}, c)$ is an edge in T_1. So coloring the leafs of $b(T_1)$ according to edge (u, c) is equivalent to coloring the leafs of T_1 according to edge $(u_{\texttt{root}}, c)$. Every right edge of $b(T_1)$ corresponds to exactly one edge of T_1, coloring the leafs of $b(T_1)$ according to this right edge as described above will produce exactly the same coloring as the coloring according to the corresponding edge in T_1 as described in the $O(n^2)$ algorithm. This implies correctness, i.e. in the $O(n^2)$ algorithm using $b(T_1)$ instead of T_1 will yield $S(T_1, T_2)$.

For notation purposes, for the edge (u, c) in $b(T_1)$, we let $s''(u, c)$ be the set of triplets that are anchored in the edge (u_{root}, c) in T_1, i.e. $s''(u, c) = s'(u_{\texttt{root}}, c)$. For the number of shared triplets we then have:

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

The reason behind this transformation 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 do 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 the children of every node u in $MCD(b(T_1))$ are visited from left to right.

In Figure 6 we can see that a component C_u of $b(T_1)$ structurally looks like a component of T_1 in the binary algorithm of Section 3. However, the edges crossing its boundary can now be orange edges as well, which in T_1 that now is a general tree, translates to more than one consecutive subtrees.

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.

Contraction. The contraction of T_2 with respect to a set of leafs $\Lambda \subset L(T_2)$, 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 $L(T_2(u)) = L(C_u)$. Just 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 6 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 part of the orange 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.

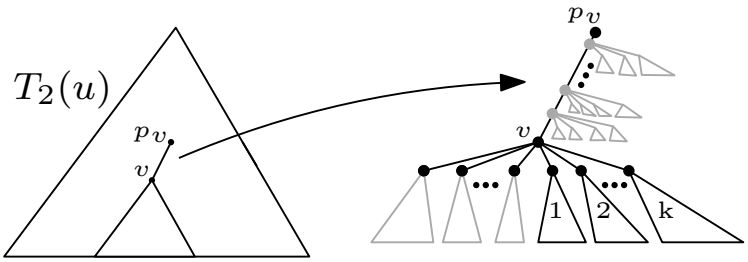


Figure 7 $T_2(u)$: Contracted children subtrees on node v and contracted subtrees on contracted nodes (gray color) in edge (p_v, v).

Counting. In Figure 7 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 $O(k)$ time. At the same time, in $O(1)$ 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 $O(|T_2(u)|)$ time, hence we will get the same bounds like in the binary algorithm.

In v we have the following counters:

- $v_{\texttt{red}}$: total number of red leafs (including the contracted leafs) in the subtree of v.
- $v_{\texttt{blue}}$: total number of blue leafs in the subtree of v.
- $v_{\texttt{green}}$: total number of green leafs (including the contracted leafs) in the subtree of v.
- $\overline{v}_{\texttt{black}}$: total 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 $v_{A.\{\}}$. The second category corresponds to the leafs in the subtrees in edge (p_v, v) and each counter will be stored in a variable of the form $v_{B.\{\}}$.

For the first category we have the following counters:

- $v_{A.\texttt{red}}$: total number of red leafs in the contracted children subtrees of v.
- $v_{A.\texttt{green}}$: total number of green leafs in the contracted children subtrees of v.
- $v_{A.\texttt{black}}$: total number of black leafs in the contracted children subtrees of v.
- $v_{A.\texttt{red,green}}$: 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 edges of the children of v from left to right, for a child c' that is the $m-th$ child of v, we also maintain the following:

- $a_{\texttt{red}}$: total number of red leafs from the first $m-1$ children subtrees, including the contracted children subtrees.
- $a_{\texttt{blue}}$: total number of blue leafs from the first $m-1$ children subtrees.
- $a_{\texttt{green}}$: total number of green leafs from the first $m-1$ children subtrees, including the contracted children subtrees.
- $p_{\texttt{red,green}}$: total number of pairs of leafs from the first $m-1$ children subtrees and all contracted children subtrees, where one is red, the other is green and they both come from different subtrees (one might be contracted and the other non-contracted).
- $p_{\texttt{red,blue}}$: total number of pairs of leafs from the first $m-1$ children subtrees and all contracted children subtrees, where one is red, the other is blue and they both come from different subtrees (one might be contracted and the other non-contracted).
- $p_{\texttt{blue,green}}$: total number of pairs of leafs from the first $m-1$ children subtrees and all contracted children subtrees, where one is blue, the other is green and they both come from different subtrees (one might be contracted and the other non-contracted).
- $t_{\texttt{red,blue,green}}$: the total number of leaf triples from the first $m-1$ children subtrees and all 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).

For every child c', each variable can be updated in $O(1)$ in exactly the same manner like in the $O(n^2)$ algorithm. The main difference is the values of the variables before we begin scanning the children edges of v. Each variable is initialized as follows:

- $a_{\texttt{red}} = v_{A.\texttt{red}}$

- $a_{\texttt{blue}} = 0$
- $a_{\texttt{green}} = v_{A.\texttt{green}}$
- $p_{\texttt{red,green}} = v_{A.\texttt{red,green}}$
- $p_{\texttt{red,blue}} = p_{\texttt{blue,green}} = t_{\texttt{red,blue,green}} = 0$

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 $\texttt{tot}_{\texttt{A.res}}$, we have that $\texttt{tot}_{\texttt{A.res}} = p_{\texttt{red,blue}} \cdot \overline{v}_{\texttt{black}}$ and for the total number of shared unresolved triplets, denoted $\texttt{tot}_{\texttt{A.unres}}$, we have that $\texttt{tot}_{\texttt{unres}} = t_{\texttt{red,blue,green}}$.

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

- $v_{B.\texttt{red}}$: total number of red leafs in all contracted subtrees in edge (p_v, v).
- $v_{B.\texttt{green}}$: total number of green leafs in all contracted subtrees in edge (p_v, v).
- $v_{B.\texttt{black}}$: total number of black leafs in all contracted subtrees in edge (p_v, v).
- $v_{B.\texttt{red,green}}$: total number of pairs of leafs where one is red and the other is green such that one leaf comes from one 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 one contracted child subtree of a contracted node v' and the black leaf comes from one 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 $\texttt{tot}_{B.\texttt{res}} = v_{\texttt{blue}} \cdot v_{B.\texttt{red,black}} + v_{\texttt{blue}} \cdot v_{B.\texttt{red}} \cdot (\overline{v}_{\texttt{black}} - v_{B.\texttt{black}})$.

