Cache Oblivious Algorithms for Computing the Triplet Distance Between Trees

GERTH STØLTING BRODAL, Aarhus University, Denmark
KONSTANTINOS MAMPENTZIDIS, Aarhus University, Denmark

We consider the problem of computing the triplet distance between two rooted unordered trees with n labeled leaves. Introduced by Dobson in 1975, the triplet distance is the number of leaf triples that induce different topologies in the two trees. The current theoretically fastest algorithm is an $O(n \log n)$ algorithm by Brodal *et al.* (SODA 2013). Recently Jansson and Rajaby 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 is 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 in practice, for this problem.

CCS Concepts: • **Theory of computation → Design and analysis of algorithms**.

Additional Key Words and Phrases: Phylogenetic tree, tree comparison, triplet distance, cache oblivious algorithm

ACM Reference Format:
Gerth Stølting Brodal and Konstantinos Mampentzidis. 2021. Cache Oblivious Algorithms for Computing the Triplet Distance Between Trees. *ACM J. Exp. Algor.* 26, 1, Article 1.2 (April 2021), 44 pages. https://doi.org/10.1145/3433651

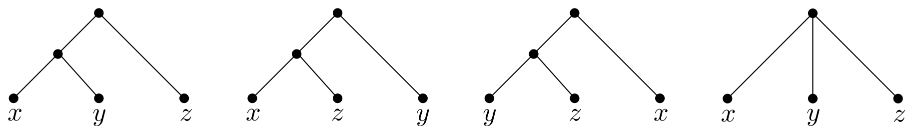
1 INTRODUCTION

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 leaves 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* [4], neighbor joining [17]) can lead to trees that are 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 that are *unordered*, i.e., trees in which the order of the siblings is not taken into account. A class of them includes distance

Authors' addresses: Gerth Stølting Brodal, Aarhus University, Department of Computer Science, Åbogade 34, 8200 Aarhus N, Denmark, gerth@cs.au.dk; Konstantinos Mampentzidis, Aarhus University, Department of Computer Science, Åbogade 34, 8200 Aarhus N, Denmark, kmampent@gmail.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2021 Association for Computing Machinery.
1084-6654/2021/4-ART1.2 $15.00
https://doi.org/10.1145/3433651



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

Fig. 1. All possible topologies of a triplet with leaves x, y, and z.

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 [16], the triplet distance [9] for rooted trees and the quartet distance [11] for unrooted trees. 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 [8]. The triplet distance can be computed in $O(n \log n)$ time [5]. The quartet distance can be computed in $O(dn \log n)$ time [5], where d is the maximal degree of any node in the two input trees, or for trees with unbounded degree in $O(n^{1.48})$ time [10].

The above bounds are in the RAM model [21]. Previous work did not consider any other models, for example external memory models like the I/O model [1] and the cache oblivious model [12]. In the external memory model one assumes that the data transfer between two levels of the memory hierarchy is the bottleneck of the computation, e.g., between disk and RAM, or between RAM and cache. External memory algorithms aim at minimizing the data transfer between these two levels. In the cache-oblivious model, algorithms try to optimize for an unknown memory hierarchy, and will therefore automatically adapt to multi-level memory hierarchies. 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 and give a significant performance improvement even for small inputs [2, 6].

The algorithm in [8] for computing the Robinson-Foulds distance can easily be adapted to external memory to achieve the sorting bound of $O(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B})$ I/Os instead of $O(n)$ I/Os for the standard implementation: The main bottleneck is the transfer of labels between the two trees, which can be done I/O efficiently using a cache oblivious sorting routine [12]. For the triplet and quartet distance measures, no such trivial modifications exist.

In this paper we focus on the triplet distance computation and present 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.

1.1 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 (leaf triple) x, y, and z and their induced topology in T (the induced topology of a set of leaves Λ in a tree T is achieved first by removing all nodes from T without any leaf from Λ in its subtree, and then by repeatedly contracting edges between nodes and their parents if the parent only has one child, see Figure 4). 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 leaves has the same lowest common ancestor. This triplet can only appear if we allow nodes with degree three or larger in T. From here on, when using the word "tree" we imply a "rooted unordered tree".

For two given 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 leaf triples that induce different topologies 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 then have the relationship $D(T_1, T_2) + S(T_1, T_2) = \binom{n}{3}$.

Previous and new results for computing the triplet distance are shown in Table 1. Note that the papers [3, 5, 7, 14, 18] do not provide an analysis of the algorithms in the cache oblivious model, so here we provide an upper bound. From here on and unless otherwise stated, any asymptotic bound refers to time.

Year	Reference	Time	I/Os	Space	Non-Binary Trees
1996	Critchlow *et al.* [7]	$O(n^2)$	$O(n^2)$	$O(n^2)$	no
2011	Bansal *et al.* [3]	$O(n^2)$	$O(n^2)$	$O(n^2)$	yes
2013	Sand *et al.* [18]	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(n)$	no
2013	Brodal *et al.* [5]	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	yes
2015	Jansson and Rajaby [14]	$O(n \log^3 n)$	$O(n \log^3 n)$	$O(n \log n)$	yes
	new	$O(n \log n)$	$O(\frac{n}{B} \log_2 \frac{n}{M})$	$O(n)$	yes

Table 1. Previous and new results for computing the triplet distance between two trees that are built on the same leaf label set of size n.

1.2 Related Work

The triplet distance was first suggested as a method of comparing the shapes of trees by Dobson in 1975 [9]. The first non-trivial algorithmic result dates back to 1996, when Critchlow *et al.* [7] proposed an $O(n^2)$ algorithm that however works only for binary trees. Bansal *et al.* [3] introduced an $O(n^2)$ algorithm that works for general (binary and non-binary) trees. Both of these algorithms use $O(n^2)$ space. Sand *et al.* [18] 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.* [5] 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 $O(n \log n)$. The space for binary trees was still $O(n)$. The algorithms from [18] and [5] were implemented and added to the library tqDist [19]. Interestingly, it was shown in [13] that for binary trees the $O(n \log^2 n)$ algorithm had a better practical performance than the $O(n \log n)$ algorithm. Jansson and Rajaby [14, 15] showed that an theoretically even slower 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 survey of previous results until 2013 can be found in [20].

1.3 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 $O(n \log n)$ algorithms for computing the triplet distance on binary and non-binary trees that use $O(n)$ space in the RAM model. Our results are the first to scale to external memory and achieve $O(n)$ space for non-binary trees. 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 performance in practice, thus essentially documenting that the theoretical results carry over to practice.

1.4 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 the algorithm to also work for general trees. In Section 5 we provide some implementation details. Section 6 contains our experimental evaluation. The Appendix contains more experimental results. Finally, in Section 7 we provide our concluding remarks.

2 PREVIOUS APPROACHES

A naive approach would enumerate over all $\binom{n}{3}$ sets of three leaf 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 algorithm does not exploit the fact that the triplets are not completely independent. For example, the triplets $xy|z$ and $yx|u$ share the leaves 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 shared triplets faster.

Critchlow *et al.* [7] exploit the depth of the leaves' ancestors to achieve the first improvement over the naive approach. Bansal *et al.* [3] exploit the shared leaves between subtrees and reduce the problem to computing the intersection size (number of shared leaves) of all pairs of subtrees, one from T_1 and one from T_2, which can be solved with dynamic programming.

2.1 The $O(n^2)$ Algorithm for Binary Trees in [18]

The algorithm for binary trees in [18] is the basis for all subsequent improvements [5, 14, 18], including ours as well, so we will describe it in more detail here. The dependency that was exploited is the same as in [3] but the procedure for counting the shared triplets is completely different.

More specifically, each triplet in any given tree T, defined by three leaf labels i, j, and k, is implicitly *anchored* in the lowest common ancestor of i, j, and k. The set of triplets that are anchored at the different nodes in T forms a partition of all $\binom{n}{3}$ triplets of T. For two nodes 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

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

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 an internal node u in T_1 and color the leaves in the left subtree of u *red*, the leaves in the right subtree of u *blue*, let every other leaf have no color and then transfer this coloring to the leaves in T_2, i.e., identically labeled leaves get the same color. For each node w in T_2 we compute $w_{\mathtt{red}}$ and $w_{\mathtt{blue}}$, which are the number of red and blue leaves in the subtree rooted at w, respectively. This can be done in a bottom-up traversal of T_2 in time $O(n)$. The triplets anchored at u are exactly the triplets $xy|z$ where x, y are blue and z is red, or x, y are red and z is blue. To compute $|s(u)\cap s(v)|$ we do as follows: let l and r be the left and right children of v. We then have

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

2.2 Subquadratic Algorithms

To reduce the time, Sand *et al.* [18] 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 $O(\log n)$ 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 of T_2 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 leaf color change in T_1 can be updated efficiently in T_2. In [18] the HDT is a binary tree of height $O(\log n)$ and every update can be done by a leaf to root path traversal in the HDT, which in total gives $O(n \log^2 n)$ time. In [5] 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 [14] as an HDT the so called *heavy-light tree decomposition* is used. Note that the only difference between all $O(n \operatorname{polylog} n)$ results that are available right 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

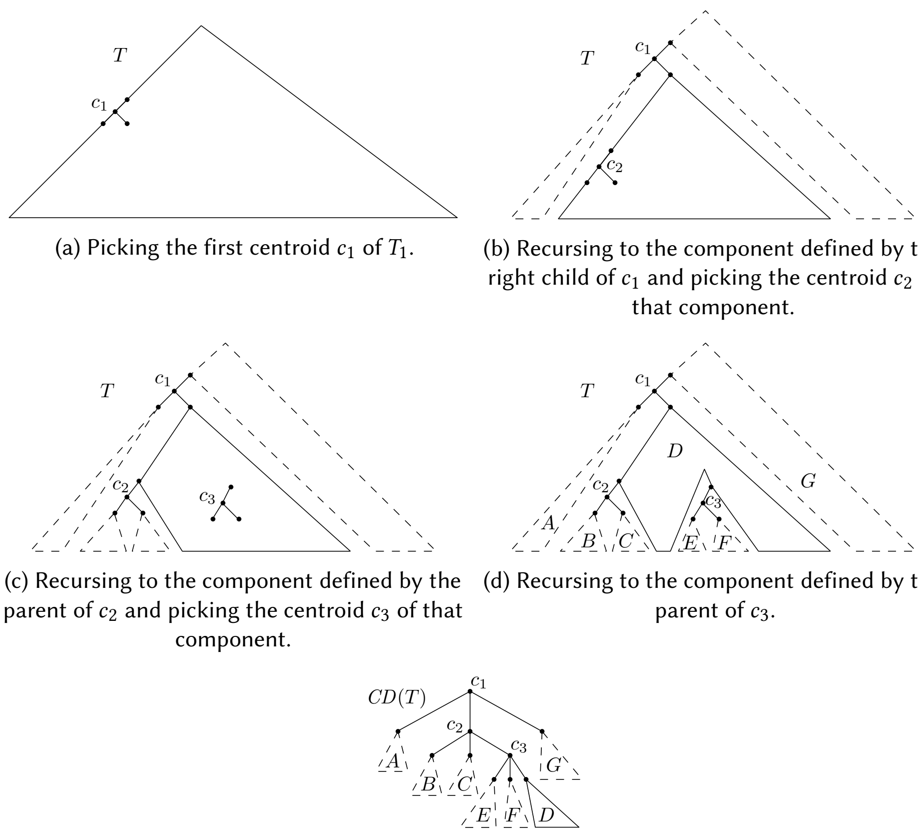
In this section, we provide a cache oblivious algorithm that for two binary trees T_1 and T_2, built on the same leaf label set of size n, computes $D(T_1, T_2)$ using $O(n \log n)$ time and $O(n)$ space in the RAM model, and $O(\frac{n}{B} \log_2 \frac{n}{M})$ I/Os in the cache oblivious model.

3.1 Overview

We use the $O(n^2)$ algorithm from Section 2.1 as a basis. The main difference between this algorithm and our new algorithm is 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 is found 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 leaves 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. Hence, we completely remove the need for 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 contracted trees in memory, so that every time we scan a tree of size s we spend $O(s/B)$ I/Os.

3.2 Modified Centroid Decomposition

For a given binary tree T let $|T|$ denote the number of nodes in T (internal nodes and leaves). For a node u in T let l and r be the left and right children of u, and p the parent of u. Removing u from T partitions T into three (possibly empty) *connected components* T_l, T_r, and T_p containing l, r,



(a) Picking the first centroid c_1 of T_1.

(b) Recursing to the component defined by the right child of c_1 and picking the centroid c_2 of that component.

(c) Recursing to the component defined by the parent of c_2 and picking the centroid c_3 of that component.

(d) Recursing to the component defined by the parent of c_3.

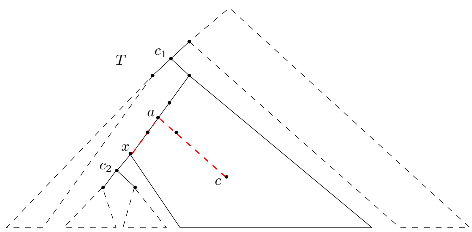
(e) The centroid decomposition $CD(T)$.

Fig. 2. (a) - (d) Generating a component D, outlined by a solid line in (d), that has two edges crossing its boundary from below. (e) The corresponding centroid decomposition $CD(T)$.

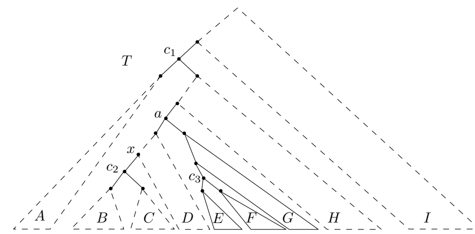
and p, respectively. A *centroid* is a node u in T such that $\max\{|T_l|, |T_r|, |T_p|\} \leq |T|/2$. A centroid always exists and can be found by starting from the root of T and iteratively visiting the child with a largest subtree, eventually we will reach a centroid. Finding the size of every subtree and identifying u takes $O(|T|)$ time in the RAM model. By recursively finding centroids in each of the three components, we in the end get a ternary tree of centroids, which is called the *centroid decomposition* of T, denoted $CD(T)$. The internal nodes of $CD(T)$ are internal nodes of T (centroids), and the leaves of $CD(T)$ are components of size one in T, which can be either leaves or internal nodes of T. We order the children of an internal node u of $CD(T)$, such that the children from left to right are the components containing the left child, right child, and parent of u in T. We can construct a level of $CD(T)$ in $O(|T|)$ time, given the decomposition of T into components by the previous level. Since $CD(T)$ has at most $1 + \log_2(|T|)$ levels, the total time required to construct $CD(T)$ is $O(|T| \log |T|)$, thus we get Lemma 3.1.

Lemma 3.1. *For any given binary tree T with n leaves, there exists an algorithm that constructs $CD(T)$ using $O(n \log n)$ time and $O(n)$ space in the RAM model.*

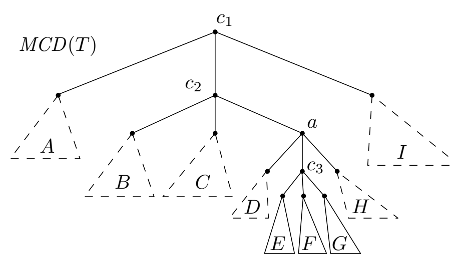
A component in a centroid decomposition $CD(T)$, might have several edges crossing its boundaries (connecting nodes inside and outside the component). An example of creating a component



(a) The solid line shows a component with an edge from below to x and centroid c. The component is split at the common ancestor a of x and c.



(b) Recursing to the component defined by the right child of a in (a). Notice that c_3 might be different from c in (a).



(c) The modified centroid decomposition $MCD(T)$ corresponding to (b).

Fig. 3. Modified centroid decomposition. Generating a component D in $MCD(T)$, outlined by a solid line in (d), that has two edges crossing its boundary from below. (e) The corresponding centroid decomposition $MCD(T)$.

that has two edges crossing its boundary from below can be found in Figure 2 (an edge *from below* is an edge between a node u inside the component and a node v outside the component where v is a child of u). By following the same pattern of generating components as depicted in Figure 2d, $CD(T)$ 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.

An $MCD(T)$ is built as follows: The first component is defined by T, just like in $CD(T)$. To find recursively the rest of the components, if a component C has no edge from below, we select the centroid c of C as a splitting node, just like when constructing $CD(T)$. Otherwise, let (x, y) be the edge that crosses the boundary from below, where x is in C and y the child of x not in C. Let c be the centroid of C (possibly $x = c$). As a splitting node choose the lowest common ancestor a of x and c, possibly x or c (see Figure 3). 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 3.2. *For a binary tree T, the height of $MCD(T)$ is at most* $2 + 2\log_2 |T|$.

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 a 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, where C_l contains x and C_r contains c, i.e., we have

$|C_l| + |C_p| \leq \frac{|C|}{2}$ 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 that C_r is going to be a component corresponding to a full 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, thus in the following recursion level the three components produced from C_r will be such that their sizes are at most half the size of C. It follows that for a component of size $|C|$ with one edge from below, we will need at most two levels in $MCD(T)$ before having components with size at most $\frac{|C|}{2}$. □

Similarly to the construction of $CD(T)$, we can construct in $O(|T|)$ time a level of $MCD(T)$ given the decomposition of T into components by the previous level of $MCD(T)$. Hence, every level of $MCD(T)$ can be constructed in $O(|T|)$ time. Since we have $|T| = 2n - 1$, we then obtain the following:

Theorem 3.3. *For any given binary tree T with n leaves, there exists an algorithm that constructs $MCD(T)$ using $O(n \log n)$ time and $O(n)$ space in the RAM model.*

3.3 The Main Algorithm

There is a preprocessing step and a counting (of shared triplets between T_1 and T_2) step.

