On the Scalability of Computing Triplet and Quartet Distances*

Morten Kragelund Holt† Jens Johansen† Gerth Stølting Brodal†

Abstract

In this paper we present an experimental evaluation of the algorithms by Brodal *et al.* [SODA 2013] for computing the triplet and quartet distance measures between two leaf labelled rooted and unrooted trees of arbitrary degree, respectively. The algorithms count the number of rooted tree topologies over sets of three leaves (*triplets*) and unrooted tree topologies over four leaves (*quartets*), respectively, that have different topologies in the two trees.

The algorithms by Brodal *et al.* maintain a long sequence of variables (hundreds for quartets) for counting different cases to be considered by the algorithm, making it unclear if the algorithms would be of theoretical interest only. In our experimental evaluation of the algorithms the typical overhead per node is about 2 KB and 10 KB per node in the input trees for triplet and quartet computations, respectively. This allows us to compute the distance measures for trees with up to millions of nodes. The limiting factor is the amount of memory available. With 31 GB of memory all our input instances can be solved within a few minutes.

In the algorithm by Brodal *et al.* a few choices were made, where alternative solutions possibly could improve the algorithm, in particular for quartet distance computations. For quartet computations we expand the algorithm to also consider alternative computations, and make two observations: First we observe that the running time can be improved from $O(\max(d_1, d_2) \cdot n \cdot \lg n)$ to $O(\min(d_1, d_2) \cdot n \cdot \lg n)$, where n is the number of leaves in the two trees, and d_1 and d_2 are the maximum degrees of the nodes in the two trees, respectively. Secondly, by taking a different approach to counting the number of disagreeing quartets we can reduce the number of calculations needed to calculate the quartet distance, improving both the running time and the space requirement by our algorithm by a constant factor.

1 Introduction

Trees appear in many branches of science, for instance in biology, where so-called phylogenetic trees can be used to represent the evolutionary relationship between species. There are, however, different ways to construct such trees from the same data, or different datasets might be available. In both cases a number of different trees for the same set of species can be constructed, and it is useful to have a distance measure to compare how similar (or dissimilar) the constructed trees are [14]. A number of distance measures exist, among them the Robinson-Foulds distance measure and the triplet and quartet distance measures. The Robinson-Foulds distance measure enumerates the number of non-common splits in two unrooted input trees [12]. It can be computed in linear time, but is sensitive to outliers, i.e. changes to a few leaves might significantly influence the output of the algorithm, a drawback that the *triplet* and *quartet distances* do not suffer from [5].

The triplet distance [6], and the quartet distance, introduced in [7], enumerate all subsets of leaves of size three and four respectively, and count the number of different induced topologies. The triplet distance is applied to rooted trees whereas the quartet distance is used on unrooted trees. Naive algorithms, that in fact enumerate all of the $\binom{n}{3}$ or $\binom{n}{4}$ topologies, have running time at least $\Omega(n^3)$ and $\Omega(n^4)$ respectively, and are thus not practical for large inputs.

1.1 Previous work. In 1993 an algorithm calculating the quartet distance in time $O(n^3)$ was reported by Steel and Penny [14]. It is unclear whether or not the algorithm operates on trees of arbitrary degree.

An algorithm calculating the triplet distance for binary trees in time $O(n^2)$ was given by Critchlow *et al.* [6], and the quartet distance followed with Bryant *et al.* [5], where, although the article states that "The algorithm can be easily extended to handle partially-resolved trees", it only appears to work for binary trees. Brodal *et al.* [2] improved the quartet calculation runtime to $O(n \lg^2 n)$ for binary trees, and subsequently impro>ved this to $O(n \lg n)$ [3].

For trees of arbitrary degree Stissing *et al.* [15] gave an algorithm for calculating the quartet distance in time $O(d^9 n \lg n)$, where d is the maximum degree of any node in the two trees. An algorithm for arbitrary degree trees using matrix multiplication and with running time $O(n^{2.688})$ was given by Mailund *et al.* [11]. Bansal *et al.* [1] gave an algorithm for calculating the triplet distance of arbitrary degree trees in time $O(n^2)$.

Recently an $O(n \lg^2 n)$ algorithm calculating the triplet distance for binary trees was given by Sand *et al.* [13], and an algorithm calculating the quartet distance in time $O(dn \lg n)$, as well as the triplet distance of two trees of arbitrary degree in time $O(n \lg n)$ was given by Brodal *et al.* [4]. Tables 1 and 2 summarize the theoretical work on triplet and quartet distance compu-

*This paper summarizes the results obtained in the M.Sc. thesis by the first two authors [8].