Scaling to External Memory. Just like in Section 3, we store T_1 in an array of size $2n-1$ by using a preorder layout, that is 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 build $b(T_1)$ in $\mathrm{O}(n/B)$ I/Os and store it in a different array following the same layout as T_1. Making $b(T_1)$ left heavy requires $\mathrm{O}(n/B)$ I/Os and the preprocessing step that changes the labels of the leafs in $b(T_1)$ and T_2 can be done in $\mathrm{O}(\frac{n}{B}\log_2 \frac{n}{M})$ I/Os with a cache oblivious sorting routine, e.g. using merge sort. Finding the splitting node of a component C_u requires $\mathrm{O}(|C_u|/B)$ I/Os, so in total the number of I/Os spent from processing T_1 and $b(T_1)$ becomes $\mathrm{O}(\frac{n}{B}\log_2 \frac{n}{M})$.

The proof of Lemma 4 can be trivially modified to apply to general trees as well, so we use it to initialize a large enough array that can fit the contractions that we need to remember while traversing $MCD(b(T_1))$. 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 $\mathrm{O}(|T_2(u)|/B)$ I/Os.

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

5 Implementation

Figure 8 Implementation overview.

The implementation of both algorithms was made using the C++ programming language. A high level overview is illustrated in Figure 8. The source code can be found in `https://github.com/kmampent/CacheTD`.

5.1 Input

The two input trees T_1 and T_2 are stored in two separate text files following the Newick format. Both trees have n leafs and the label of each leaf is assumed to be a number in {1,2,...,n}. Two leafs can not have the same label.

5.2 Parser

The parser receives the files that store T_1 and T_2 in Newick format, and returns T_1 and T_2 but now with T_1 stored inside an array following the preorder layout and T_2 inside an array following the postorder layout. The parser requires $O(n)$ time and $O(n/B)$ I/Os.

5.3 Algorithm

Having T_1 and T_2 stored in the right layouts, we can now proceed with the main part of the algorithm. Both implementations (binary, general) follow the same approach. There exists an *initialization* phase and a *distance computation* phase.

Initialization. In the initialization phase the first component of T_1 is built, and the corresponding contracted version of T_2, from now on referred to as *corresponding component* of T_2, is built as well. At the end of this phase, the first component of T_1 will be stored in the preorder layout in a new array (different than the one produced by the parser) and similarly the corresponding component of T_2, which however will be stored in the postorder layout instead.

Note that the first component of T_1 is T_1 and the component of T_2 is T_2. The main difference lies in the representation of the nodes of T_1 and T_2 that now have to store different variables/counters. The reason why we do not immediately build these components while parsing the input, is because we want to make the experiments as fair as possible. The main focus in the experiments is comparing the algorithms and not the different parser implementations. All previous implementations after parsing the input, stored the trees in a layout format and then proceeded with the main part of the algorithm. We do exactly the same, so that only the time of the initialization phase contributes to the total execution time of the algorithm, and not the parser.

Distance Computation. At the end of the initialization phase, the first component of T_1, denoted $\texttt{comp}(T_1)$, and the corresponding component of T_2, denoted $\texttt{comp}(T_2)$, are available. We are ready to start counting shared triplets to compute $S(T_1, T_2)$. The following steps are recursively applied:

- Starting from the root of $\texttt{comp}(T_1)$, scan the left most path of $\texttt{comp}(T_1)$ to find the splitting node u.
- Scan $\texttt{comp}(T_2)$ to compute for the binary algorithm $\sum_{v \in T_2} |s(u) \cap s(v)|$, or for the general algorithm $\sum_{(v,c') \in T_2} |s''(u,c) \cap s'(v,c')|$.
- Using the splitting node u, generate the next three components of $\texttt{comp}(T_1)$. Call the component determined by the left child of u $\texttt{comp}(T_1(u_l))$, the component determined by the right child of u $\texttt{comp}(T_1(u_r))$ and the component determined by the parent of $\texttt{comp}(T_1)$ $\texttt{comp}(T_1(u_p))$.
- Scan and contract $\texttt{comp}(T_2)$ to generate the three corresponding components, denoted $\texttt{comp}(T_2(u_l))$, $\texttt{comp}(T_2(u_r))$ and $\texttt{comp}(T_2(u_p))$, which are contracted versions of T_2 with all the necessary counters (see Section 3 and Section 4) properly maintained.
- Recurse on the three pairs of components, one from T_1 and one from T_2, and for each pair repeat the steps above.

As a last step, print $\binom{n}{3} - S(T_1, T_2)$, which is the distance $D(T_1, T_2)$ that we wanted to find.

Correctness. The correctness was extensively tested by generating hundreds of thousands of random trees of varying size and varying degree and comparing the output of our implementations against the output of the implementations of the $O(n \log^3 n)$ time algorithm in [11] and the $O(n \log n)$ time algorithm in [16].

Changing the Leaf Labels. To get the right theory bounds, changing the leaf labels of T_1 and T_2 must be done with a cache oblivious sorting routine, e.g. merge sort. In the RAM model this approach requires $O(n \log n)$ time and in the cache oblivious model $O(\frac{n}{B} \log_2 \frac{n}{M})$ I/Os. A second approach is to exploit the fact that each label is between 1 and n and use an auxiliary array that stores the new labels of the leafs in T_1, which we then use to update the leaf labels of T_2. In the RAM model the second approach requires $O(n)$ time but in the cache oblivious model $O(n)$ I/Os. In practice the problem with the first approach is that the number of instructions that it incurs eliminates any advantage that we would expect to get due to its cache related efficiency for L_1, L_2 and L_3 cache. For the input sizes tested, the array of labels easily fits into RAM, so in our implementation of both algorithms we use the second approach.

6 Experiments

In this section we present experiments illustrating the practical performance of the algorithms described in Sections 3 and 4.

6.1 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, version 16.04.2 LTS. The compiler that was used was g++ version 5.4 and cmake version 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 with n leafs by the following procedure:
  - 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 *alpha* 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 the fraction $\frac{|T(l)|}{|T(u)|}$ approaches α, more precisely $|T_l| = \max(1, \min(\lfloor a \cdot n \rfloor, n-1))$ 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', i.e make the children of u be the children of u's parent and remove u.

Note that in both models, after creating a tree, the leaf labels are shuffled by using `std::shuffle`[1] together with `std::default_random_engine`[2].

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 a natural overhead 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 [11] and [15, 4] 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 [11] 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`. The tqdist library [16], which we will refer to as `tqDist`, has an implementation of the binary $O(n \log^2 n)$ algorithm from [15] and the general $O(n \log n)$ algorithm from [4]. 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++. 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`.