In the preprocessing step, first we apply a depth-first traversal on T_1 to 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. This ensures that the additional centroids required by the *MCD* are on leftmost paths, and allows for an I/O efficient memory layout of the tree. 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 leaves are numbered 1 to n from left to right. Both steps take $O(n)$ time in the RAM model. The new labels are then transferred to T_2 using either hashing or sorting in expected $O(n)$ time or $O(n \log n)$ time in the RAM model, respectively. The relabelling is performed 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. Finally, we construct $MCD(T_1)$ as described in Section 3.2.

In the counting step, 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 compute $\sum_{v \in T_2} |s(u) \cap s(v)|$. We achieve this by processing T_2 in two phases, the *contraction* phase and the *counting* phase.

3.3.1 Contraction Phase of T_2. Let $L(T_2)$ denote the set of leaves in T_2 and $\Lambda \subseteq L(T_2)$ be a subset of the leaves of T_2. In the contraction phase, T_2 is compressed into a binary tree of size $O(|\Lambda|)$ whose leaf set is Λ and all internal nodes have two children. The contraction is done such that all the topologies induced by Λ in T_2 are preserved in the compressed binary tree (see Figure 4). This is achieved by the following three steps, which can be done in a single depth-first traversal of T_2 in time $O(|T_2|)$:

- Prune all leaves 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 traversing $T_2(u)$ instead of T_2. One can imagine $MCD(T_1)$ as being a tree where each node u is augmented with $T_2(u)$.

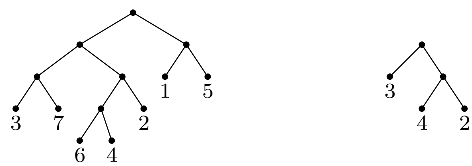


Fig. 4. Contraction of a tree for the leaf set $\Lambda = \{2, 3, 4\}$.

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 $MCD(T_1)$, with total size $O(|T_1|)$. For each component C_u at level $i-1$ we contract $T_2(u)$ up to three times, once for each child u' of u in $MCD(T_1)$, where we apply the above contraction to $T_2(u)$ with $\Lambda = L(C_{u'})$. This means that we can generate the contractions of T_2 for level i in $O(n)$ time, thus we can generate all contractions of T_2 in $O(n \log n)$ time. Note that by explicitly storing all contractions, we would also need to use $O(n \log n)$ space. For our problem, because we traverse $MCD(T_1)$ in a depth-first manner, we only need to store the contractions corresponding to the stack of nodes of $MCD(T_1)$ that we have to remember during the traversal of $MCD(T_1)$. Since the components at every second level of $MCD(T_1)$ have at most half the size of the components two levels above, Lemma 3.4 states that the size of this stack is always $O(n)$.

Lemma 3.4. *Let T_1 and T_2 be two binary trees with n leaves and $u_1, u_2, \ldots, u_k$ a root to leaf path of $MCD(T_1)$. For the sizes of 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 $T_2(u_1) = T_2$, thus $|T_2(u_1)| \le 2n$. From the proof of Lemma 3.2 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 have a size of at most $\frac{x}{2}$. This means that $\sum_{i=1}^{k} |T_2(u_i)| \le 2n + 2n + \frac{2n}{2} + \frac{2n}{2} + \frac{2n}{4} + \frac{2n}{4} + \cdots + \frac{2n}{2^i} + \frac{2n}{2^i} + \cdots = 2\sum_{j=0}^{\infty} \frac{2n}{2^j} \le 8n = O(n)$. □

3.3.2 Counting Phase of T_2. In the counting phase, we find the value of $\sum_{v \in T_2} |s(u) \cap s(v)|$ by traversing $T_2(u)$ instead of T_2. This makes the total time of the algorithm in the RAM model $O(n \log n)$, with the space being $O(n)$ because of Lemma 3.4. We consider the following two cases:

- *C_u has no edges from below.*

  In this case C_u corresponds to a full subtree of T_1. We act exactly like in the $O(n^2)$ algorithm (Section 2) but now instead of traversing T_2 we do a bottom up traversal of $T_2(u)$ and compute for each node v in $T_2(u)$ the values $v_{\mathtt{blue}}$ and $v_{\mathtt{red}}$, and the number of shared anchored triplets (1) rooted at u and v.

  Note that to find shared triplets between T_1 and T_2 anchored at u in T_1 and v in T_2, it is sufficient to consider triplets anchored at a node v in $T_2(u)$, since a node v removed from T_2 by the contraction has at most one child containing leaves from C_u.
- *C_u has one edge from a subtree X_u from below.*

  In this case C_u does not correspond to a full subtree of T_1, since X_u is outside of C_u (see Figure 5). Note that because in the preprocessing step T_1 was made left-heavy, it follows by induction on the MCD construction steps that X_u is always rooted at a node on the leftmost path from u, i.e., all leaves in X_u are red and can be part of triplets that are anchored in u. 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 counters v_{ts} and v_{ps} about the leaves from X_u that were contracted away

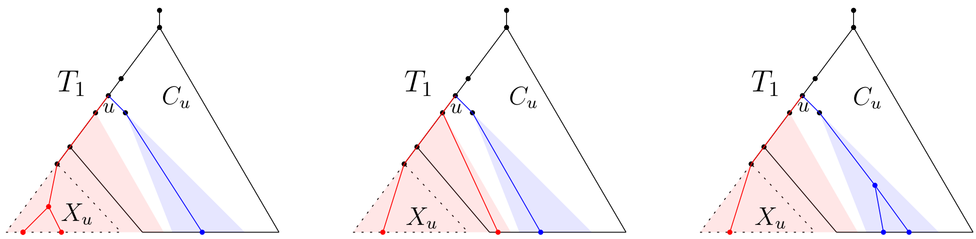


Fig. 5. $MCD(T_1)$: Triplets (red and blue) that can be anchored in u with the leaves not being in the component C_u.

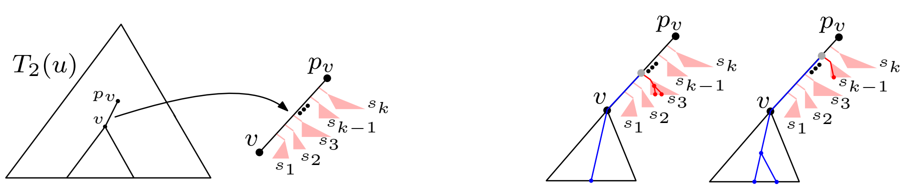


Fig. 6. Contracted subtrees on edges in $T_2(u)$ and shared triplets rooted at contracted nodes.

in T_2. If v is the root of $T_2(u)$, we add an extra edge to store this information. For every such edge (p_v, v), let $s_1, s_2, \ldots, s_k$ be the contracted subtrees rooted at the edge (see Figure 6). Every such subtree contains either leaves with no color (leaves outside the subtree rooted at u in T_1) or red leaves from X_u. For every node v in $T_2(u)$ we compute the following counters while contracting $T_2(u')$ to $T_2(u)$, where u' is the parent of u in $MCD(T_1)$:

– $v_{\texttt{red}}$: total number of red leaves in the subtree of v (including those coming from X_u).
– $v_{\texttt{blue}}$: total number of blue leaves in the subtree of v.
– v_{ts}: total number of red leaves in $s_1, s_2, \ldots, s_k$.
– v_{ps}: total number of pairs of red leaves in $s_1, s_2, \ldots, s_k$ 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 leaves in s_i.

During the traversal of $T_2(u')$ we compute for each node v the number of leaves x_v in the subtree that are in X_u, including adding the w_{ts} counters for all w in the subtree below v. If an internal node v is pruned, we add x_v to p_{ts} and add $\binom{x_v}{2}$ to p_{ps}, where p is the parent of v. Whenever a unary node p is contracted with its child c, we set $c_{ts} = c_{ts} + p_{ts}$ and $c_{ps} = c_{ps} + p_{ps}$. For the initial tree, i.e., $T_2(u') = T_2$, we have all v_{ts} and v_{ps} counters equal to zero.

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 $\binom{v_{\texttt{blue}}}{2} \cdot v_{ts} + v_{\texttt{blue}} \cdot v_{ps}$.

Note that the first case can be treated as a special case of the second case, where $X_u = \emptyset$ and all v_{ts} and v_{ps} counters are zero.

3.4 Scaling to External Memory

We now describe how to make the algorithm scale to external memory. The tree T_1 is stored in an array of size $2n - 1$ in a preorder layout, i.e., if a node w of T_1 is stored in position p, the left child of w is stored in position $p + 1$ and if x is the size of the left subtree of w, the right child of w is stored in position $p + x + 1$. In general a component C_u in T_1 with missing subtree X_u on the leftmost path, in this layout will consist of the nodes on the leftmost path to X_u, followed by the recursive layout of X_u, and then the remaining subtrees of C_u left to right, i.e., the layout of C_u consists of two consecutive pieces of the layout. For T_2 and its contractions, we use the proof of Lemma 3.4 to initialize a large enough array that can fit T_2 and every contraction of T_2 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. To maintain a consecutive layout of T_2 during the contraction phase, the tree T_2 and its contractions are stored in memory following a postorder layout, i.e., if a node w is stored in position p and y is the size of the right subtree of w, the left child of w is stored in position $p - y - 1$ and the right child of w is stored in position $p - 1$.

In the preprocessing step, T_1 can be made left-heavy with two depth-first traversals. The first traversal computes for every node u in T_1 the size of the subtree rooted at u. The second traversal starts from the root of T_1, recursively visits the children by first visiting a largest child, and prints all nodes visited along the way to an output array. This output array will at the end of the traversal contain the left-heavy version of T_1 in a preorder layout. From the following Lemma 3.5 we have that both the first and second depth-first traversal of T_1 require $O(n/B)$ I/Os in the cache oblivious model, i.e., making T_1 left-heavy requires $O(n/B)$ I/Os in the cache oblivious model.

In Lemma 3.5 we consider the I/Os required to apply a depth-first traversal on a binary tree T that is stored in memory following a local layout, i.e., the nodes of every subtree of T are stored consecutively in memory and every node has at most three occurrences in memory: possibly before, after, and/or between the layout of the children (see Figure 7). From here on, when we refer to an edge (u, v), we imply that u is the parent of v in T. During a depth-first traversal of T, an edge (u, v) is *processed* to either visit v from u or to backtrack from v to u. W.l.o.g. we assume that when an edge (u, v) is processed, both u and v are visited, i.e., all blocks of memory containing copies of u and v must be in cache.

Lemma 3.5. *Let T be a binary tree with n leaves that is stored in an array following a local layout, i.e., the nodes of every subtree of T are stored consecutively in memory and every node has at most three occurrences in memory. Any depth-first traversal that starts from the root of T, and in which for every internal node u in T the children of u are traversed in any order, requires* $O(n/B)$ *I/Os in the cache oblivious model.*

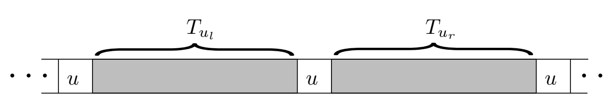


Fig. 7. Positions of the occurrences of a node u in memory with respect to the two children subtrees of u.

Proof. For a node u in T, let T_u denote the set of nodes in the subtree of T rooted at u. Let u_l and u_r be the two children of u. We assume that u is stored at all the three possible occurrences in memory with respect to the layout of T_{u_l} and T_{u_r}, as illustrated in Figure 7. This assumption is w.l.o.g. because in any local layout one or more of these positions is used, thus the number of I/Os is upper bounded by the number of I/Os incurred. This placement of u in memory implies that when u is visited in a depth-first traversal of T, all the three copies of u are accessed in memory. Note that according to the definition of a local layout, T_{u_l} and T_{u_r} can be interchanged in Figure 7. In the following, the aim is to bound the number of I/Os implied.

Define a node u in T to be *B-light* if $3|T_u| \leq B - 2$, otherwise the node is said to be *B-heavy*. Observe that the children of a B-light node are all B-light. We consider the following disjoint sets of nodes from T:

S_1: B-light nodes,

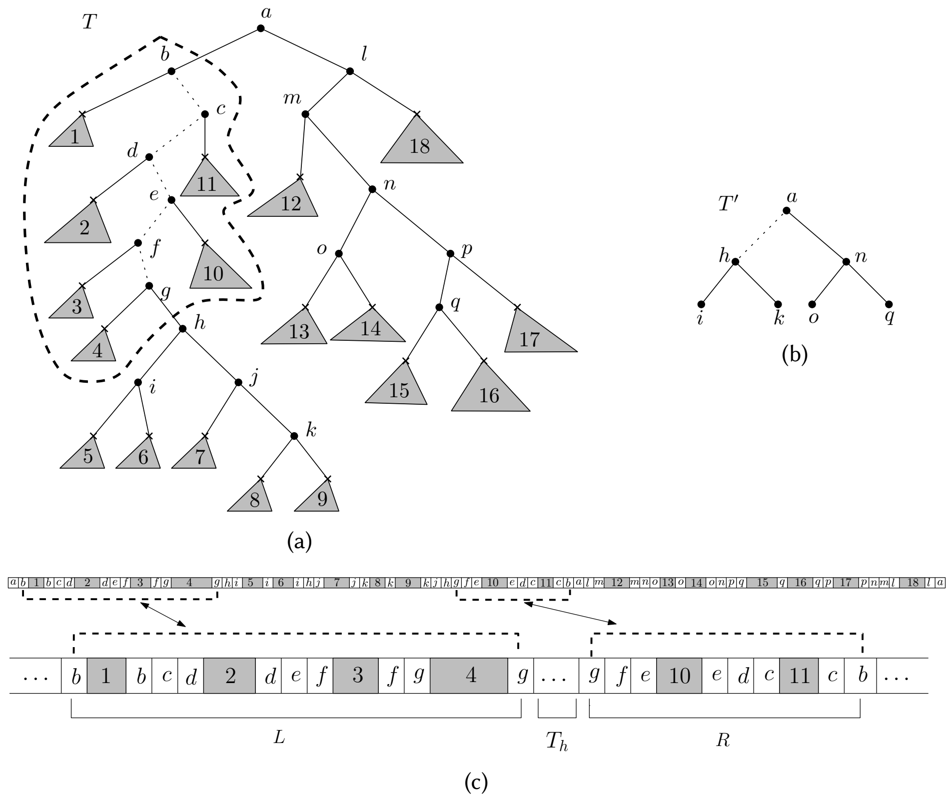


Fig. 8. (a) A tree T. The gray subtrees are B-light subtrees and every node not in a B-light subtree is a B-heavy node. (b) The corresponding tree T' according to the proof of Lemma 3.5. (c) How T is stored in memory, the two segments of memory (in dashed lines) that correspond to the edge (a, h) in T' and how the nodes in $P_{(a,h)}$ are visited (defined by the one directional lines) during a depth-first traversal of T.

S_2: B-heavy nodes with only B-light children,
S_3: B-heavy nodes with two B-heavy children, and
S_4: B-heavy nodes with one B-heavy child and one B-light child.

For a B-light node u in S_1, let w be the first B-heavy node we reach in the path from u to the root of T. Any I/O incurred by visiting the node u in T is charged to w. This node w can be either in S_2 or S_4. Let w' be the B-light child of w such that $T_{w'}$ contains u. Since a subtree is stored in a contiguous piece of memory and each node has at most three occurrences, then $3|T_{w'}| \leq B - 2$ implies that at most $O(1)$ I/Os are sufficient to visit all nodes in $T_{w'}$. We say that $T_{w'}$ is a subtree that is B-light (see Figure 8a).

We now argue that S_2 and S_3 have size $O(n/B)$. Let T' be the binary tree created by pruning every B-light node and their incident edges from T, and subsequently contracting unary nodes. Observe that S_2 are the leaves of T', S_3 the internal nodes of T', and $S_1 \cup S_4$ are the nodes pruned from T to achieve T'. An example for T and the corresponding tree T' can be found in Figures 8a and 8b. Since the leaves of T' correspond to disjoint subtrees of T of size larger than $\frac{B-2}{3}$, we have $|S_2| < 3|T|/(B-2) = O(n/B)$. Since T' is a binary tree, the number of internal nodes equals the number of leaves minus one, and we have $|S_3| = |S_2| - 1 = O(n/B)$.

We now argue that the total number of I/Os incurred by the nodes in S_1 and S_4 is $O(n/B)$, thus proving the statement. Let v be a node in T' and u the parent of v. The edge (u, v) corresponds to the path from u to v in T except u and v, denoted $P_{(u,v)}$, containing B-heavy nodes from S_4. For example the edge (a, h) in Figure 8b corresponds to $P_{(a,h)} = (b, c, d, e, f, g)$. Let $C_{(u,v)}$ be $P_{(u,v)}$ together with all B-light subtrees rooted at a child of a node in $P_{(u,v)}$. By the local layout of T, the nodes in $C_{(u,v)}$ are stored in two segments of memory L and R on the left and right side of the layout of T_v, respectively (see Figure 8c). The layout of T_u can be obtained by starting with the layout of T_v, and then considering the B-light subtrees hanging of from the path from v to u bottom-up, and then incrementally adding the layout of these B-light subtrees either to the left or the right of the current layout.