†MADALGO (Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation), Department of Computer Science, Aarhus University, {thethawk,jensjoha}@gmail.com, gerth@cs.au.dk.

Year	Reference	Runtime	Arb.
	Naive	$O(n^4)$	✓
1996	[6]	$O(n^2)$	
2011	[1]	$O(n^2)$	✓
2013	[13]	$O(n \cdot \lg^2 n)$	
2013	[4]	$O(n \cdot \lg n)$	✓

Table 1: Triplet distance calculation algorithms.

Year	Reference	Runtime	Arb.
	Naive	$O(n^5)$	✓
1993	[14]	$O(n^3)$	
2000	[5]	$O(n^2)$	
2001	[2]	$O(n \cdot \lg^2 n)$	
2004	[3]	$O(n \cdot \lg n)$	
2007	[15]	$O(d^9 \cdot n \lg n)$	✓
2011	[11]	$O(n^{2.688})$	✓
2013	[4]	$O(\max(d_1, d_2) \cdot n \cdot \lg n)$	✓
2014	New	$O(\min(d_1, d_2) \cdot n \cdot \lg n)$	✓

Table 2: Quartet distance calculation algorithms.

tations. The last column in the tables indicates if the algorithms also work for trees of arbitrary degree. An experimental evaluation of the algorithms in [4] is the subject of this paper.

2 Results of triplet and quartet calculations

2.1 Existing implementations. The algorithm for calculating the quartet distance for binary trees in time $O(n \lg^2 n)$ from [2] has been implemented and documented to be useful in practice [10].[1] The algorithm calculating the quartet distance for trees of arbitrary degree in time $O(n^{2.688})$ has also been implemented [11].[2] The algorithm calculating the triplet distance for binary trees in time $O(n \lg^2 n)$ has been implemented and documented to be useful in practice [13]. While the source code is not generally available at the time of writing, a copy has been provided to us by the first author of [13]. In Sect. 5 we present experiments on these algorithms, comparing them to the algorithms described in this paper.

2.2 Our Results. We present an implementation, evaluation and improvements to the algorithms in [4]. The algorithm in [4] for computing the triplet distance between two trees of arbitrary degree uses time $O(n \cdot \lg n)$ and space $O(n \cdot \min(d_1, \lg n))$. Runtime and memory usage results for this algorithm are presented in Sect. 5.

The algorithm in [4] for computing the quartet distance between two trees of arbitrary degree uses time $O(\max(d_1, d_2) \cdot n \cdot \lg n)$ and space $O(\max(d_1, d_2) \cdot n \cdot \min(\max(d_1, d_2), \lg n))$. The algorithm in [4] maintains over a hundred variables per tree node for counting different topological subsets. To limit the complexity of the algorithm description in [4], a symmetry is exploited in the computation where the rôle of the two input trees is swapped. By extending the algorithm in [4] to maintain even more counters but circumventing the symmetric computations, we in Sect. 3 describe how to improve both the runtime and memory usage bounds for the quartet distance calculation to $O(\min(d_1, d_2) \cdot n \cdot \lg n)$ and $O(\min(d_1, d_2) \cdot n \cdot \min(d_1, d_2, \lg n))$, respectively. These changes from a maximum to a minimum can be significant for input consisting of one tree with large maximum degree and one tree with small maximum degree. By extending the algorithm with even more counters, we are able to compute the final quartet distance in several alternative ways. By an experimental comparison of the different approaches (Sect. 5), we arrive at an algorithm reducing both the asymptotic runtime and memory usage of the quartet distance calculation by a constant factor.

Using our implementation with the improvements introduced above, the quartet distance between two balanced binary trees with 1,000,000 leaves, can be calculated in approximately 1 minute and 45 seconds on the system presented in Sect. 5. The limiting factor in our computations is the amount of memory available. To our knowledge, the results presented are the current state of the art. The source code will be available for download embedded into a package with Python and R interfaces and available under the GNU LGPL license from the Bioinformatics Research Center, Aarhus Univeristy.[3]

3 Theory

In this paper we consider rooted and unrooted trees with n leaves and where the leaves are labeled uniquely. Given three leaves labeled a, b, c in a rooted tree, the subtree induced by a, b, c is denoted a *triplet*. Similarly four leaves labeled a, b, c, d in an unrooted tree induce a *quartet*. Both triplet and quartet topologies can be in one of two configurations: Resolved and unresolved, see Fig. 1a. Note that unresolved triplet configurations only occur in non-binary rooted trees and unresolved quartet configurations only occur in unrooted trees of

[1]Source code at `http://cs.au.dk/~mailund/qdist.html`
[2]Source code at `http://birc.au.dk/software/qdist/`

[3]`http://birc.au.dk/software/`

degree larger then three.