[1] `http://www.cplusplus.com/reference/algorithm/shuffle/`
[2] `http://www.cplusplus.com/reference/random/default_random_engine/`
[3] `http://en.cppreference.com/w/cpp/container/unordered_map`
[4] `https://github.com/sparsehash/sparsehash`
[5] `http://icl.utk.edu/papi/`

6.2 Results

The experiments are divided into two parts. In the first part we look at how the different algorithms perform when the memory requirements do not exceed the available main memory (8G RAM). In the second part we look at how the different algorithms 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 yields the very expensive disk I/Os. All figures can be found in Appendix A.

RAM experiments. For the random model, in Figure 9 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 the new algorithms achieve the best performance. We note that for the case where $p_1 = 0.95$ and $p_2 = 0.2$, `CPDT` behaves in a different way compared to the experiments in [11]. The reason is because of the differences in the implementation of `unordered_map` that exist between the different versions of the g++ compilers. For example, if we compile with GCC 4.7, `CPDT` will produce the same performance as in [11] (see Figure 10). In Figure 11 we compare the space consumption of the algorithms. `CacheTD` is the only algorithm that uses $O(n)$ space for both binary and general trees. In theory we expect that the space consumption is better and this is also what we get in practice. In Figures 12 and 13 we can see how the contraction parameter affects the running time and the space consumption of the algorithms respectively. Finally, in Figures 14, 15 and 16 we compare the cache performance of the algorithms, i.e. how many cache misses (L1, L2 and L3 respectively) the algorithms perform for increasing input sizes and varying contraction parameters. As expected, the new algorithm achieves a significant improvement over all previous algorithms.

For the alpha model, the main interesting experimental results are illustrated in Figure 17, where 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. As mentioned in 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.

Table 1 Random model: Time performance when limiting the available RAM to 1GB. 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^{15}	0m:01s	0m:02s	0m:01s
2^{16}	0m:01s	0m:02s	0m:01s
2^{17}	0m:01s	0m:05s	0m:01s
2^{18}	0m:03s	2m:38s	0m:01s
2^{19}	0m:06s	1h:32m	0m:01s
2^{20}	0m:36s	-	0m:02s
2^{21}	19m:40s	-	0m:08s
2^{22}	>9h	-	1m:13s
2^{23}	-	-	8m:12s
2^{24}	-	-	31m:31s

n	`CPDT`	`CPDTg`	`tqDist`	`CacheTD`
2^{15}	0m:01s	0m:01s	0m:01s	0m:01s
2^{16}	0m:01s	0m:01s	0m:01s	0m:01s
2^{17}	0m:01s	0m:01s	0m:04s	0m:01s
2^{18}	0m:03s	0m:03s	0m:22s	0m:01s
2^{19}	0m:07s	0m:08s	16m:49s	0m:01s
2^{20}	29m:06s	4h:36m	>10h	0m:02s
2^{21}	>5h	-	-	1m:04s
2^{22}	-	-	-	4m:58s
2^{23}	-	-	-	27m:12s
2^{24}	-	-	-	2h:17m

Table 2 Alpha model: Time performance when limiting the available RAM to 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^{15}	0m:01s	0m:01s	0m:01s
2^{16}	0m:01s	0m:02s	0m:01s
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}	11h:20m	-	0m:03s
2^{22}	-	-	0m:35s
2^{23}	-	-	10m:09s
2^{24}	-	-	43m:52s

n	CPDT	CPDTg	tqDist	CacheTD
2^{15}	0m:01s	0m:01s	0m:01s	0m:01s
2^{16}	0m:01s	0m:01s	0m:01s	0m:01s
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

I/O experiments. In Table 1 we include the results of the I/O experiments in the random model. Each cell in the table corresponds to the execution time (including the waiting time due to disk I/Os when the trees are large and can not fit in RAM). For this experiment we used the `time` function of Ubuntu and thus also took into account the time taken to parse the input trees. Every cell contains the result of 1 run and for the input trees of size 2^{23} and 2^{24} we used the 128 bit implementation of the new algorithm in order to avoid overflows. Unlike `CacheTD`, the performance of `CPDT` and `tqDist` deteriorates significantly the moment they start performing disk I/Os.

In Table 2 we have the same experiments but now for the alpha model, where we choose $\alpha = 0.5$. Here we can clearly see how much `CPDT` and `CPDTg` are affected by the alpha parameter, e.g. for binary trees, when we go from $n = 2^{20}$ tp $n = 2^{21}$ the performance in the alpha model degrades by a factor of 1200, while in the random model, the performance degrades only by a factor of 32. We do want to emphasize that if we want to be more precise about these factors, we have to run each experiment several times and not just once. However, the overall outcome of the experiments is that the new algorithm, `CacheTD`, is clearly scaling to external memory much better than all previous algorithms.

7 Conclusion

In this paper we presented two cache oblivious algorithms for computing the triplet distance between two rooted unordered trees, one that works for binary trees and one that works for arbitrary degree trees. Both require $O(n \log n)$ time in the RAM model and $O(\frac{n}{B} \log_2 \frac{n}{M})$ I/Os in the cache oblivious model. We implemented the algorithms in C++ and showed with experiments that their performance surpasses the performance of previous implementations for this problem. In particular, our algorithms are the first to scale to external memory.

Future Work/Open Problems.

- In the cache oblivious model could we change our algorithm so that the base of the logarithm becomes M/B, i.e. achieve the sorting bound? Would that algorithm be even more efficient practice?
- Is it possible to compute the triplet distance in $O(n)$ time?

- For the quartet distance computation, could we apply similar techniques to those described in this paper in order to get an algorithm with better time bounds in the RAM model that also scales to external memory?

References

1. A. Aggarwal and J.S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
2. M.S. Bansal, J. Dong, and D. Fernández-Baca. Comparing and aggregating partially resolved trees. *Theoretical Computer Science*, 412(48):6634 – 6652, 2011.
3. V. Berry and O. Gascuel. Inferring evolutionary trees with strong combinatorial evidence. *Theoretical Computer Science*, 240(2):271–298, 2000.
4. 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.
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. William H.E. Day. Optimal algorithms for comparing trees with labeled leaves. *Journal of Classification*, 2(1):7–28, 1985.
7. A.J. Dobson. Comparing the shapes of trees. *Combinatorial Mathematics III. Lecture Notes in Mathematics*, pages 95–100, 1975.
8. 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.
9. 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.
10. 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.
11. 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.
12. J. Jansson and R. Rajaby. A More Practical Algorithm for the Rooted Triplet Distance. *Journal of Computational Biology*, 24(2):106–126, 2017.
13. D.F. Robinson and L.R. Foulds. Comparison of phylogenetic trees. *Mathematical Biosciences*, 53(1):131–147, 1981.
14. N. Saitou and M. Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4(4):406, 1987.
15. 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.
16. 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.
17. A. Sand, M.K. Holt, J. Johansen, R. Fagerberg, G.S. Brodal, C.N.S. Pedersen, and T. Mailund. Algorithms for computing the triplet and quartet distances for binary and general trees. *Biology - Special Issue on Developments in Bioinformatic Algorithms*, 2(4):1189–1209, 2013.