A general depth-first traversal of T will visit the nodes on the path from u to v, first top-down and then bottom-up. The B-light subtrees hanging of from a node on the path will then be recursively visited either on the way down or on the way up, i.e., a subset of the trees will be visited on the way down, and the remaining subtrees on the way up. On the way down the subtrees will be considered left to right in L and right to left in R, alternating between the two sides depending on the layout. Similarly, on the way up we will alternate to consider subtrees in L right to left and R left to right. Since each of the B-light subtrees in L and R uses at most $B - 2$ positions in memory, by accessing all three copies of a node w in $P_{(u,v)}$ every time w is visited in a depth-first traversal of T, we guarantee that the corresponding B-light subtree rooted at w is in cache, i.e., it can be accessed in memory for free. We bound the I/O cost for accessing $C_{(u,v)}$ by the cost of scanning L and R on the way down and up, i.e., the cost of scanning L and R once in both directions. Hence, the total number of I/Os that are sufficient to pay for traversing all nodes in $C_{(u,v)}$ is $4 + 2 \cdot 3|C_{(u,v)}|/B$, where the +4 comes from the 4 I/Os we need to pay (in the worst case) to visit the first and last block of L and R. The total number of I/Os we need to spend for all $O(n/B)$ paths of T that correspond to edges of T' is $\sum_{(u,v)\in T'}(4 + 6|C_{(u,v)}|/B) = O(n/B)$. Together with the fact that for each of the $O(n/B)$ nodes in S_2 and S_3 we only spend $O(1)$ I/Os, the statement follows. □

Changing the labels of T_1 can be done in $O(\frac{n}{B} \log_2 \frac{n}{M})$ I/Os with a cache oblivious sorting routine, e.g., with binary merge sort. Making T_1 left-heavy can be done by two depth-first traversals of T_1: In the first traversal we for each node compute the size of the subtree, and in the second traversal we traverse the heaviest subtrees first, and output the new left-heavy layout of T_1. By Lemma 3.5 each traversal requires $O(n/B)$ I/Os – assuming the initial layout of T_1 is a local layout. Overall, the preprocessing step requires $O(\frac{n}{B} \log_2 \frac{n}{M})$ I/Os.

When constructing $MCD(T_1)$, we can find the splitting node of a component C_u by a top-down traversal from the root of C_u in T_1 in $O(1 + |C_u|/B)$ I/Os (by Lemma 3.5). In $T_2(u)$ we spend $\Theta(1+|T_2(u)|/B)$ I/Os for the contraction and counting phase (by Lemma 3.5). Since $|T_2(u)| = \Theta(|C_u|)$, the overall cost to construct a $(C_u, T_2(u))$ pair and to count the shared triplets anchored in u is $\Theta(1 + |C_u|/B)$ I/Os. To account for the total I/O cost for constructing all pairs, we need a refined analysis. Assign to each node u of $MCD(T_1)$ the *rank* $\lfloor \log_2 |C_u| \rfloor$. The ranks of the nodes are non-increasing on a root to leaf path in $MCD(T_1)$, and similarly to the proof of Lemma 3.2, at most two consecutive nodes on the path have equal rank, since the component sizes decrease by at least a factor two for every second node on the path. For a given rank r, consider all components of rank r, where all child components have smaller rank. There are at most $n/2^r$ such components in T_1, since these are disjoint components in T_1 and have size at least 2^r. Since only the parent components of these rank r components could also have rank r, there are at most $n/2^{r-1}$ rank r components in $MCD(T_1)$. Since a component of size at least M has rank at least $\lfloor \log_2 M \rfloor$, it follows that the total number of components in $MCD(T_1)$ of size at least M is at most $\sum_{r=\lfloor \log_2 M \rfloor}^{\infty} n/2^{r-1} = n/2^{\lfloor \log_2 M \rfloor - 2} = O(n/M)$. Furthermore at most $O(n/M)$ components of size less than M are constructed as the result of splitting

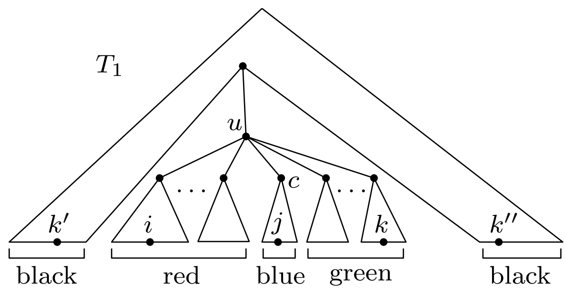


Fig. 9. Coloring of T_1 with respect to edge (u, c). The unresolved triplet ijk and the resolved triplets $ij|k'$ and $ij|k''$ will be anchored in (u, c).

a component of size at least M. Similarly to the proof of Lemma 3.4, the recursion stack to store the recursive contractions of $T_2(u)$ for a component C_u of size at most M requires size at most $O(M)$ and fits into cache, i.e, the total I/O cost for the recursive handling of C_u is $O(1 + |C_u|/B)$ I/Os. Summing over all the $O(n/M)$ maximal disjoint components of $MCD(T_1)$ of size at most M, gives total cost $O(n/M + n/B) = O(n/B)$ I/Os. For handling the $O(n/M)$ components of size at least M, we observe that each leaf of T_1 can be in at most $2\lceil \log_2 \frac{n}{M} \rceil$ recursive components of size at least M, before it is in a component of size at most M. It follows that the total I/O cost for the components of size at least M is $O(n/M + n/B \cdot \log_2 \frac{n}{M})$. Overall, the 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 can have internal nodes with an arbitrary number of children. By anchoring the triplets of T_1 and T_2 in edges instead of nodes, we show that with only four colors we can count all the shared triplets between the two trees. We start by describing a new $O(n^2)$ algorithm for general trees. We then show how we can use the same ideas presented in the previous section to extend the $O(n^2)$ algorithm and reduce the time to $O(n \log n)$.

4.1 Quadratic Algorithm

To anchor the triplets in the edges of a general tree T, we assume an arbitrary left to right ordering of T. Three leaves of T induce a triplet t. If t is an unresolved triplet ijk, assume i is to the left of j, and j is to the left of k. Let u be the lowest common ancestor of i, j, and k, and (u, c) the edge from u to the child c whose subtree contains j. We anchor t in the edge (u, c). If t is a resolved triplet $ij|k$, assume i is to the left of j, and k is either to the left of i or to the right of j (but k cannot be between i and j). Let u be the lowest common ancestor of i and j and (u, c) the edge from u to the child c whose subtree contains j. We anchor t in the edge (u, c) (see Figure 9).

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

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

For the efficient computation of $S(T_1, T_2)$ we use the following coloring procedure: Fix a node u in T_1 and a child c. Color the leaves of every child subtree of u to the left of c red, the leaves of the subtree defined by c blue, the leaves of every child subtree to the right of c green and give the color black to every other leaf of T_1 (see Figure 9). We then transfer this coloring to the leaves of 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 are going to compute all shared triplets that are anchored in the k children edges of v in $O(k)$ time. This will give an $O(n^2)$ total running time, because for every edge in T_1 we spend $O(n)$ time in T_2 and there are $O(n)$ edges in T_1. In v we have the following counters:

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

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

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

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

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

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

4.2 Subquadratic Algorithm

Similarly to the case of binary trees in Section 3, there is a preprocessing step and a counting step. The counting step is divided into two phases, the contraction and counting phase of T_2.

In a preprocessing step we first rearrange the children of each node in T_1 such that the leftmost child has the most leaves. The remaining children are kept in the original order. Next, we recursively transform T_1 into a binary tree $b(T_1)$ (see Figure 10). Let w be an internal node of T_1 with k children. We replace w by a binary left path of w followed by $k-2$ *orange* nodes. We denote w the root of the orange path. The leaves below this path are the k children of w from T_1, in the same left to right

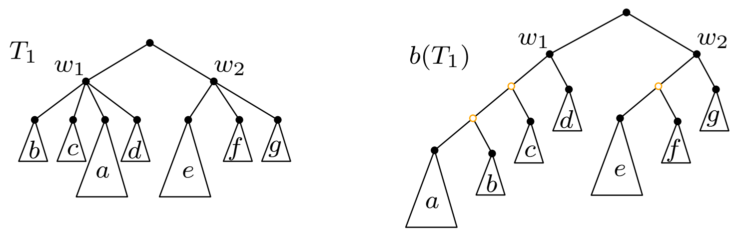


Fig. 10. Transformation of T_1 to $b(T_1)$.

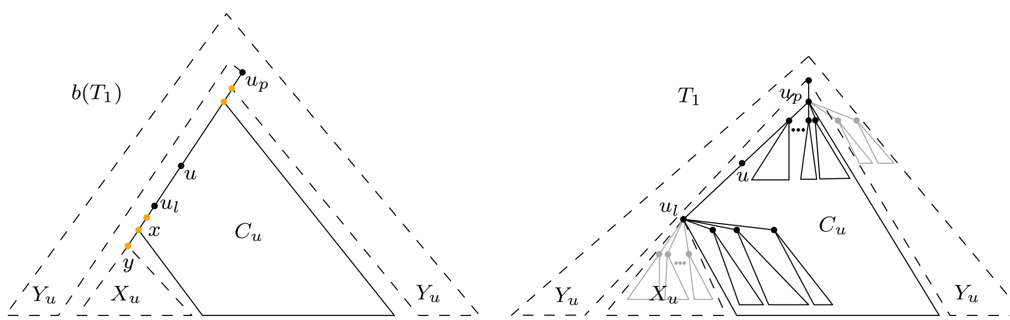


Fig. 11. How a component in $b(T_1)$ translates to a component in T_1.

order. We let node w and its k children from T_1 have the color *black*. Since the preprocessing of T_1 ensures the leftmost child of each node of T_1 has the most most leaves, the resulting tree $b(T_1)$ is left-heavy.

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 $u_{\mathtt{root}}$ be the root of the orange path that u is part of. If u is black, then let $u_{\mathtt{root}} = u$. Again by construction, $u_{\mathtt{root}}$ 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 the edge $(u_{\mathtt{root}}, c)$ of T_1, i.e., $s''(u, c) = s'(u_{\mathtt{root}}, c)$. 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 leaves of $b(T_1)$ are colored according to edge (u, c) as follows: the leaves in the left subtree of u are colored red, the leaves in the right subtree of u are colored blue. If u is an orange node, then the black leaves in the remaining subtrees of the orange path that u is part of are colored green. All other leaves 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 need to make it left-heavy because by construction it already is. However, we change the labels of the leaves in $b(T_1)$ and T_2, so that the leaves in $b(T_1)$ are numbered 1 to n from left to right. The order in which we visit the nodes of $b(T_1)$ is determined 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.

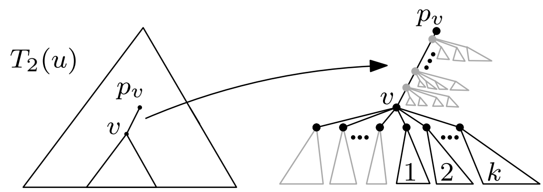


Fig. 12. $T_2(u)$: Contracted children subtrees rooted at node v and contracted subtrees rooted at contracted nodes (gray color) on the edge (p_v, v).

Figure 11 shows a component C_u as the result of applying the MCD algorithm of Section 3 to $b(T_1)$ and the corresponding component in T_1. For an edge (x, y) in $b(T_1)$ crossing the boundary of C_u from below, the node y can either be orange or black. If y is black, the subtree rooted at y corresponds to the leftmost subtree of a node u_l in T_1, whereas if y is orange, the subtree of $b(T_1)$ corresponds to a prefix of the children of u_l in T_1.

Like in the case of binary input trees, 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 between the algorithm for binary trees and the algorithm for general trees, is in the counters that we have to maintain in the contracted versions of T_2. Otherwise, the same analysis from Section 3 holds.

4.2.1 Contraction Phase of T_2. The contraction of T_2 with respect to a set of leaves $\Lambda \subseteq L(T_2)$, happens in the exact same way as described in Section 3, i.e., we start by pruning all leaves 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 $T_2(u)$, with leaf sets $L(T_2(u)) = L(C_u)$. Like in the binary algorithm of Section 3, to count the shared triplets anchored in an edge (u, c) in T_1, 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.

The colors of the leaves that were contracted when constructing $T_2(u)$ will all be stored in counters (details are in Section 4.2.2). The color of each contracted leaf depends on the type of the corresponding component that we have in $b(T_1)$ and the splitting node that is used for that component. For example, in Figure 11 the contracted leaves from X_u will be red because $b(T_1)$ is left-heavy. The contracted leaves from the children subtrees of u_p in T_1 can either be green or black: If u in $b(T_1)$ happens to be orange and part of the orange path 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.

4.2.2 Counting Phase of T_2. In Figure 12 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 leaves. The counters that we 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 the edge (p_v, v). Then, the time required by the counting phase becomes $O(|T_2(u)|)$, giving the same time bounds as in the binary algorithm of Section 3. In v we have the following counters:

- v_{red}: total number of red leaves (including the contracted leaves) in the subtree of v.

- $v_{\mathtt{blue}}$: total number of blue leaves in the subtree of v.
- $v_{\mathtt{green}}$: total number of green leaves (including the contracted leaves) in the subtree of v.
- $\overline{v}_{\mathtt{black}}$: total number of black leaves (including the contracted leaves) not in the subtree of v.

We divide the rest of the counters into two categories: Category A corresponds to the leaves in the contracted children subtrees of v and each counter is stored in a variable of the form $v_{A.x}$. Category B corresponds to the leaves in the contracted subtrees on the edge (p_v, v), and each counter is stored in a variable of the form $v_{B.x}$.

For category A we have the following counters:

- $v_{A.\mathtt{red}}$: total number of red leaves in the contracted children subtrees of v.
- $v_{A.\mathtt{green}}$: total number of green leaves in the contracted children subtrees of v.
- $v_{A.\mathtt{black}}$: total number of black leaves in the contracted children subtrees of v.
- $v_{A.\mathtt{red,green}}$: total number of pairs of leaves 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 m-th child of v, we also maintain the following:

- $a_{\mathtt{red}}$: total number of red leaves from the first $m-1$ children subtrees, including the contracted children subtrees.
- $a_{\mathtt{blue}}$: total number of blue leaves from the first $m-1$ children subtrees.
- $a_{\mathtt{green}}$: total number of green leaves from the first $m-1$ children subtrees, including the contracted children subtrees.
- $p_{\mathtt{red,green}}$: total number of pairs of leaves from the first $m-1$ children subtrees, including the 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).
- $p_{\mathtt{red,blue}}$: total number of pairs of leaves from the first $m-1$ children subtrees, including the 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).
- $p_{\mathtt{blue,green}}$: total number of pairs of leaves from the first $m-1$ children subtrees, including the 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).
- $t_{\mathtt{red,blue,green}}$: 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 leaves come from different subtrees (some might be contracted, some might be non-contracted).

Every variable is updated in $O(1)$ time in exactly the same manner like in the $O(n^2)$ algorithm of Section 4.1. The main difference is in the values of the variables before we begin scanning the children edges of v. Every variable is initialized as follows:

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

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

The category B counters helps us count triplets involving leaves (contracted and non-contracted) from the subtree of v and leaves from the contracted subtrees rooted at the edge (p_v, v). We maintain the following:

- $v_{B.\texttt{red}}$: total number of red leaves in all contracted subtrees rooted at the edge (p_v, v).
- $v_{B.\texttt{green}}$: total number of green leaves in all contracted subtrees rooted at the edge (p_v, v).
- $v_{B.\texttt{black}}$: total number of black leaves in all contracted subtrees rooted at the edge (p_v, v).
- $v_{B.\texttt{red,green}}$: total number of pairs of leaves 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 leaves 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'', where v'' is closer to p_v 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 (p_v, 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 (p_v, v), we have that $\texttt{tot}_{B.\texttt{res}} = v_{\texttt{blue}} \cdot v_{B.\texttt{red,black}} + v_{\texttt{blue}} \cdot v_{B.\texttt{red}} \cdot (\overline{v}_{\texttt{black}} - v_{B.\texttt{black}})$.

4.3 Scaling to External Memory

The analysis is the same as in Section 3, except for minor details. The proof of Lemma 3.4 can be trivially modified to apply to general trees as well. Finally, Lemma 3.5 is generalized to non-binary trees in the following Lemma 4.1. In Lemma 4.1, we consider the I/Os required to apply a depth-first traversal on a non-binary tree T that is stored in memory following a local layout, i.e., the nodes of every subtree of T are stored consecutively in memory and every node has at most two occurrences in memory, before the first child subtree and/or after the last child subtree (see Figure 13). Similarly to the assumptions we made for Lemma 3.5, w.l.o.g. we assume that when an edge (u, v) of T is processed in a depth-first traversal of T, both u and v are visited, i.e., both u and v are in cache.

Lemma 4.1. *Let T be a non-binary tree with n leaves that is stored in an array following a local layout, i.e., the nodes of every subtree of T are stored consecutively in memory and every node has at most two occurrences in memory. Any depth-first traversal that starts from the root of T and in which for every internal node u in T, after the traversal of the first child of u the remaining children are traversed in the order that they appear in memory from left to right, requires* $O(n/B)$ *I/Os in the cache oblivious model.*

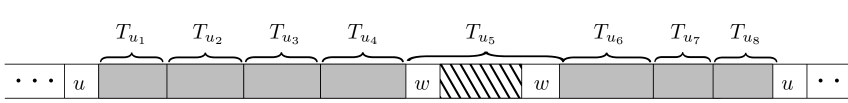


Fig. 13. Position of a node u in memory with respect to the 8 subtrees defined by the children of u, with T_{u_5} being a largest subtree.

Proof. This proof is an extension of the proof of Lemma 3.5. For a node u in T, let T_u denote the set of nodes in the subtree rooted at u, and $T_{u_1}, \dots, T_{u_i}$ the subtrees rooted at the children $u_1, \dots, u_i$ of u. We assume that these subtrees are ordered from left to right in order that they appear in memory, and u is stored before the first child subtree and after the last child subtree (see Figure 13). In the proof of Lemma 3.5, we implicitly assumed that the positions of the two children of u are

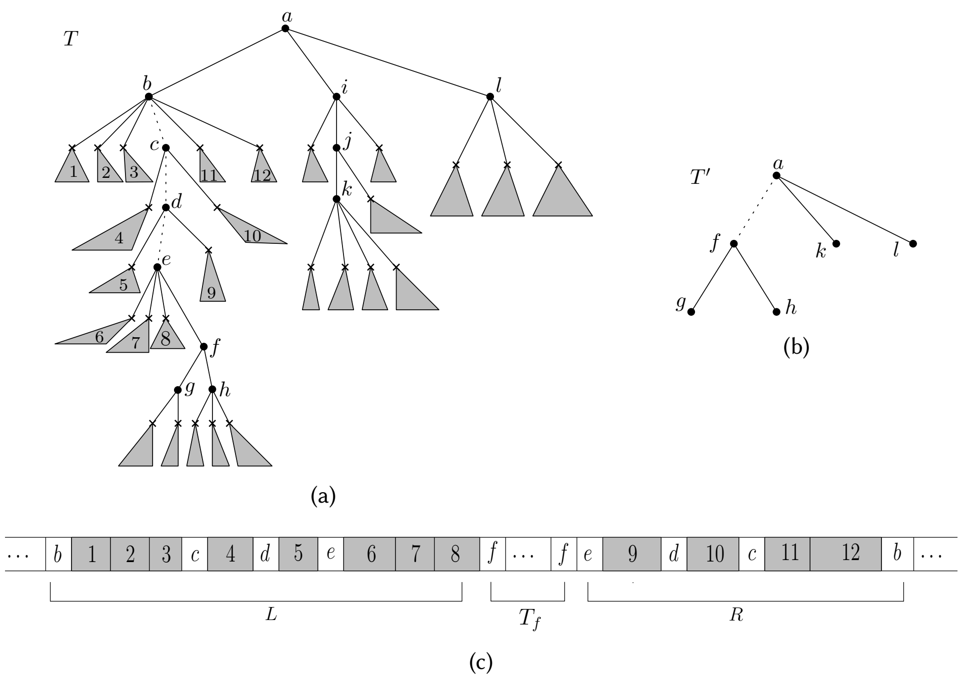


Fig. 14. (a) A general tree T. The gray subtrees are B-light subtrees and every node not in a B-light subtree is a B-heavy node. (b) The corresponding tree T' according to the proof of Lemma 4.1. (c) How T is stored in memory and the two segments of memory that correspond to the edge (a, f) in T'.

stored together with u in memory. For general trees, together with u we need to store a list of arbitrary size i containing the positions in memory of every child of u. To avoid complicating the presentation of the proof, we assume that we can find the position in memory of every child of u without this list, i.e., this list is not stored together with u, thus finding the position of any child of u incurs no I/Os. An easy way to support this is to store in every node u in T, one pointer to the first child and one pointer to the sibling appearing next in memory.

Define a node u in T to be *B-light* if $2|T_u| \leq B - 2$, otherwise the node is said to be *B-heavy* (see Figure 14a). Observe that the children of a B-light node are all B-light. We consider the following disjoint partition of the sets of nodes from T:

S_1: B-light nodes,
S_2: B-heavy nodes with no B-heavy child,
S_3: B-heavy nodes with at least two B-heavy children, and
S_4: B-heavy nodes with exactly one B-heavy child.

For a B-light node w' with B-heavy parent w in T, we say that $T_{w'}$ is a B-light subtree. The node w can be either in S_2, S_3, or S_4. Since $2|T_{w'}| \leq B - 2$, at most $O(1)$ I/O are sufficient to visit all nodes in $T_{w'}$. Below we charge these I/Os to visiting w.

Similarly to the proof of Lemma 3.5, we have that $|S_2| = O(n/B)$ and $|S_3| = O(n/B)$. Let T' be defined as in the proof of Lemma 3.5, as well as $P_{(u,v)}$ and $C_{(u,v)}$ for an edge (u, v) in T'. Since T is non-binary, we have to argue that the number of I/Os spent traversing the B-light subtrees that are rooted at every node in S_2 and S_3 is $O(n/B)$. For a node u in T, let G_u be the size of all B-light subtrees rooted at a child of u. For every node u in S_2, all children are B-light subtrees. We spend at

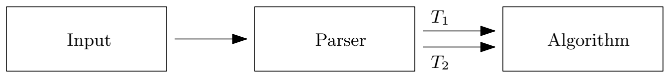


Fig. 15. Implementation overview.

most O(1) I/Os to traverse the first child subtree of u and $O(1+G_u/B)$ I/Os to traverse the remaining B-light subtrees left to right, thus $O(1 + G_u/B)$ I/Os in total to traverse T_u. Since $|S_2| = O(n/B)$ and the B-light subtrees in T are disjoint, i.e., $\sum_{u \in T} G_u = O(n)$, we spend in total $O(n/B)$ I/Os traversing the subtrees rooted at the nodes in S_2. For a node u in S_3, let $d_H(u)$ denote the number of B-heavy children of u. For a B-heavy node u, the number of consecutive groups of B-light subtrees rooted at a child of u are at most $1 + d_H(u)$. The I/O cost of traversing the B-light subtrees rooted at a child of u is $O(1 + d_H(u) + G_u/B)$, where the +1 comes from the I/Os to traverse the first B-light child (if the first child visited is B-light, then this can by anywhere in the layout of the children), $+d_H(u)$ for traversing the first B-light subtree in each group, and $+G_u/B$ to traverse the remaining B-light trees from left to right. Since $|S_3| = O(n/B)$, we have $\sum_{u \in T'} d_H(u) = O(n/B)$. Together with the fact that $\sum_{u \in T} G_u = O(n)$, we spend $O(n/B)$ I/Os traversing the B-light subtrees rooted at every node in S_3.

We now argue that the total number of I/Os incurred by the nodes in S_4 and the B-light subtrees rooted at the children of nodes in S_4 is $O(n/B)$, thus proving the statement. By the local layout, the nodes in $C_{(u,v)}$ are stored in two segments of memory L and R to the left and right of the layout of T_v, respectively (see Figure 14c). Let w be a node in $P_{(u,v)}$ and G_w be the total size of the B-light subtrees rooted at a child of w. We say that w is G-light if $2G_w \leq B - 2$, otherwise G-heavy. There can be $O(n/B)$ G-heavy nodes in T, thus by the same argument as in the previous paragraph, scanning the B-light subtrees for all G-heavy nodes together incurs $O(n/B)$ I/Os. For the G-light nodes we follow a similar argument as in the proof of Lemma 3.5. and w.l.o.g. assume that every node in $P_{(u,v)}$ is G-light. During the depth-first traversal we on the way down along $P_{(u,v)}$ alternate to visit L from left to right and R from right to left, and then on the way up along $P_{(u,v)}$ alternate to visit L from right to left and R and from left to right. Let c be the child of w that is B-heavy. Since for every node w in $P_{(u,v)}$ we have $2G_w \leq B - 2$, by accessing both copies of w and c when c is visited in a depth-first traversal of T, we guarantee that all the B-light subtrees rooted at w are in cache, i.e., they can be accessed in memory for free. Hence, $O(n/B)$ I/Os are sufficient to pay to traverse the B-light subtrees of all G-light nodes. □

5 IMPLEMENTATION

The algorithms of Sections 3 and 4 have been implemented in the C++ programming language. A high level overview of each implementation is illustrated in Figure 15. The source code is publicly available and can be found at 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 (that is a local layout). Both trees have n leaves and the label of each leaf is assumed to be a number in $\{1, 2, \ldots, n\}$. Two leaves cannot 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 in an array following the preorder layout and T_2 in an array following the postorder layout. The parser takes $O(n)$ time and space in the RAM model and $O(n/B)$ I/Os in the cache oblivious model.

5.3 Algorithm

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

5.3.1 Initialization. In the initialization step, the preprocessing parts of the algorithms are performed (see Sections 3.3 and 4.2), where 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. After this step, the first component of T_1 is stored in an array (different than the one produced by the parser) following the preorder layout. Similarly, the corresponding component of T_2 is stored in an array following the postorder layout.

5.3.2 Distance Computation. Let $\texttt{comp}(T_1)$ and $\texttt{comp}(T_2)$ be the component of T_1 and the corresponding component of T_2 produced by the initialization step. Having these two components available, we can begin counting shared triplets in order to compute $S(T_1, T_2)$. The following steps are recursively applied:

- Starting from the root of $\texttt{comp}(T_1)$ and according to Section 3.2, scan the leftmost path of $\texttt{comp}(T_1)$ to find the splitting node u.
- Scan $\texttt{comp}(T_2)$ to compute for the binary algorithm $\sum_{v \in T_2} |s(u) \cap s(v)|$ (see counting phase of T_2 in Section 3.3), or for the general algorithm $\sum_{(v,c') \in T_2} |s''(u,c) \cap s'(v,c')|$ (see counting phase of T_2 in Section 4.2).
- Using the splitting node u, generate the next three components of T_1. Let $\texttt{comp}(T_1(u_l))$, $\texttt{comp}(T_1(u_r))$, and $\texttt{comp}(T_1(u_p))$ be the components determined by the left child, right child, and parent of u respectively. Let $\texttt{comp}(T_2(u_l))$, $\texttt{comp}(T_2(u_r))$ and $\texttt{comp}(T_2(u_p))$ be the corresponding contracted versions of T_2 with all the necessary counters properly maintained (see contraction phase of T_2 in Section 3.3 for the binary case and in Section 4.2 for the general case).
- Scan and contract $\texttt{comp}(T_2)$ to generate $\texttt{comp}(T_2(u_l))$ and then recurse on the pair defined by $\texttt{comp}(T_1(u_l))$ and $\texttt{comp}(T_2(u_l))$.
- Scan and contract $\texttt{comp}(T_2)$ to generate $\texttt{comp}(T_2(u_r))$ and then recurse on the pair defined by $\texttt{comp}(T_1(u_r))$ and $\texttt{comp}(T_2(u_r))$.
- Scan and contract $\texttt{comp}(T_2)$ to generate $\texttt{comp}(T_2(u_p))$ and then recurse on the pair defined by $\texttt{comp}(T_1(u_p))$ and $\texttt{comp}(T_2(u_p))$.

As a final step, print $\binom{n}{3} - S(T_1, T_2)$, which is equal to the triplet distance $D(T_1, T_2)$.

5.3.3 Correctness. The correctness of our implementations 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 $O(n \log^3 n)$ algorithm in [14] and the $O(n \log n)$ algorithm in [19].

5.3.4 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 takes $O(n \log n)$ 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 in the preprocessing step that stores the new labels of the leaves in T_1, which we then use to update the leaf labels of T_2. In the RAM model this second approach takes $O(n)$ time but in the cache oblivious model O(n) I/Os. 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 provide an extensive experimental evaluation of 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 16.04.2 LTS. The compilers used were g++ 5.4 and g++ 4.7 with optimization level -O3, together with cmake 3.5.1. The experiments were performed in text mode, i.e., by booting into the terminal of Ubuntu, to minimize the interference from other programs running at the same time.

6.1.1 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 leaves in this model is generated as follows:

- Create a binary tree T with n leaves as follows: start with a binary tree T with two leaves. Iteratively pick $n-1$ times 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 leaves 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.

Jansson *et al.* used similar input by contracting nodes of a random binary tree, although their initial random binary trees were generated using the uniform model [15].

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 leaves in this model is generated as follows:

- Create a binary tree T with n leaves 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 at 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., if n' is the number of leaves below $T(u)$, and $|\Lambda_l|$ and $|\Lambda_r|$ are the number of leaves in $T(l)$ and $T(r)$ respectively, first choose $|\Lambda_l| = \max(1, \min(\lfloor \alpha \cdot n' \rfloor, n' - 1))$ and then let $|\Lambda_r| = n' - |\Lambda_l|$.
- With probability p contract every internal node u of T' like in the random model.

Holt *et al.* [13] only considered perfectly balanced input trees, i.e., the special case $\alpha = 0.5$.

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

6.1.2 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 the experiments we use the algorithm from Section 3. In every other case, 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, see Figure 16) compared to the algorithm from Section 3 that comes due to the additional counters that we maintain in the contractions of T_2.

We compared our implementation with previous implementations of [14] and [5, 18] available at http://sunflower.kuicr.kyoto-u.ac.jp/~jj/Software/Software.html and http://users-cs.au.dk/cstorm/

[1] http://www.cplusplus.com/reference/algorithm/shuffle/

[2] http://www.cplusplus.com/reference/random/default_random_engine/

software/tqdist/ respectively. The implementation of the $O(n \log^3 n)$ algorithm in [14] 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 [19], which we refer to as `tqDist`, has an implementation of the binary $O(n \log^2 n)$ algorithm from [18] and the general $O(n \log n)$ algorithm from [5]. If the two input trees are binary the $O(n \log^2 n)$ algorithm is used (since [13] showed that for binary trees the $O(n \log^2 n)$ algorithm had a better practical performance than the $O(n \log n)$ algorithm). We refer to our new algorithm as `CacheTD`.

6.1.3 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 include the time taken to parse the input trees in our plots. We used the PAPI library[5] for statistics related to instructions, 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`.

On typical input the parsing time of our algorithm `CacheTD` was about 50% the parsing time of `tqDist` on the same input, and 75% of the parsing time of `CPDT` and `CPDTg`. On input trees with more than 1000 nodes the parsing time of `CacheTD` was about 20 – 25% of the total running time. The initial relabelling (using a lookup table for the relabelling that fits into internal memory) and construction of the components at the root of $MCD(T_1)$ took about 10% of the computation time of `CacheTD`.

6.2 Results

The experiments are divided into two parts. In the first part, we consider the performance of the algorithms when their memory requirements do not exceed the available main memory (8G RAM). In the second part, we consider the performance 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 start using the swap space, which in turn yields the very expensive disk I/Os. All figures can be found in Appendix A.

6.2.1 RAM experiments in the Random Model. In Figure 17 we illustrate a time comparison of all implementations for trees of up to 2^{21} leaves ($\sim$ 2 million) with varying contraction probabilities. Every experiment is run 10 times, and each time on a different tree. All 10 data points are depicted together with a line that goes over their median. The compilers used were g++ 5.4 with cmake 3.5.1 for `tqDist` and g++ 5.4 for `CPDT`, `CPDTg`, and `CacheTD`. In all cases, `CacheTD` achieves 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 [14]. The same can be observed for the case where $p_1 = 0.8$ and $p_2 = 0.8$. The reason is of the differences in the implementation of `unordered_map` that exist between the different versions of the g++ compilers. In Figure 18 we compare the performance of `CPDT` when compiled with g++ 4.7 and g++ 5.4. When p_1 is large, i.e., $p_1 = 0.8$ and $p_1 = 0.95$, we observe that the older version of g++ achieves a better performance. For all other values of p_1, the version of the compiler has no effect on the performance. In Figure 19 we have another time comparison of all implementations but now with `CPDT` compiled in g++ 4.7. The new algorithm achieves the best performance again, but now the behaviour of `CPDT` is more stable when p_1 is large. From now on, in every RAM experiment `CPDT` is compiled in g++ 4.7.

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

In Figure 20 we show 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 21 and 22 we can see how the contraction parameter affects the running time and the space consumption of the algorithms respectively.

Finally, in Figures 23, 24 and 25 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.

6.2.2 RAM experiments in the Skewed Model. The main interesting experimental results are illustrated in Figure 26, 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 in every graph of Figure 26 being only a factor of 1.15 larger than the minimum. As mentioned in Section 2, CPDT and CPDTg use the heavy-light decomposition for T_2. For binary trees, when α approaches 0 or 1, the number of heavy paths that have to be updated because of a leaf color change decreases, thus the total number of operations of the algorithm decreases as well. We can verify this in Figure 27, where we have the plots of the alpha parameter against the instructions. The same cannot be said for all general trees, since the contraction parameters have an effect on the shape of the trees as well. In Figures 28, 29, and 30 we have the same graphs but for L1, L2, and L3 cache misses respectively.

Table 2. Random model: Time performance when limiting the available RAM to be 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	tqDist	CacheTD
2^{15}	0m:01s	0m:01s	0m:01s
2^{16}	0m:01s	0m:02s	0m:01s
2^{17}	0m:01s	0m:04s	0m:01s
2^{18}	0m:02s	1m:03s	0m:01s
2^{19}	0m:04s	1h:21m	0m:01s
2^{20}	0m:09s	**0%**	0m:01s
2^{21}	13m:12s	-	0m:03s
2^{22}	**0%**	-	0m:09s
2^{23}	-	-	3m:37s
2^{24}	-	-	10m:35s

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	0m:07s	0m:01s
2^{19}	0m:07s	0m:07s	5m:20s	0m:01s
2^{20}	3m:43s	1h:13m	**0%**	0m:02s
2^{21}	**15%**	**0%**	-	0m:20s
2^{22}	-	-	-	2m:02s
2^{23}	-	-	-	10m:42s
2^{24}	-	-	-	42m:06s

6.2.3 I/O experiments. In Figures 31 and 32 we illustrate the time, space, and I/O performance in the random and skewed model respectively. Every implementation was compiled with g++ 5.4. Every experiment is run 5 times, each on a different tree. Like in the RAM experiments, all 5 data points are displayed together with a line that passes through the median. To measure the execution time, we used the `time` function of Ubuntu and thus also took into account the time taken to parse the input trees. For the input trees of size 2^{23} and 2^{24} we used the 128 bit implementation of the new algorithms in order to avoid overflows.