A Experiment Figures

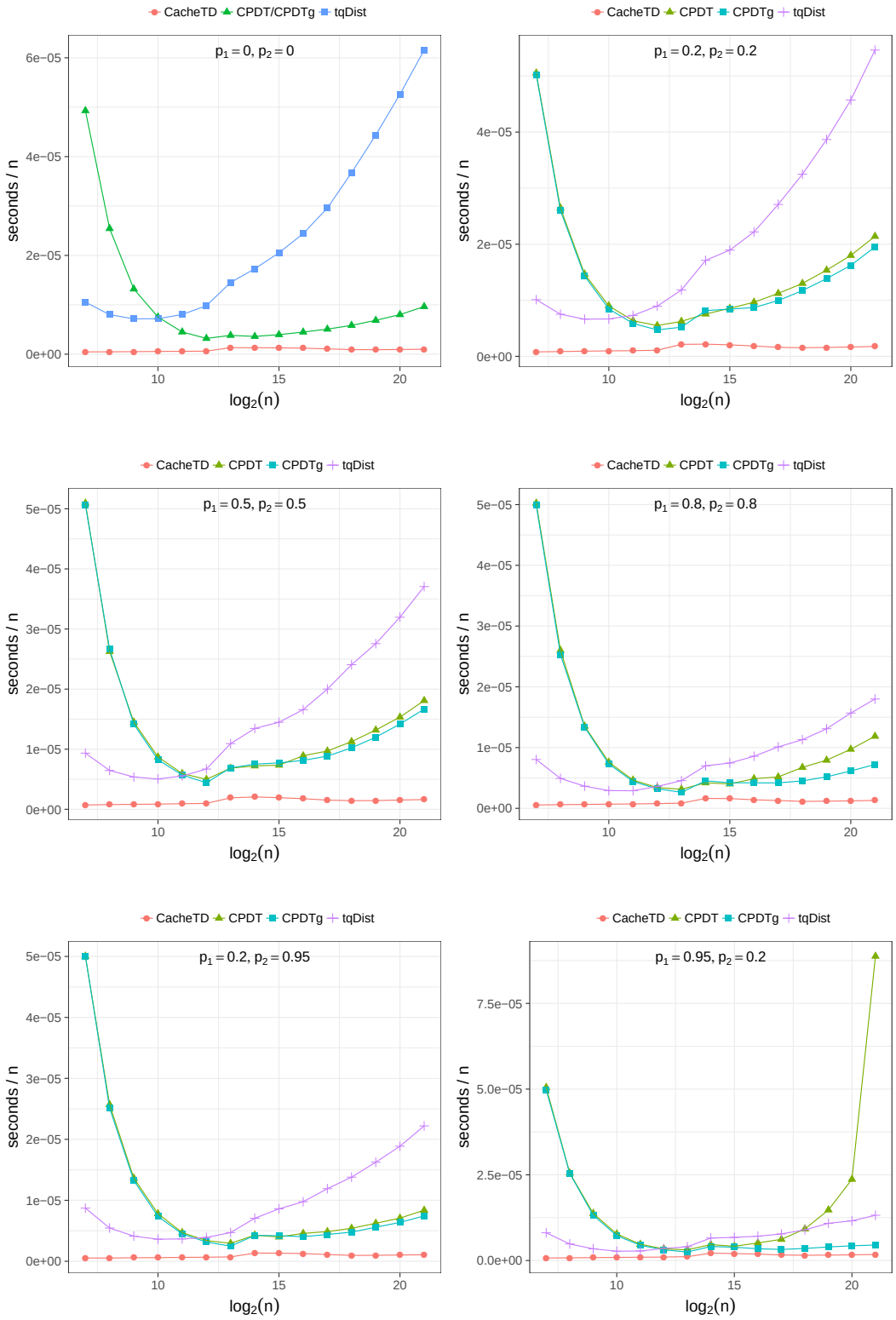


Figure 9 Time performance in the random model.

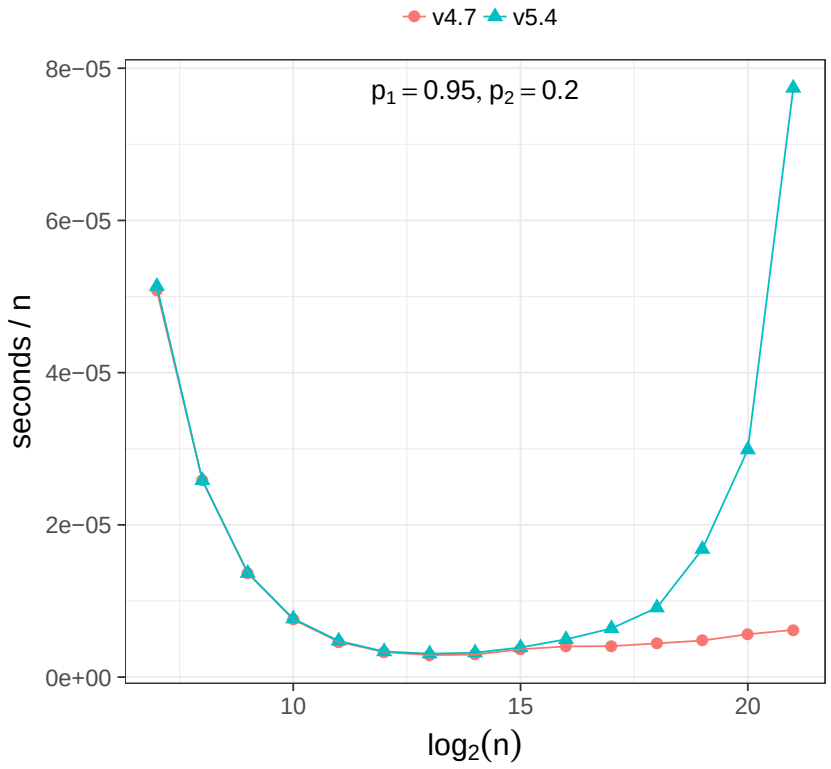


Figure 10 Comparing the time performance (random model) of `CPDT` when compiling with g++ version 4.7 and g++ version 5.4.