Unlike CacheTD, the performance of CPDT, CPDTg, and tqDist deteriorates significantly from the moment they start performing disk I/Os. Only CacheTD managed to finish running in a reasonable amount of time for all input sizes. For every other algorithm, some data points are missing because

Table 3. Skewed model: 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	tqDist	CacheTD
2^{15}	0m:01s	0m:01s	0m:01s
2^{16}	0m:01s	0m:02s	0m:01s
2^{17}	0m:01s	0m:05s	0m:01s
2^{18}	0m:02s	0m:54s	0m:01s
2^{19}	0m:05s	50m:38s	0m:01s
2^{20}	0m:13s	**0%**	0m:01s
2^{21}	20m:02s	-	0m:03s
2^{22}	**0%**	-	0m:09s
2^{23}	-	-	3m:46s
2^{24}	-	-	13m:36s

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	0m:06s	0m:01s
2^{19}	0m:07s	0m:07s	3m:21s	0m:01s
2^{20}	6m:24s	2h:31m	**7h:51m**	0m:02s
2^{21}	**12%**	**0%**	-	0m:19s
2^{22}	-	-	-	1m:58s
2^{23}	-	-	-	9m:42s
2^{24}	-	-	-	38m:19s

the execution time required was too big. To get an idea of how big, in Tables 2 and 3 we again have the time performance of the algorithms in the random and skewed models respectively. This is the exact same time performance as depicted in Figures 31 and 32, however we also include some information about how well the algorithms performed on the extra data point that is missing from the figures. We set a time limit of 10 hours, and only for one pair of input trees T_1 and T_2 we measured for how many nodes of T_1 the value of $\sum_{v \in T_2} |s(u) \cap s(v)|$ was found. Some algorithms managed to process only 0% of the total nodes in T_1, which means that they had to spend most of the time in the preprocessing step (e.g. constructing the HDT of T_2). The only algorithm that managed to produce a result was `tqDist`, requiring close to 8 hours for trees with 2^{20} leaves (see Table 3).

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 and open problems involve the following:

- Could the new algorithms be improved so that in the analysis, the base of the logarithm becomes M/B, thus giving the sorting bound in the cache oblivious model? Would the resulting algorithm be even more efficient in 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 Section 3 and 4 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. 1988. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM* 31, 9 (1988), 1116–1127. https://doi.org/10.1145/48529.48535

[2] Lars Arge, Gerth Stølting Brodal, Jakob Truelsen, and Constantinos Tsirogiannis. 2013. An Optimal and Practical Cache-Oblivious Algorithm for Computing Multiresolution Rasters. In *Proceedings 21st Annual European Symposium on*

Algorithms (Lecture Notes in Computer Science), Hans L. Bodlaender and Giuseppe F. Italiano (Eds.), Vol. 8125. Springer, 61–72. https://doi.org/10.1007/978-3-642-40450-4_6

[3] M. S. Bansal, J. Dong, and D. Fernández-Baca. 2011. Comparing and Aggregating Partially Resolved Trees. *Theoretical Computer Science* 412, 48 (2011), 6634–6652. https://doi.org/10.1016/j.tcs.2011.08.027

[4] V. Berry and O. Gascuel. 2000. Inferring Evolutionary Trees with Strong Combinatorial Evidence. *Theoretical Computer Science* 240, 2 (2000), 271–298. https://doi.org/10.1016/S0304-3975(99)00235-2

[5] G. S. Brodal, R. Fagerberg, C. N. S. Pedersen, T. Mailund, and A. Sand. 2013. Efficient Algorithms for Computing the Triplet and Quartet Distance Between Trees of Arbitrary Degree. In *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 1814–1832. https://doi.org/10.1137/1.9781611973105.130

[6] Gerth Stølting Brodal, Rolf Fagerberg, and Kristoffer Vinther. 2007. Engineering a cache-oblivious sorting algorithm. *ACM Journal of Experimental Algorithmics* 12, Article No. 2.2 (2007), 1–23. https://doi.org/10.1145/1227161.1227164

[7] D. E. Critchlow, D. K. Pearl, and C. L. Qian. 1996. The Triples Distance for Rooted Bifurcating Phylogenetic Trees. *Systematic Biology* 45, 3 (1996), 323–334. https://doi.org/10.1093/sysbio/45.3.323

[8] W. H. E. Day. 1985. Optimal Algorithms for Comparing Trees with Labeled Leaves. *Journal of Classification* 2, 1 (1985), 7–28. https://doi.org/10.1007/BF01908061

[9] A. J. Dobson. 1975. Comparing the shapes of trees. In *Combinatorial Mathematics III*, A. P. Street and W. D. Wallis (Eds.). Lecture Notes in Mathematics, Vol. 452. Springer Berlin Heidelberg, 95–100. https://doi.org/10.1007/BFb0069548

[10] Bartlomiej Dudek and Pawel Gawrychowski. 2019. Computing quartet distance is equivalent to counting 4-cycles. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC*, Moses Charikar and Edith Cohen (Eds.). ACM, 733–743. https://doi.org/10.1145/3313276.3316390

[11] G. F. Estabrook, F. R. McMorris, and C. A. Meacham. 1985. Comparison of Undirected Phylogenetic Trees Based on Subtrees of Four Evolutionary Units. *Systematic Zoology* 34, 2 (1985), 193–200. https://doi.org/10.2307/2413326

[12] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. 1999. Cache-Oblivious Algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, 285–297. https://doi.org/10.1109/SFFCS.1999.814600

[13] M. K. Holt, J. Johansen, and G. S. Brodal. 2014. On the Scalability of Computing Triplet and Quartet Distances. In *Proceedings of the 16th Workshop on Algorithm Engineering and Experiments*. Society for Industrial and Applied Mathematics, 9–19. https://doi.org/10.1137/1.9781611973198.2

[14] J. Jansson and R. Rajaby. 2015. A More Practical Algorithm for the Rooted Triplet Distance. In *Proceedings of the 2nd International Conference on Algorithms for Computational Biology*. Springer International Publishing, 109–125. https://doi.org/10.1007/978-3-319-21233-3_9

[15] J. Jansson and R. Rajaby. 2017. A More Practical Algorithm for the Rooted Triplet Distance. *Journal of Computational Biology* 24, 2 (2017), 106–126. https://doi.org/10.1089/cmb.2016.0185

[16] D. F. Robinson and L. R. Foulds. 1981. Comparison of Phylogenetic trees. *Mathematical Biosciences* 53, 1 (1981), 131–147. https://doi.org/10.1016/0025-5564(81)90043-2

[17] N. Saitou and M. Nei. 1987. The Neighbor-Joining Method: A New Method for Reconstructing Phylogenetic Trees. *Molecular Biology and Evolution* 4, 4 (1987), 406. https://doi.org/10.1093/oxfordjournals.molbev.a040454

[18] A. Sand, G. S. Brodal, R. Fagerberg, C. N. S. Pedersen, and T. Mailund. 2013. A practical $O(n \log^2 n)$ time algorithm for computing the triplet distance on binary trees. *BMC Bioinformatics* 14, 2 (2013), S18. https://doi.org/10.1186/1471-2105-14-S2-S18

[19] A. Sand, M. K. Holt, J. Johansen, G. S. Brodal, T. Mailund, and C. N. S. Pedersen. 2014. tqDist: A Library for Computing the Quartet and Triplet Distances Between Binary or General Trees. *Bioinformatics* 30, 14 (2014), 2079. https://doi.org/10.1093/bioinformatics/btu157

[20] A. Sand, M. K. Holt, J. Johansen, R. Fagerberg, G. S. Brodal, C. N. S. Pedersen, and T. Mailund. 2013. Algorithms for Computing the Triplet and Quartet Distances for Binary and General Trees. *Biology - Special Issue on Developments in Bioinformatic Algorithms* 2, 4 (2013), 1189–1209. https://doi.org/10.3390/biology2041189

[21] J. von Neumann. 1993. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing* 15, 4 (1993), 27–75. https://doi.org/10.1109/85.238389

A EXPERIMENT FIGURES

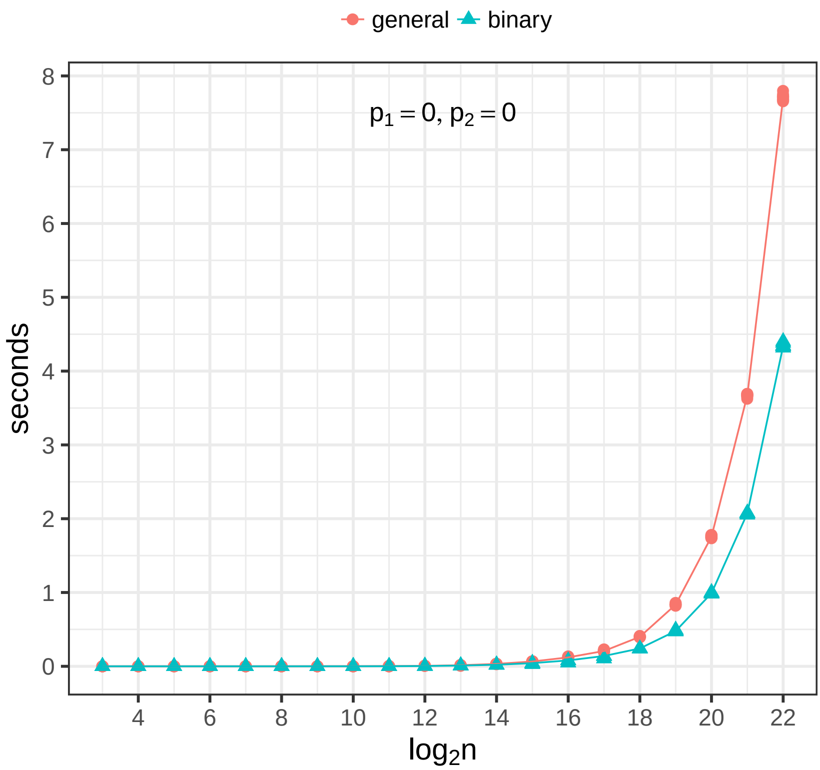


Fig. 16. CacheTD: performance of binary (Section 3) and general (Section 4) implementation on binary trees. All data points of the 10 runs are visible in the figure. Each run is on a different tree and the line connects the median of the runs.

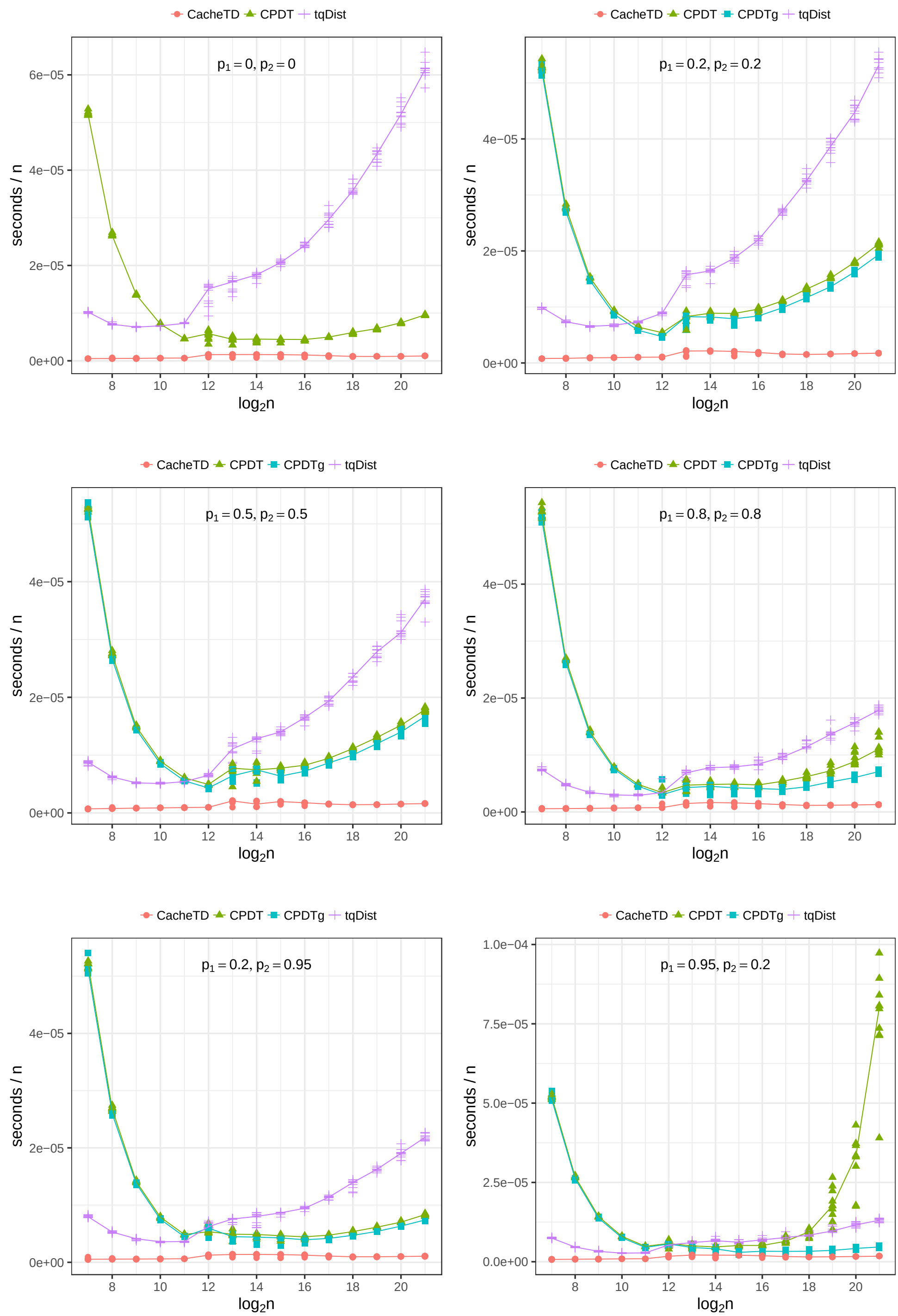


Fig. 17. Random model: Time performance, where CPDT is compiled in g++ version 5.4.

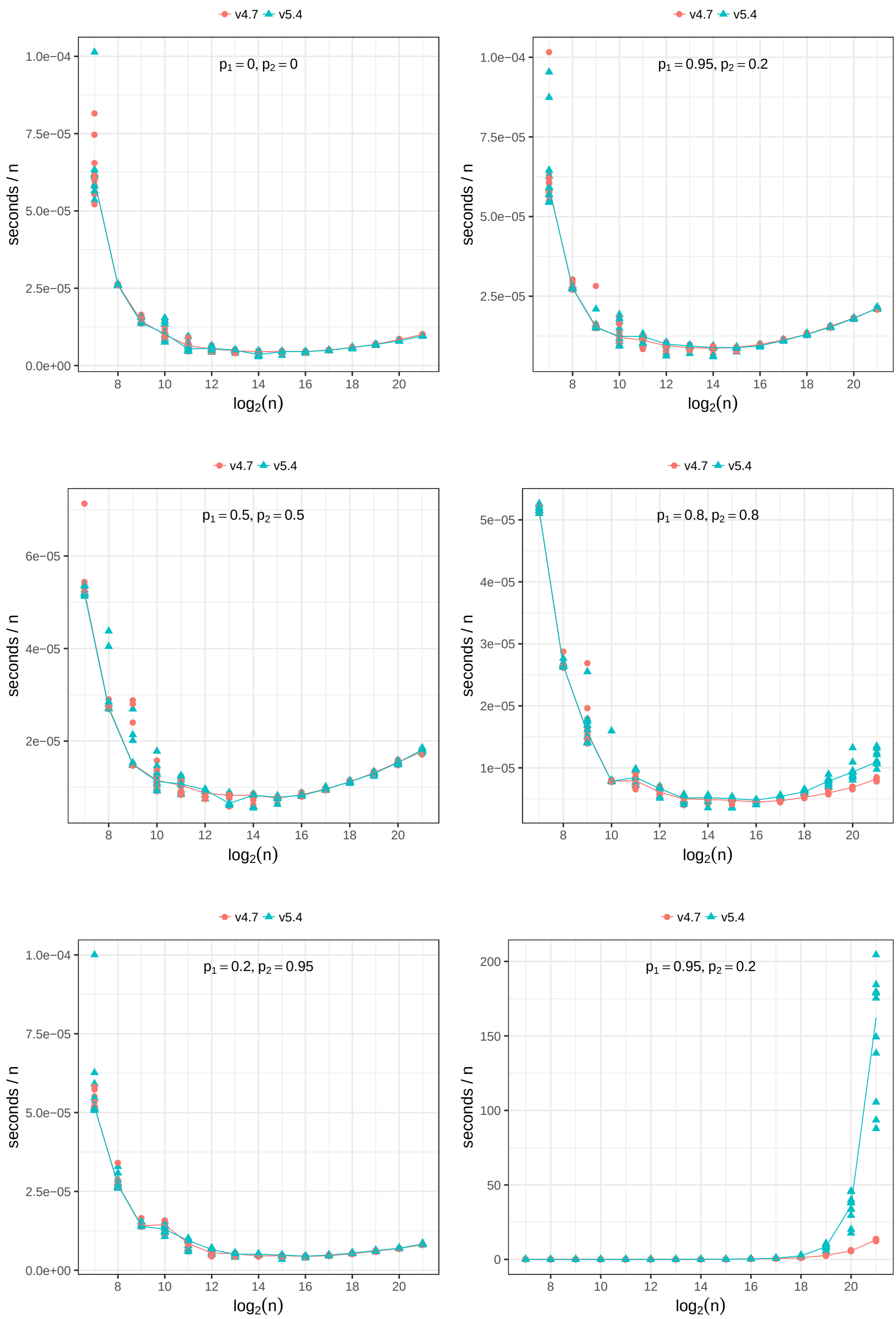


Fig. 18. Random model: Time performance of CPDT when compiled with g++ 4.7 and g++ 5.4.