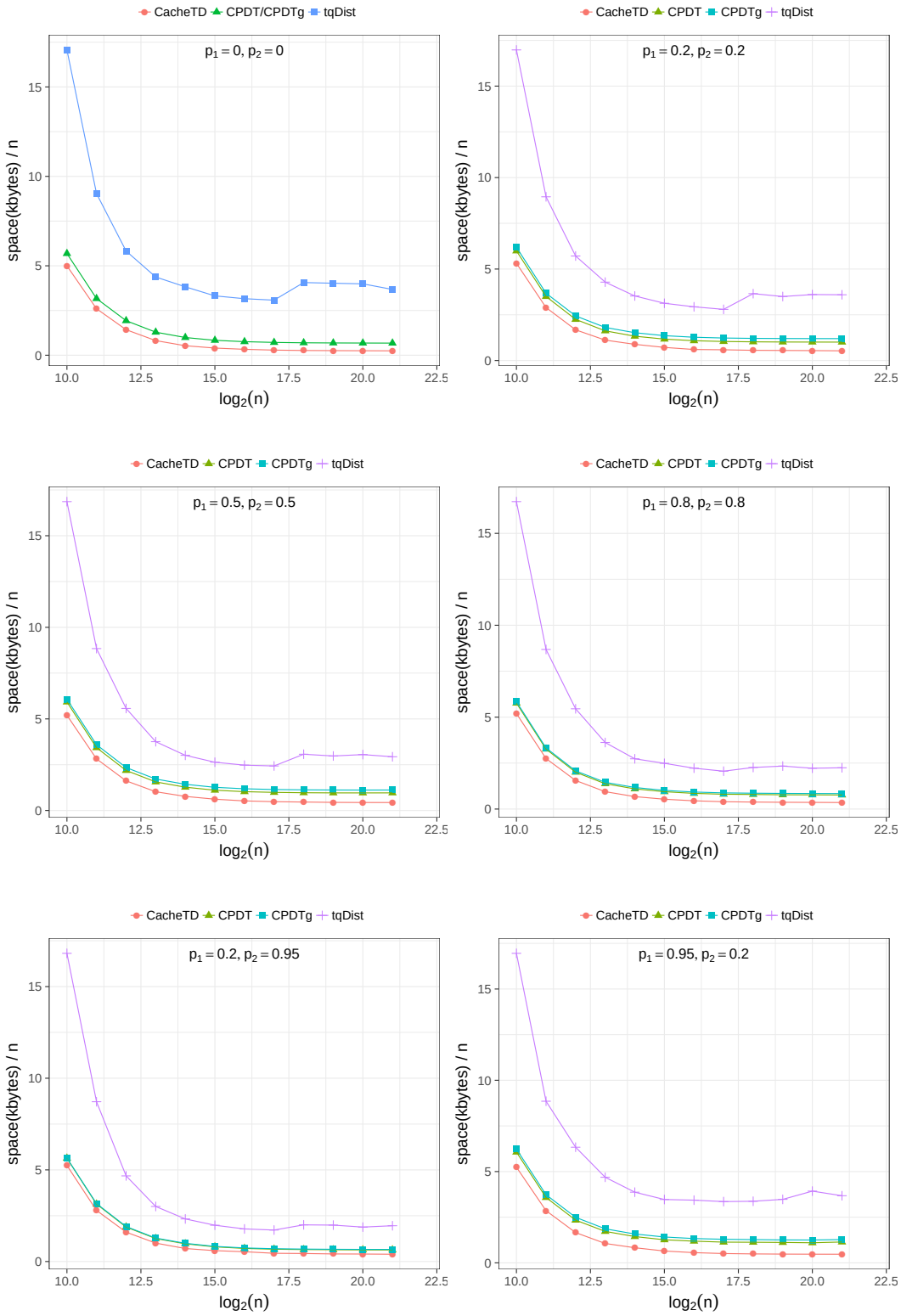


Figure 11 Space performance in the random model.

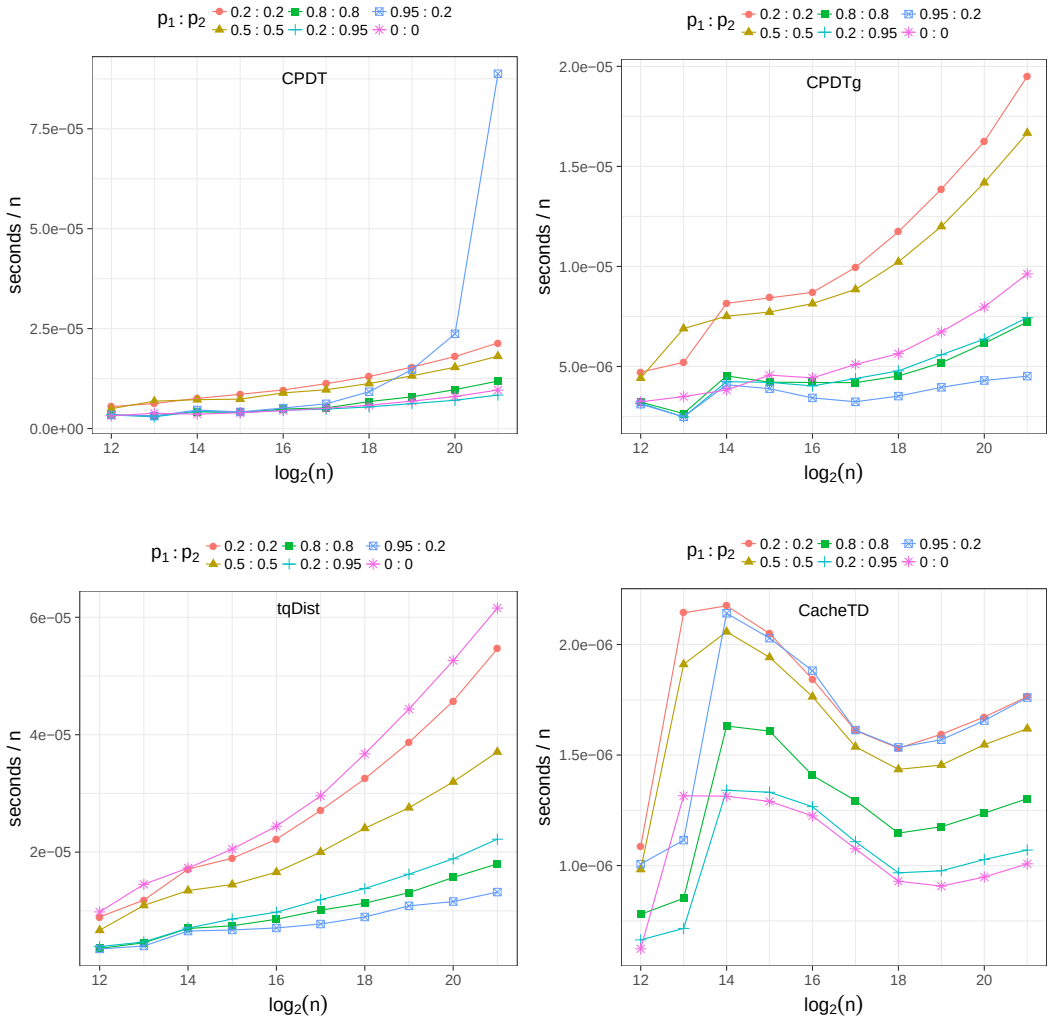


Figure 12 How the contraction parameter affects execution time (random model).