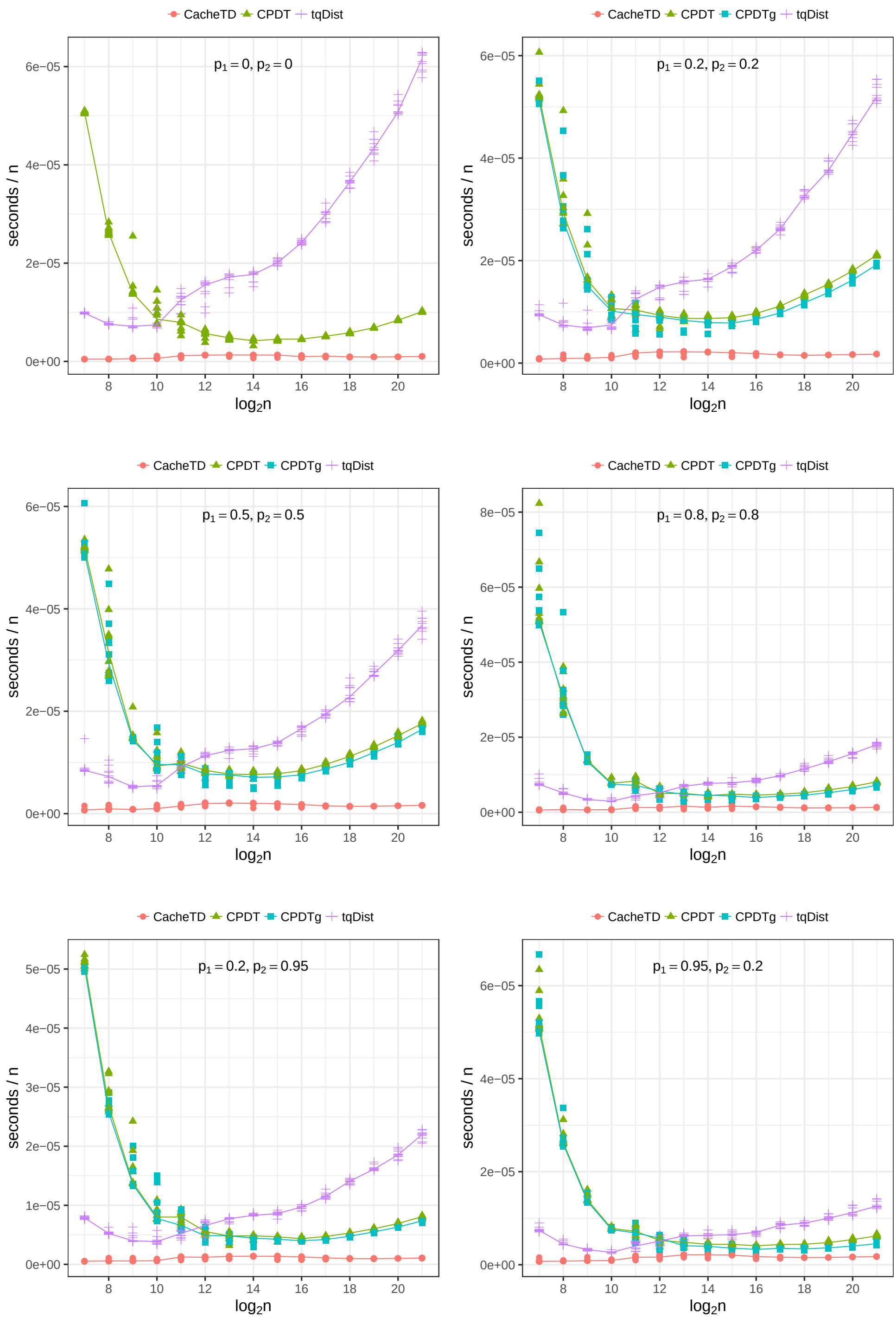


Fig. 19. Random model: Time performance, where CPDT is compiled in g++ version 4.7.

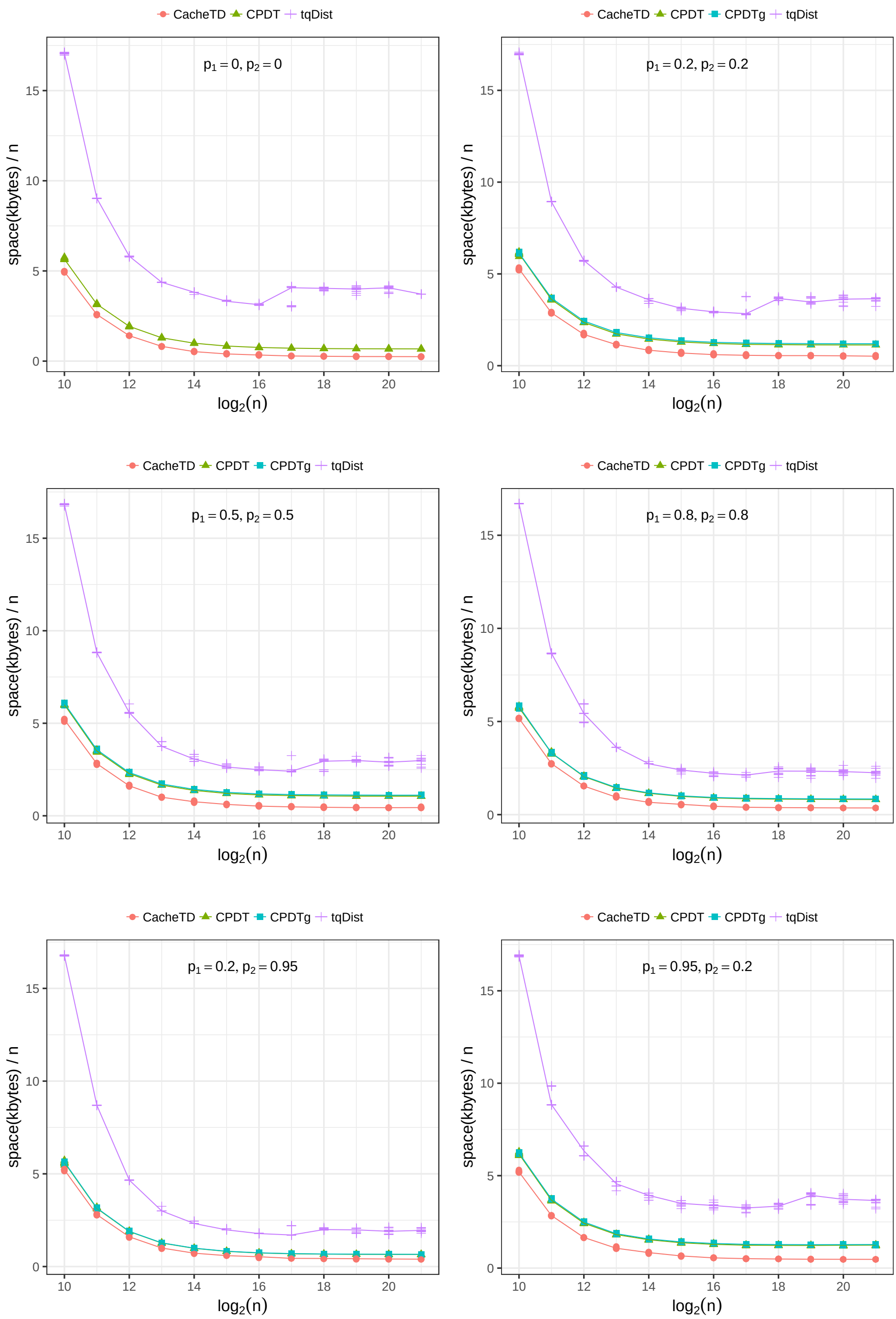


Fig. 20. Random model: Space performance.

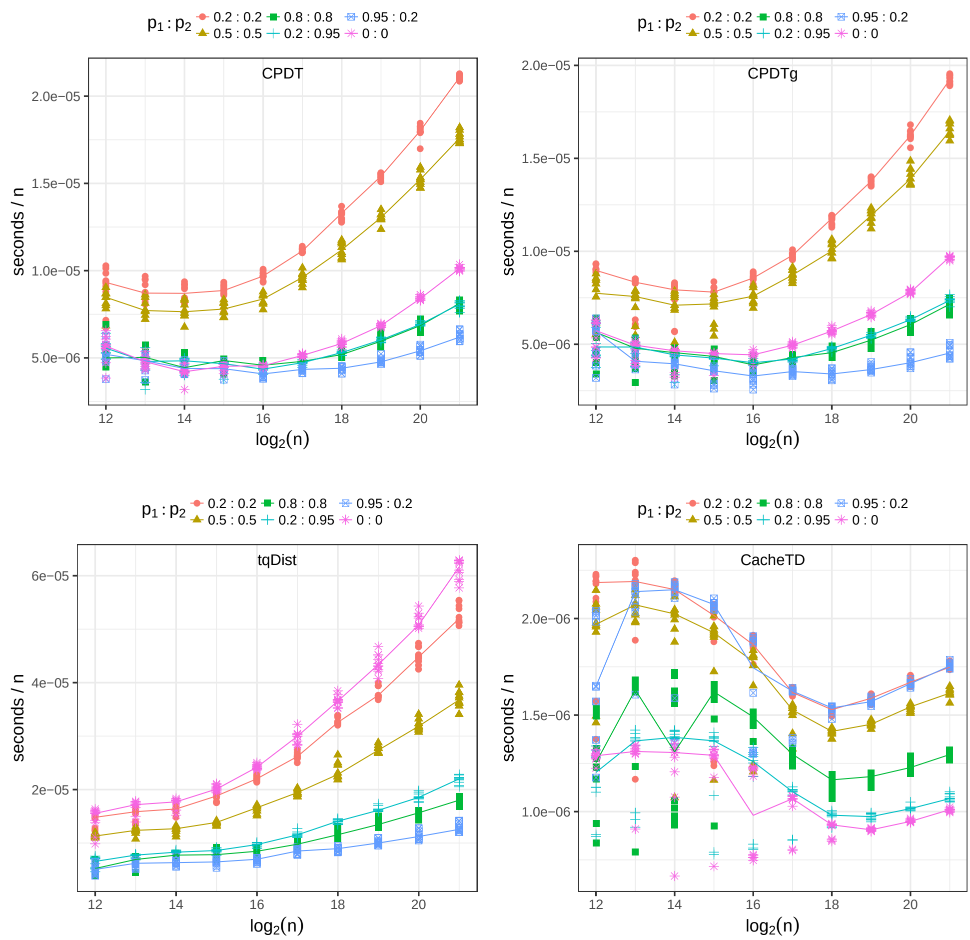


Fig. 21. Random model: How the contraction parameter affects execution time.

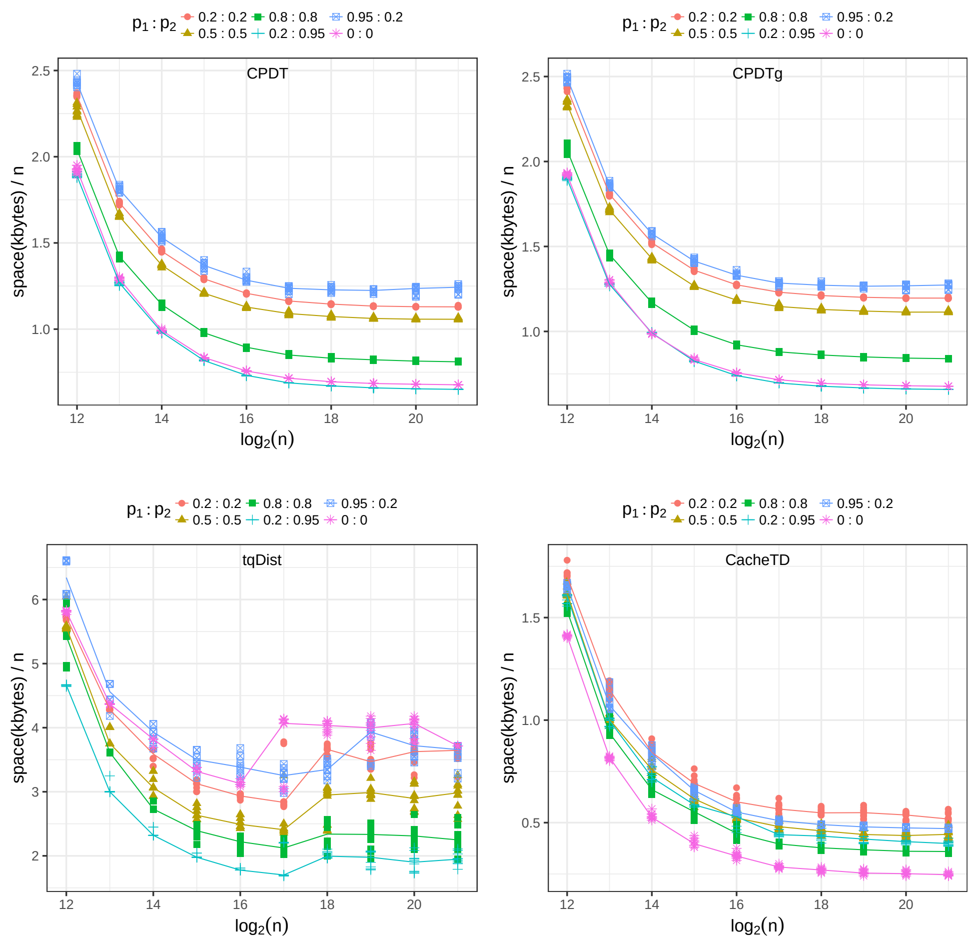


Fig. 22. Random model: How the contraction parameter affects space.

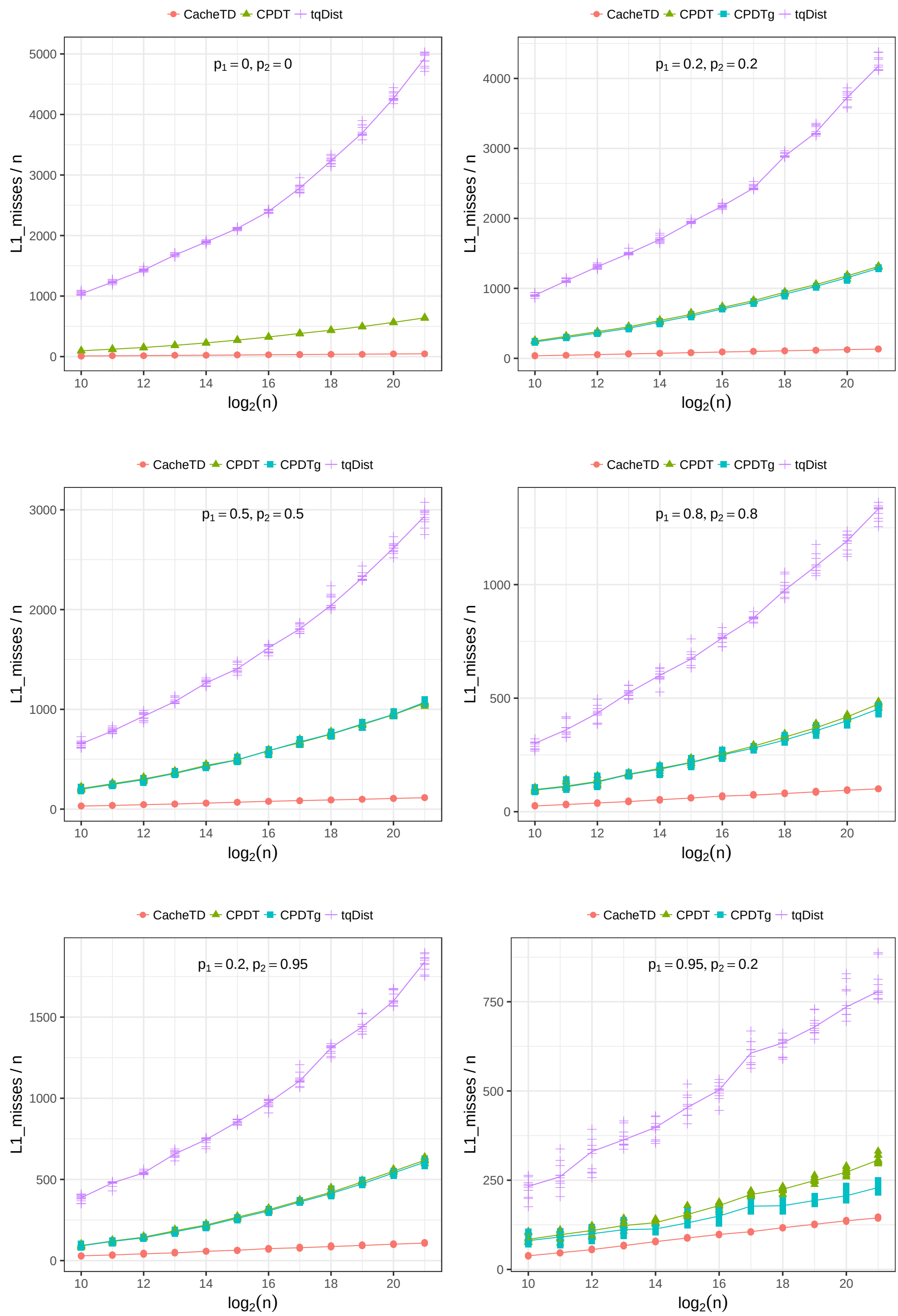


Fig. 23. Random model: L1 cache misses.