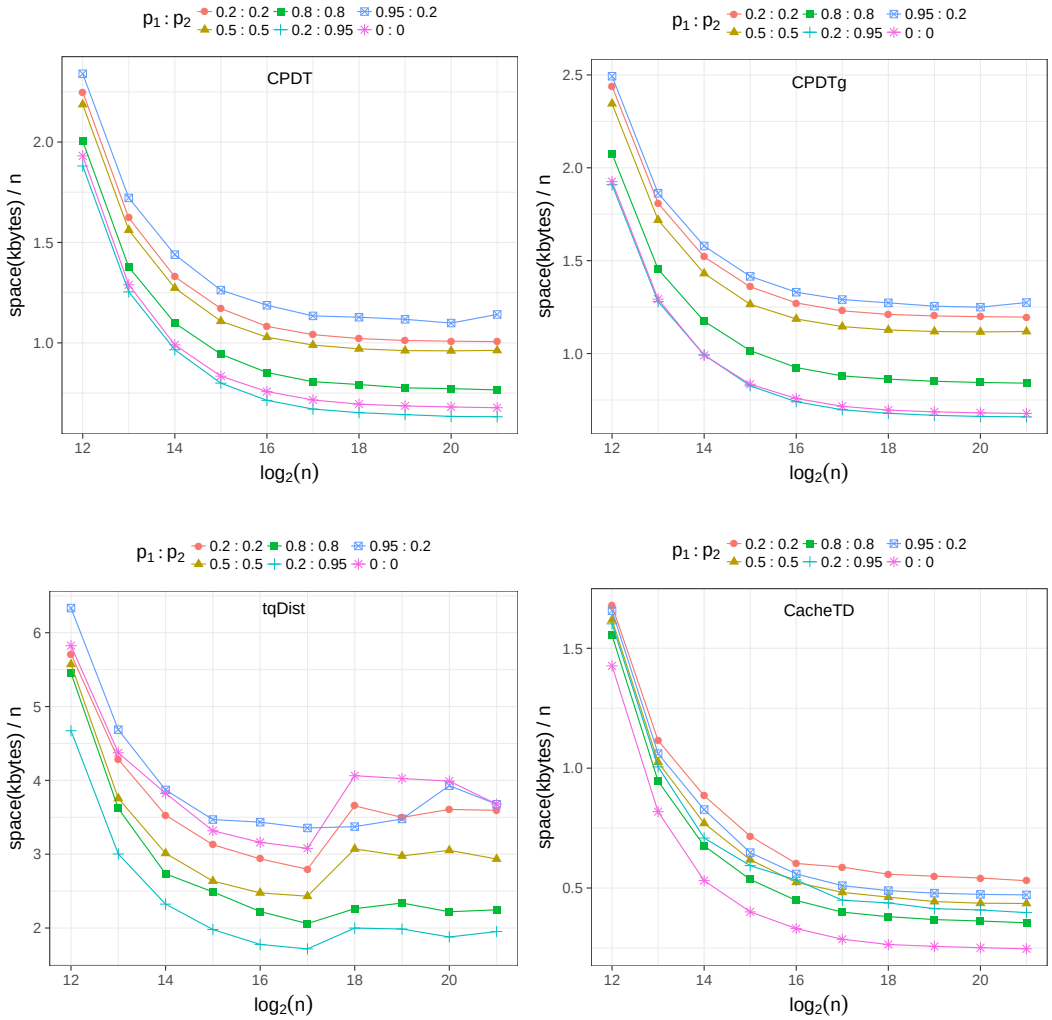


Figure 13 How the contraction parameter affects space (random model).

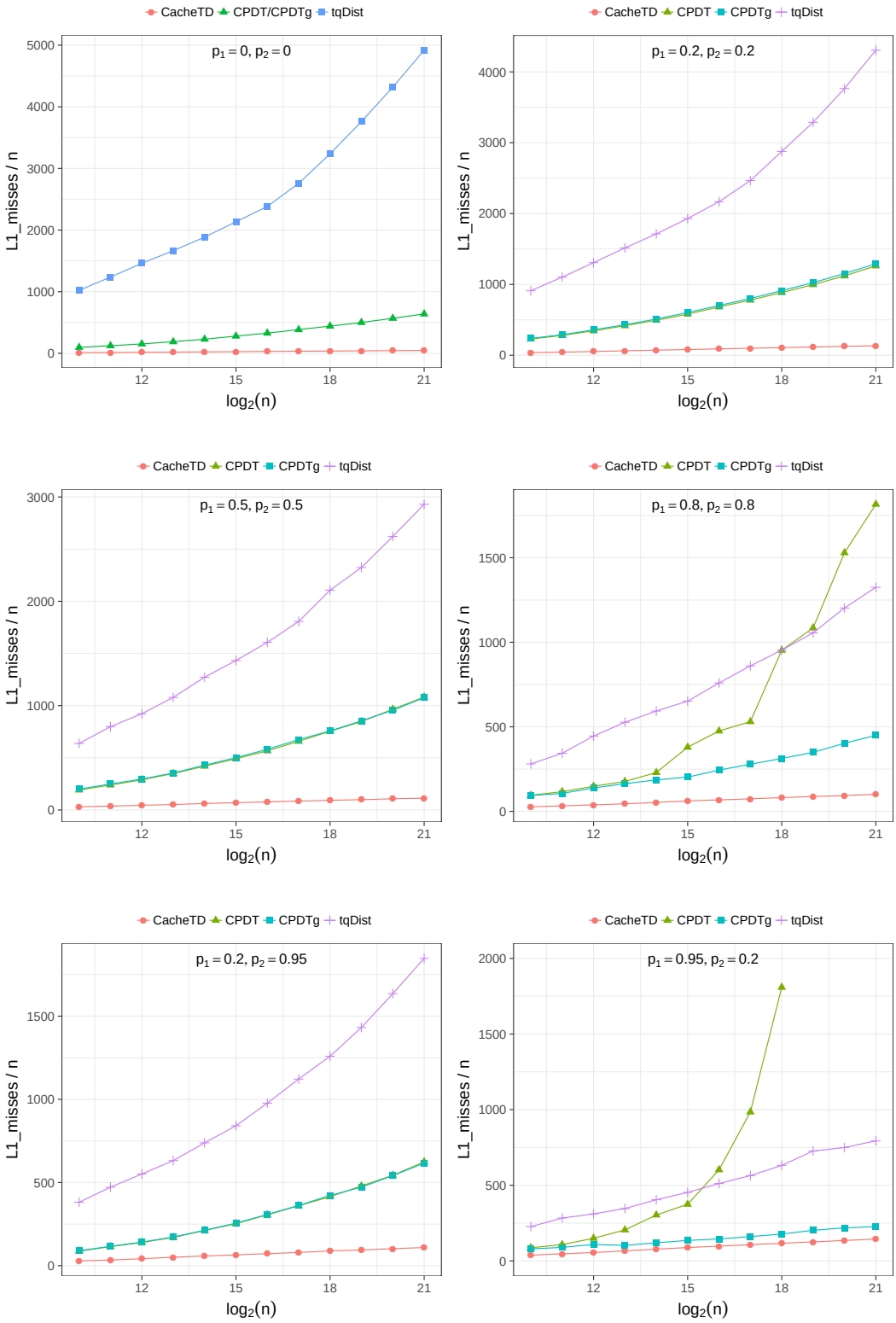


Figure 14 L1 misses in the random model.

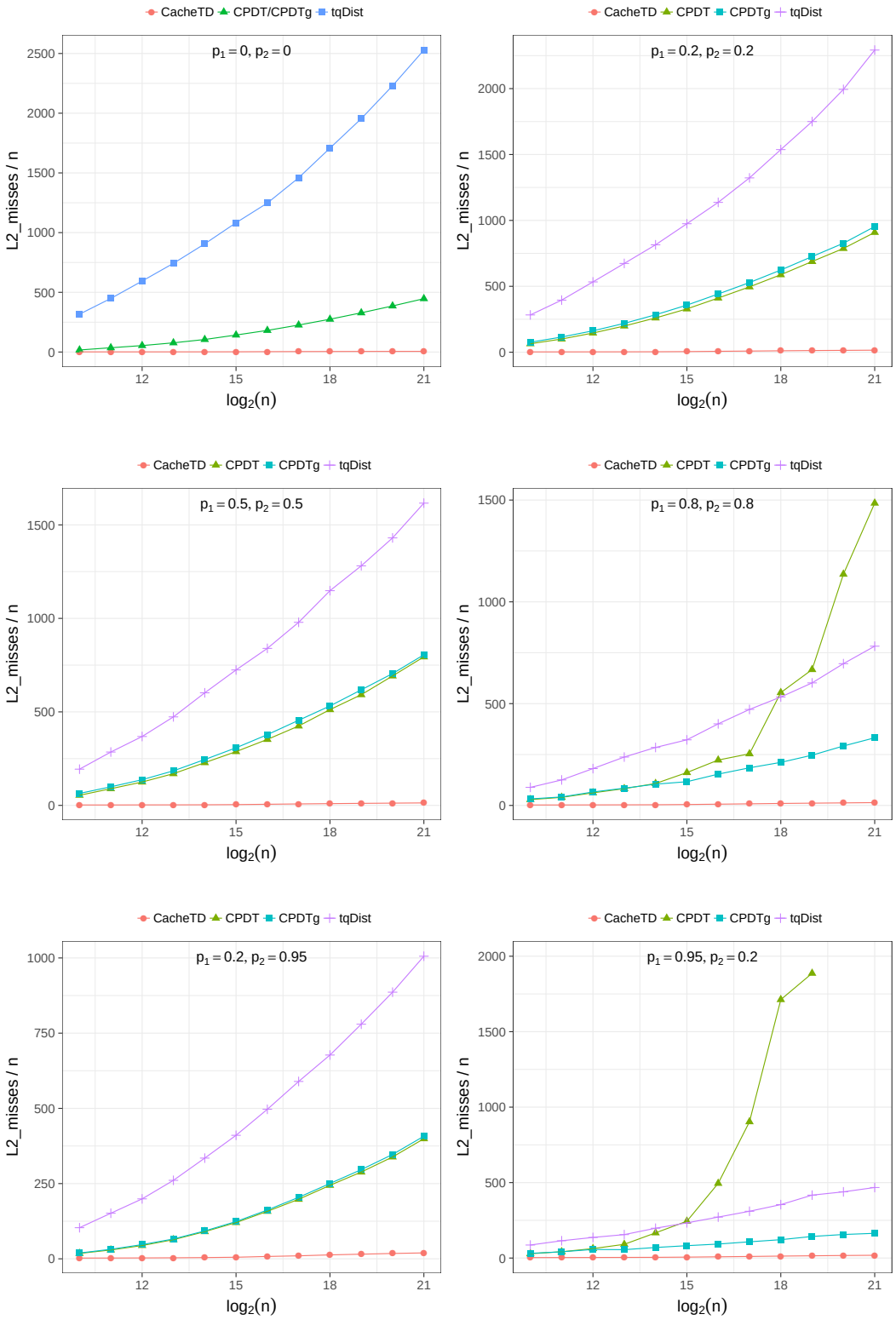


Figure 15 L2 misses in the random model.