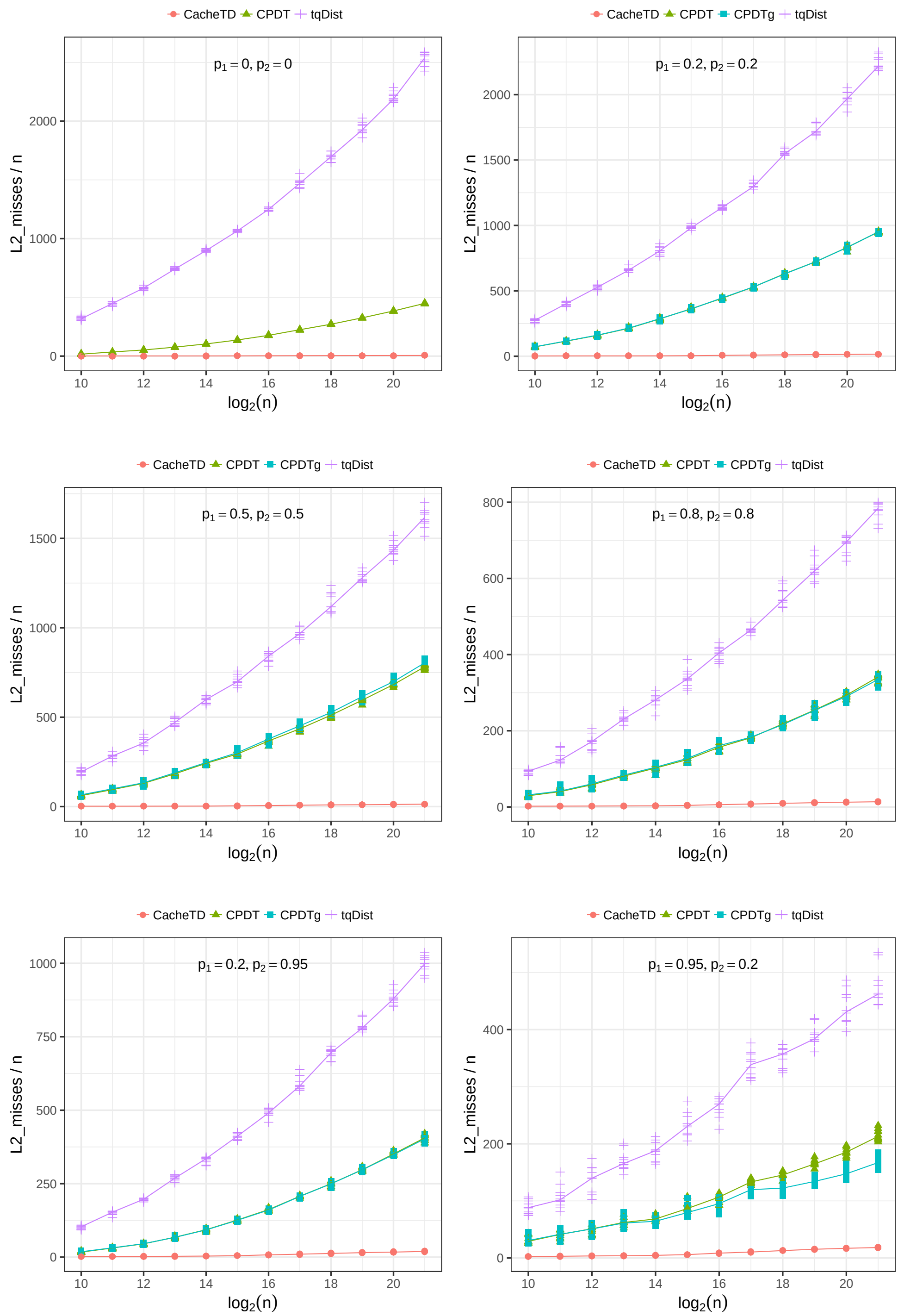


Fig. 24. Random model: L2 cache misses.

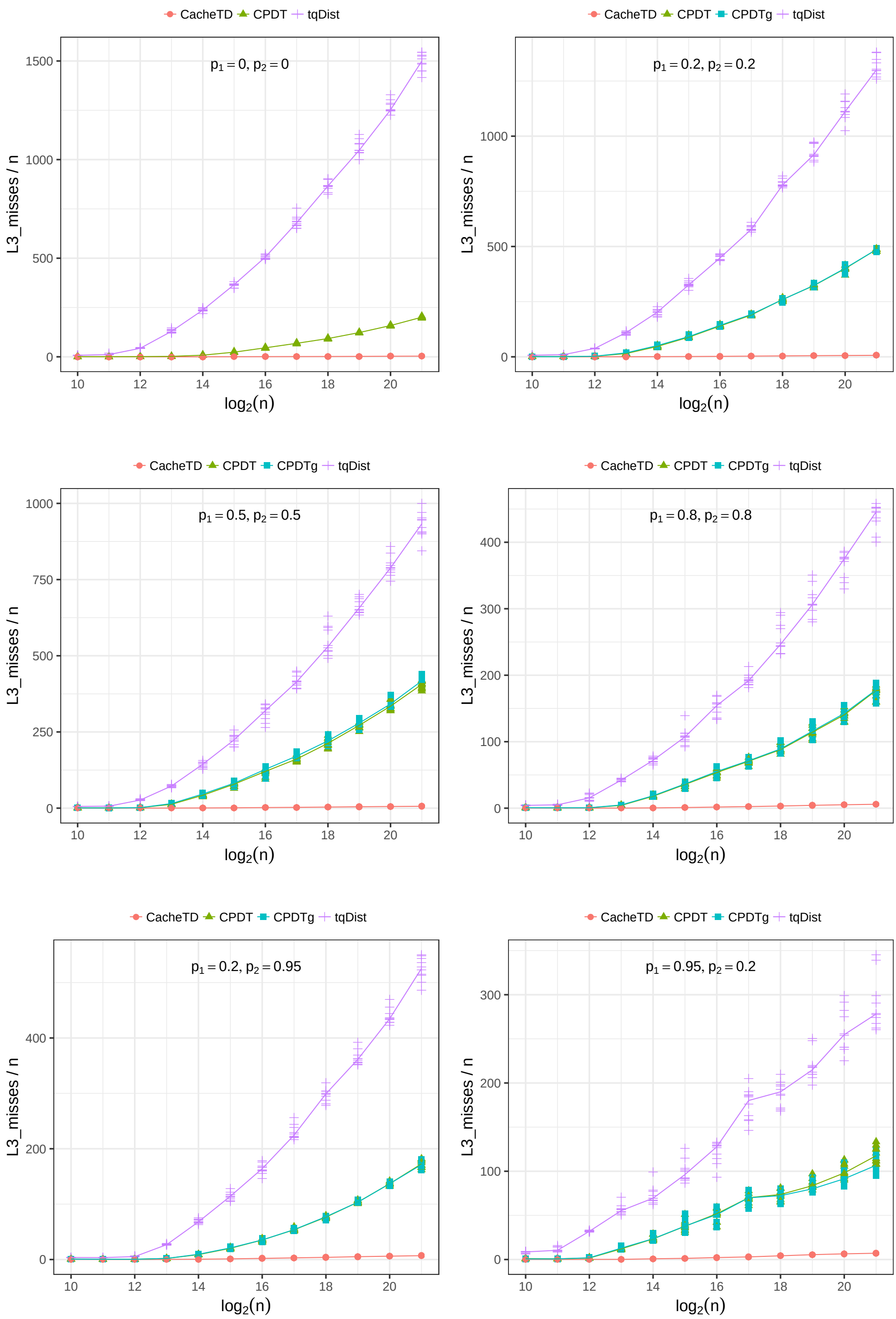


Fig. 25. Random model: L3 cache misses.

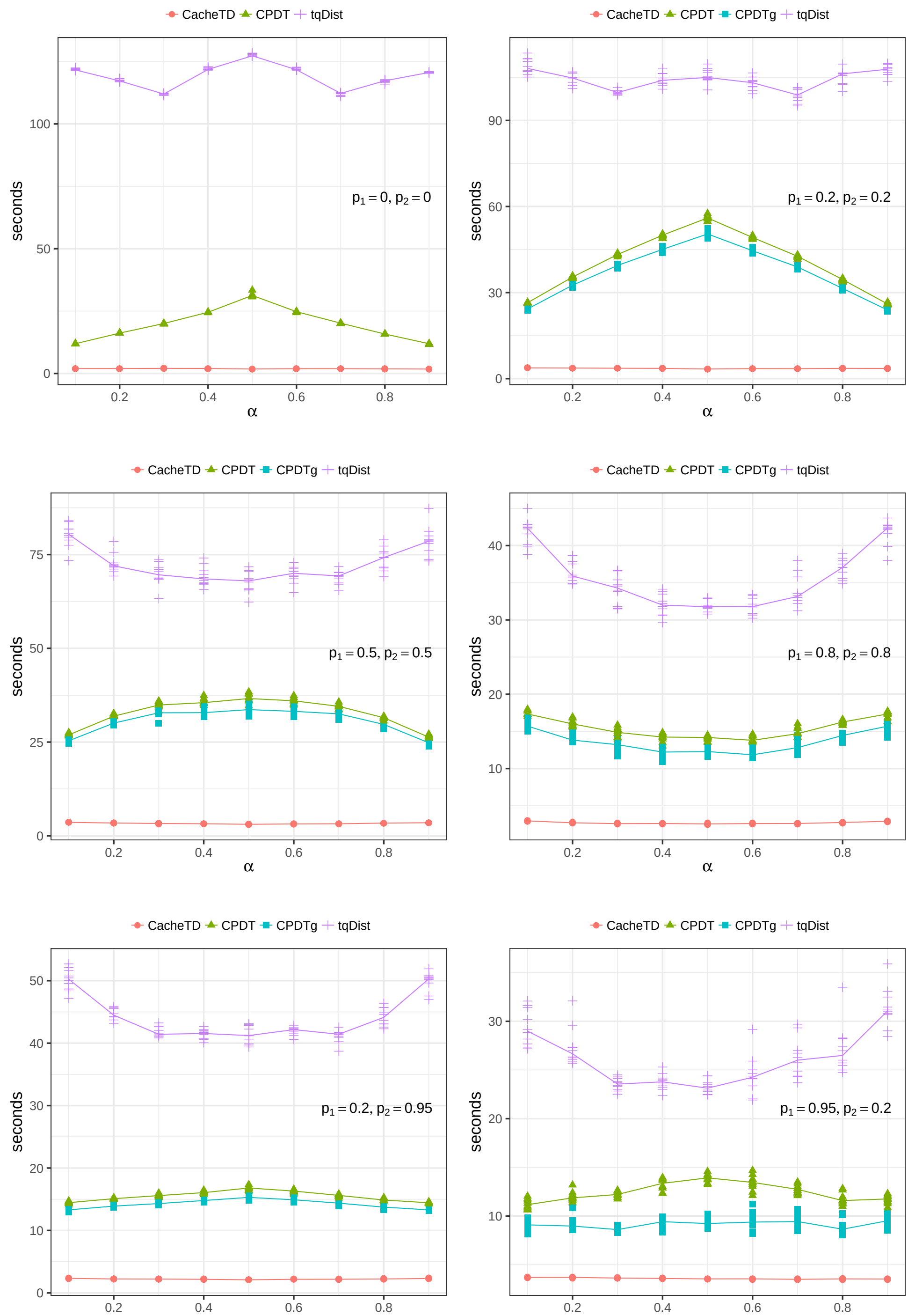


Fig. 26. Skewed model: Running time ($n = 2^{21}$).

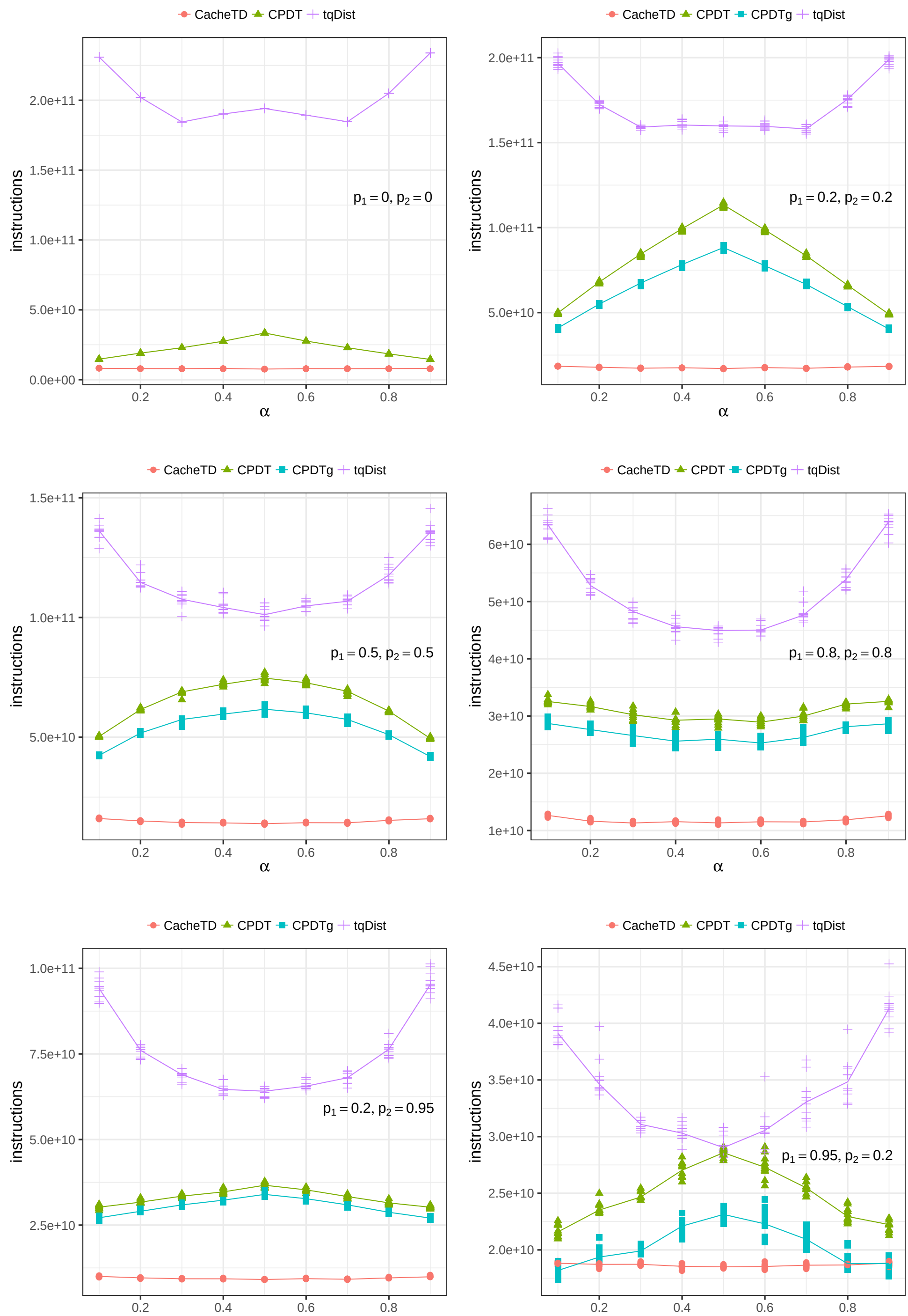


Fig. 27. Skewed model: Instructions ($n = 2^{21}$).