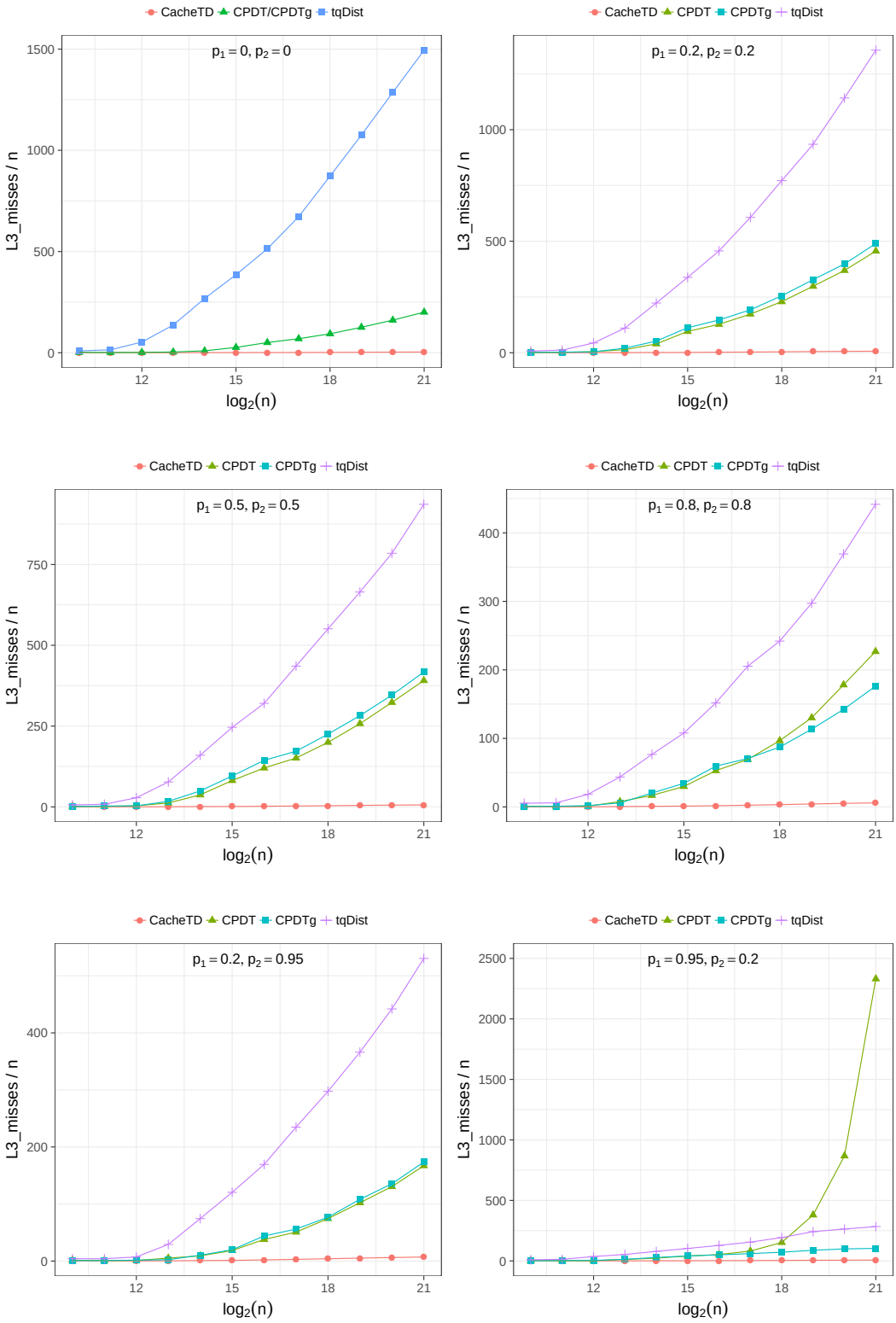


Figure 16 L3 misses in the random model.

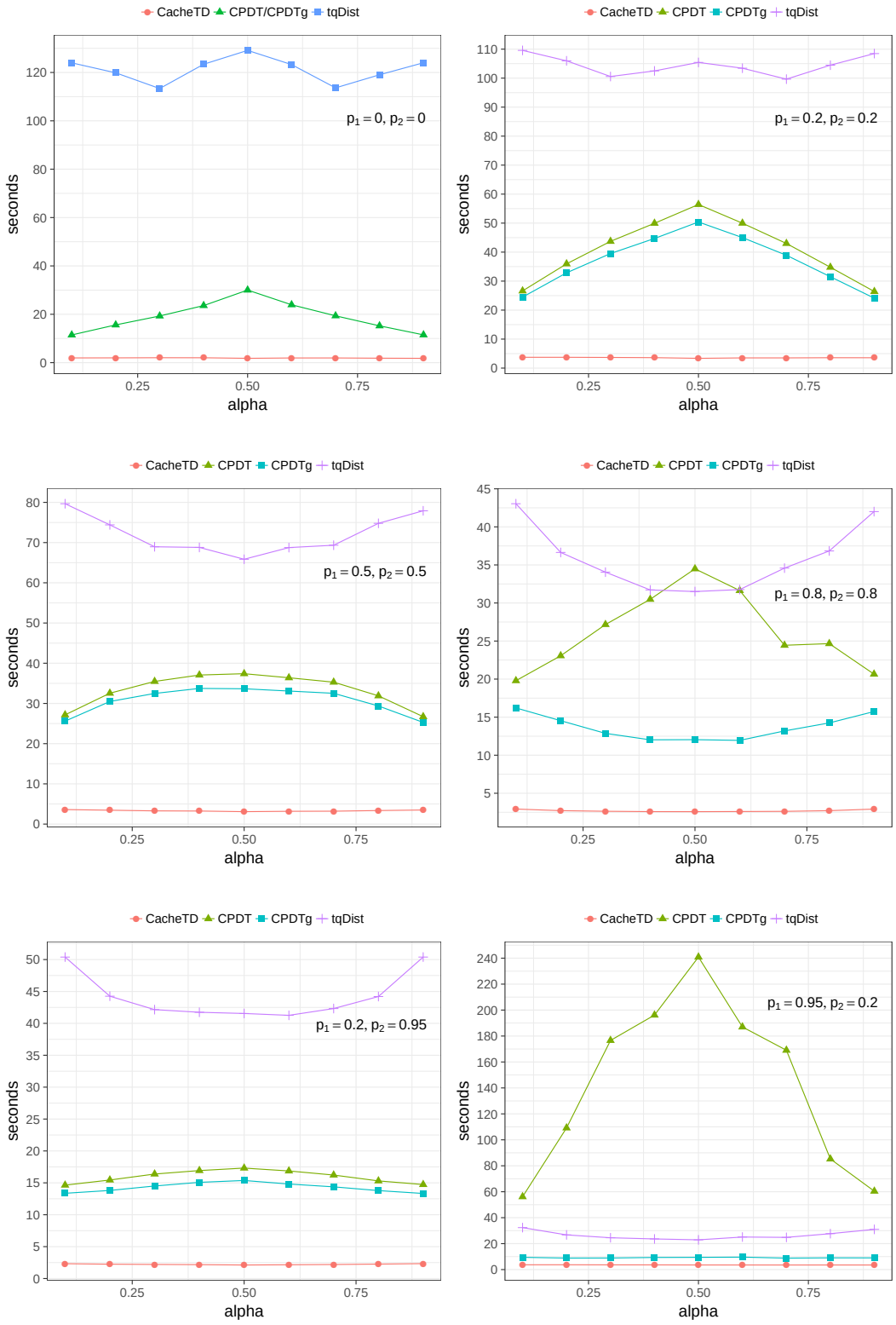


Figure 17 Alpha parameter and running time.