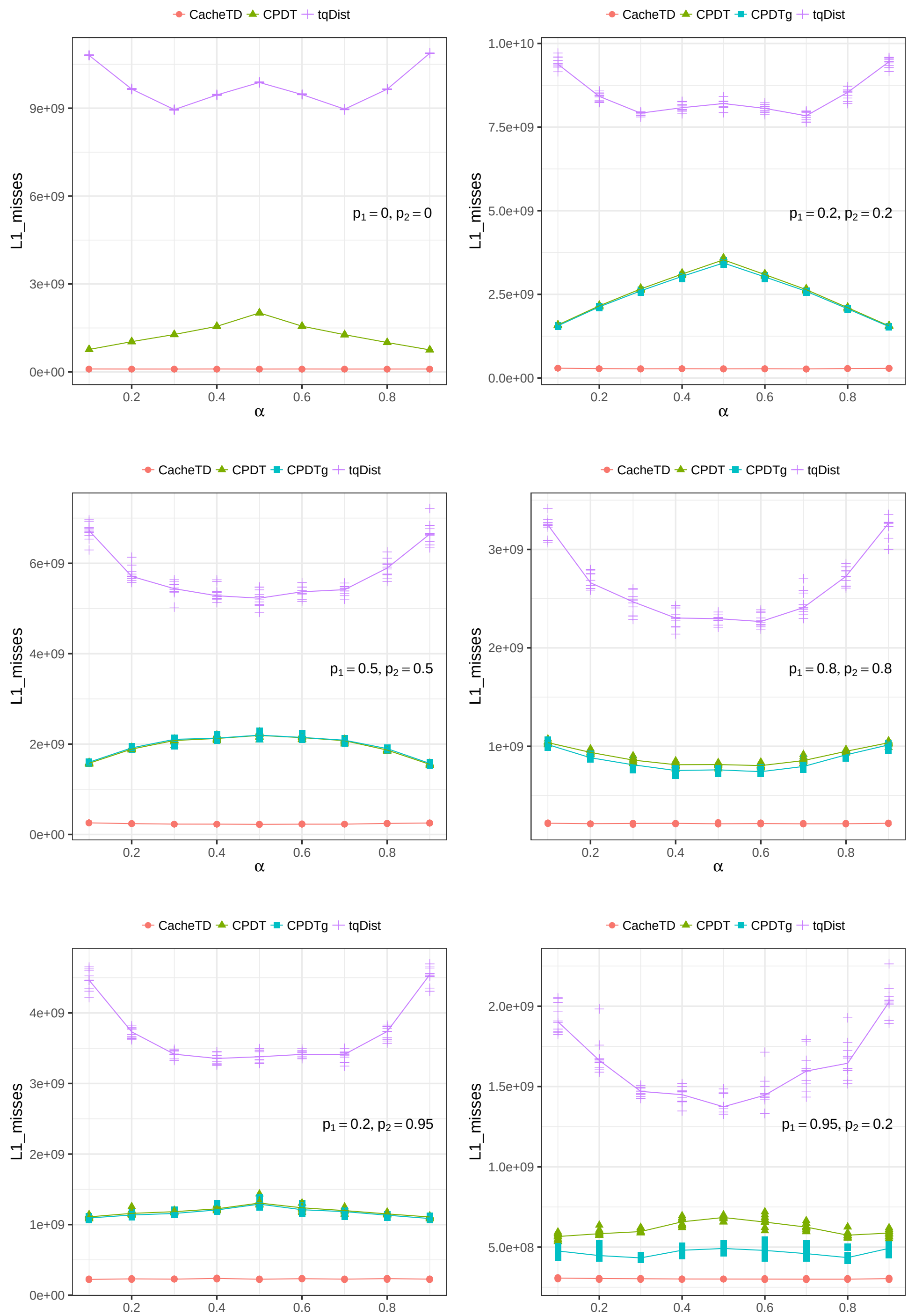


Fig. 28. Skewed model: L1 cache misses ($n = 2^{21}$).

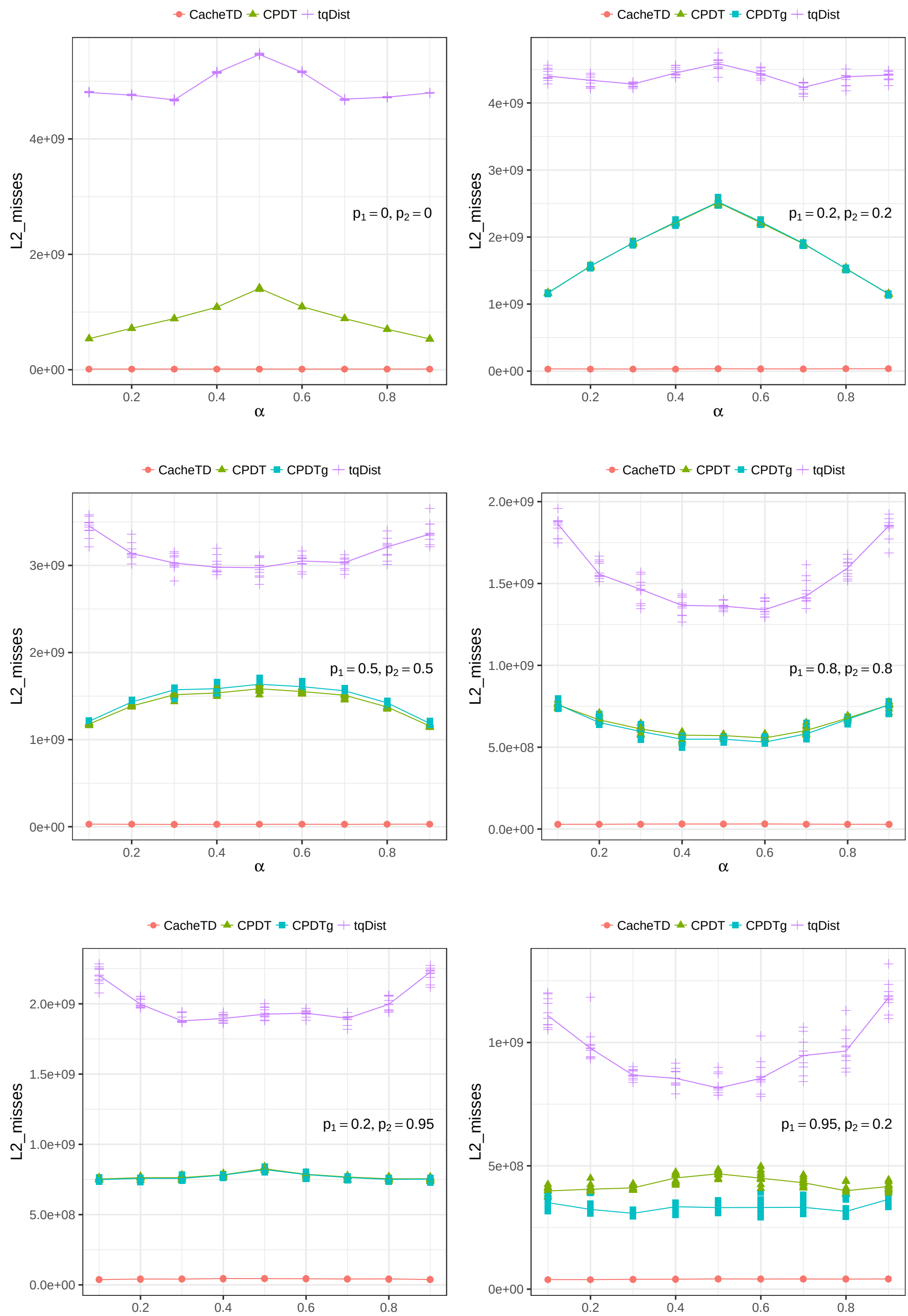


Fig. 29. Skewed model: L2 cache misses ($n = 2^{21}$).

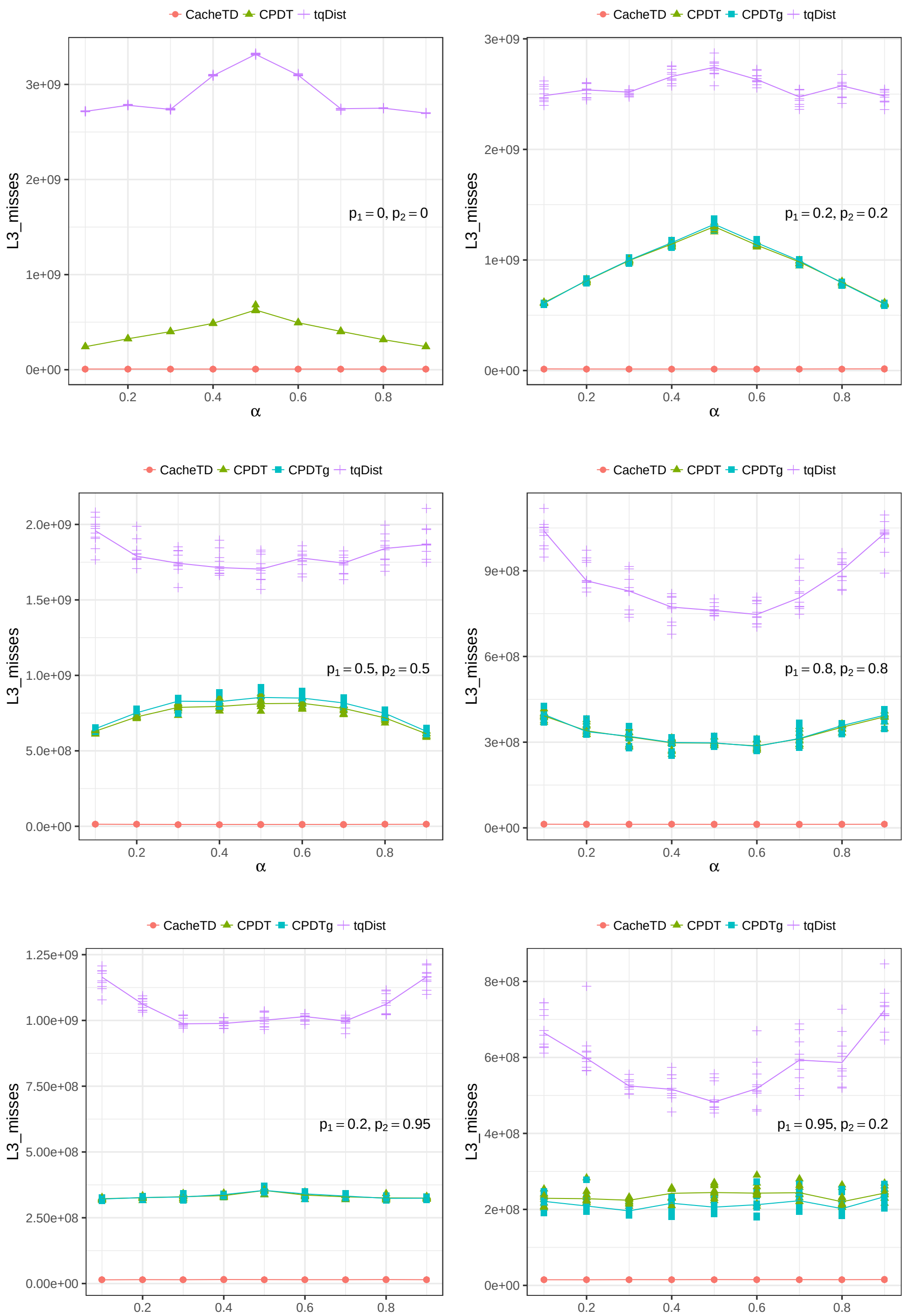


Fig. 30. Skewed model: L3 cache misses ($n = 2^{21}$).

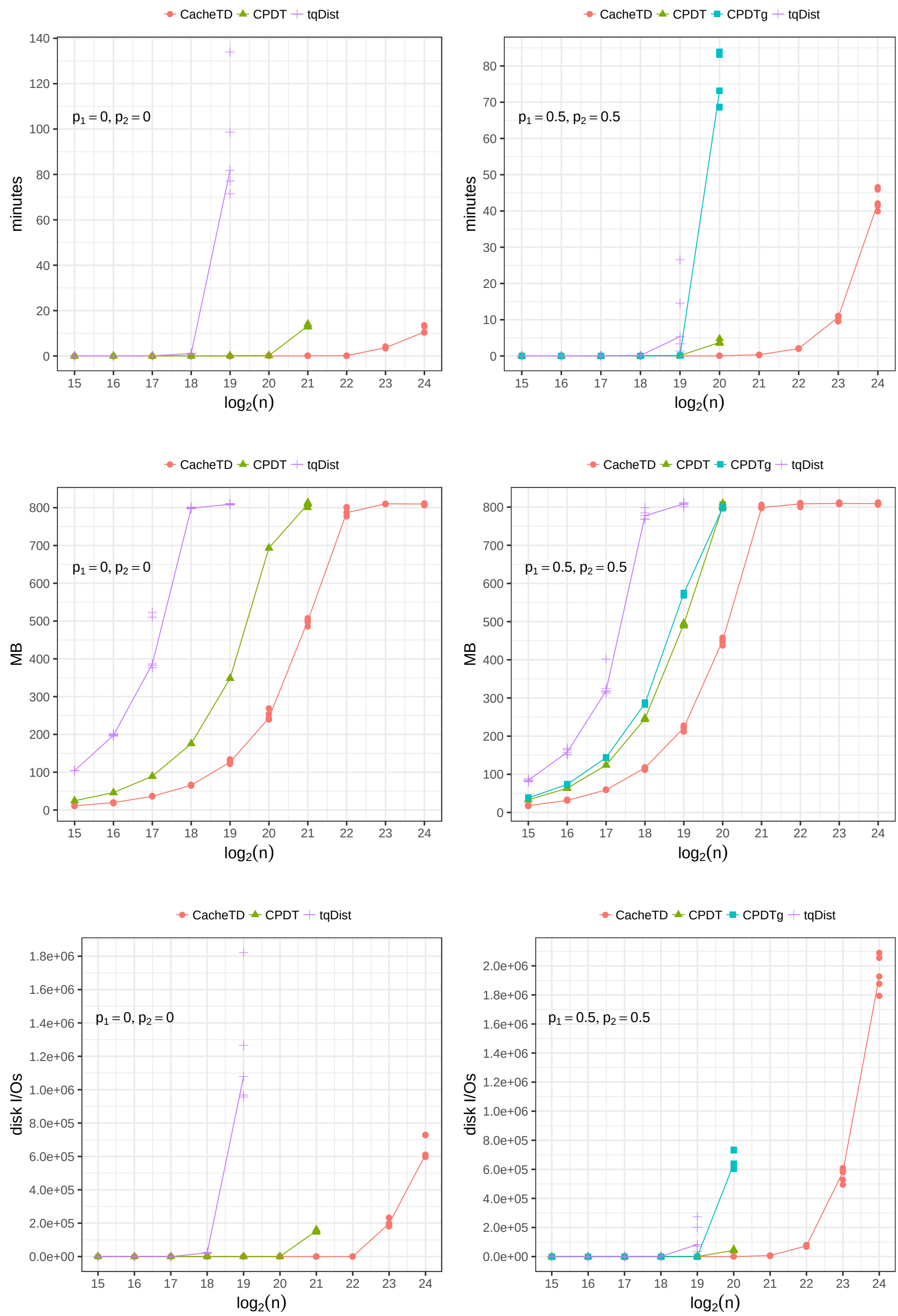


Fig. 31. Random model: I/O experiments.

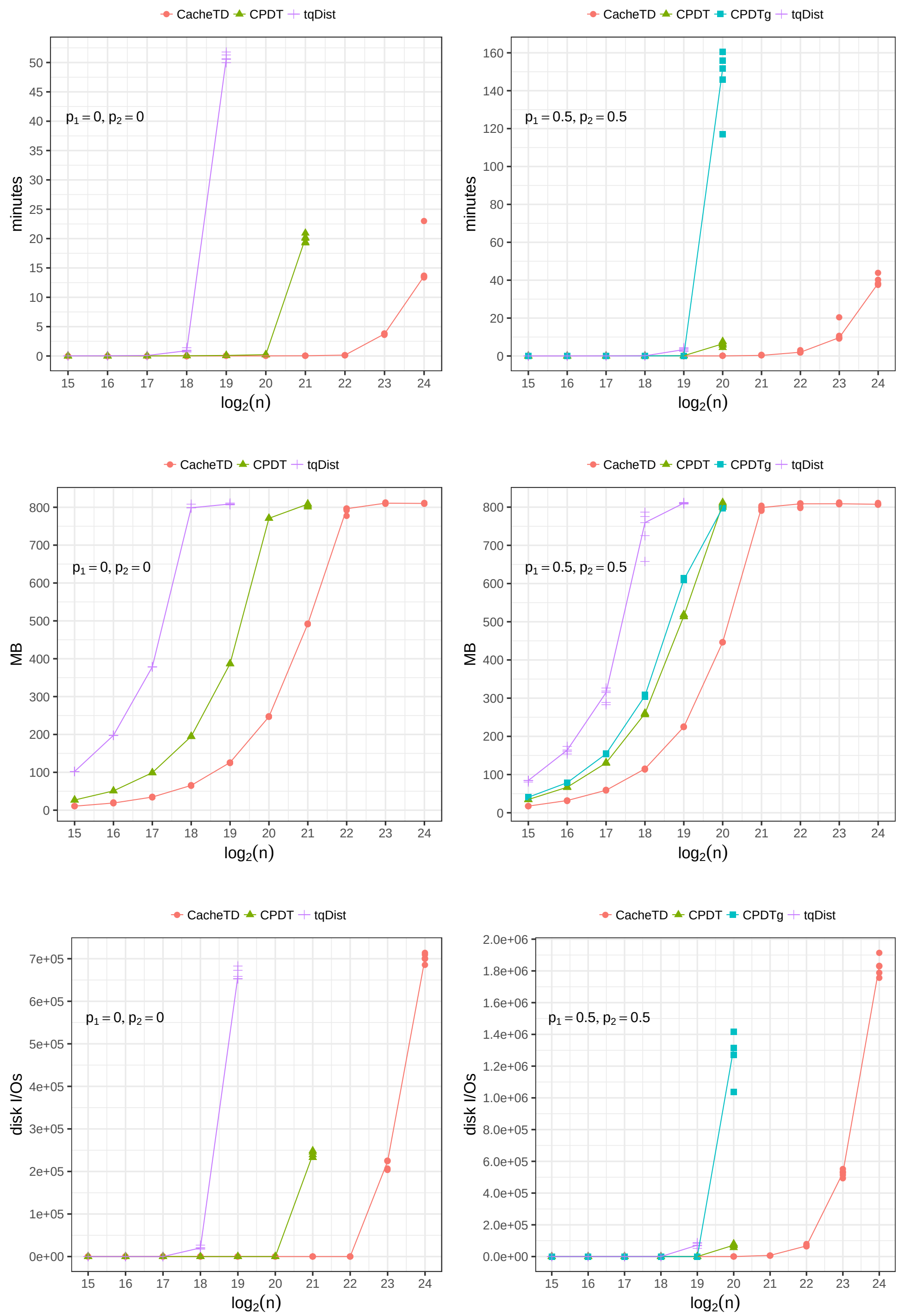


Fig. 32. Skewed model: I/O experiments with $\alpha = 0.5$.