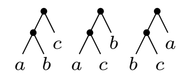
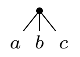
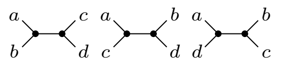
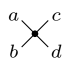
For a pair of input trees, T_1 and T_2, this results in four combinations as depicted in Fig. 1b. If the induced topology of a triplet or quartet is the same (different) in T_1 and T_2, it is denoted as *agreeing* (*disagreeing*). Unresolved-unresolved (E) topologies always agree, resolved-unresolved (C) and unresolved-resolved (D) topologies always disagree. Resolved-resolved topologies can either agree (A) or disagree (B). The triplet and quartet distance between two rooted and unrooted trees, respectively, counts the number of triplets and quartets having disagreeing topologies in the two trees, and the goal for any triplet and quartet distance algorithm is thus to find the value $B + C + D$ in Fig. 1b. Note that for triplets in binary rooted trees and quartets in unrooted trees of degree at most three, the values C, D, and E are always zero.

For a single rooted or unrooted tree the number of resolved and unresolved triplets or quartets can be computed in linear time by a simple dynamic programming, i.e. $A + B + C$, $A + B + D$, $D + E$ and $C + E$ can be computed in $O(n)$ time (see [4]). The total number of triplets or quartets $A + B + C + D + E$ in a tree is $\binom{n}{3}$ and $\binom{n}{4}$, respectively. From knowing $D + E$, $C + E$, and either $A + B$, C, D or E, we can deduce C, D, E and $A + B$ in constant time. From knowing $A + B$ and A we can obviously also deduce B by a single subtraction. It follows that from e.g. knowing *i)* A and B, or *ii)* A and E, the remaining values of $A, \dots, E$ can be deduced and the triplet or quartet distance be computed. In [4], *ii)* is used for computing triplet distances as $\binom{n}{3} - A - E$, and *i)* is used for computing quartet distances as $(A + B + D) + (A + B + C) - 2A - B$. In our implementation we consider quartet distances computed by both *i)* and *ii)*, and it turns out that *ii)* is faster and more space efficient than *i)*, see Sect. 5.

3.1 Recursive counting. The main idea of the algorithm presented in [4] (generalizing the approach of [3] from binary trees to arbitrary degree trees) is to consider both trees to be rooted trees and to construct a locally balanced version of T_2, a so-called hierarchical decomposition tree (HDT), link together the leaves of T_1 and the leaves of the HDT with bi-directional pointers and then color the leaves via a depth-first traversal of T_1. At each node in T_1, after all nodes have been colored accordingly, the contribution to a subset of $A, \dots, E$ is calculated by counting in the HDT. Before recursing on a subtree of size x, the relevant parts of the HDT are extracted and contracted so that they have size $O(x)$. As done in [4] the algorithm in its entirety runs in time $O(n \lg n)$ for triplets and $O(dn \lg n)$ for quartets, where d is the maximum degree of any node in the two input trees. In fact A can be computed in $O(n \lg n)$ time for quartets, but B requires $O(dn \lg n)$ time. The algorithms in [2] and [13] for unrooted trees of degree at most three and binary rooted trees, repectively, follow the same principle, but without contracting the HDT for a $\lg n$ factor time-penalty.

We have to refer the reader to [4] and [8] for the details of the counting. The details of all the variables maintained in the recursive counting can be found [8, Appendix A–C]—where you will find 12 pages of variable listings and equations for maintaining the variables at the nodes of the HDT for T_2 required for computing the contributions to A, B and E.

3.2 Improving the running time. When counting resolved-resolved quartet topologies in two rooted versions of T_1 and T_2, the quartets occur in three configurations named α, β and γ in [4] (see Fig. 2). This gives rise to 9 different combinations that must be handled when calculating A and B. The algorithm in [4] reduces this to six by handling only the cases marked in Fig. 2, swapping T_1 and T_2 and re-doing the calculation for the three missing symmetric values. One run of the partial quartet calculation has running time bounded by $O(dn \lg n)$ and space bounded by $O(dn \lg \min(d, n))$, where d is the largest degree of any node in T_1. Swapping means that both input trees will serve their turn as T_1 and thus d becomes the largest degree of any node in the two trees.

	Resolved	Unresolved
Triplet (rooted)		
Quartet (rooted)		

(a)

		T_2	
		Resolved	Unresolved
T_1	Resolved	A: Agree B: Disagree	C
	Unresolved	D	E

(b)

Figure 1: Configurations for triplets and quartets (a), and cases for topologies in two trees (b).

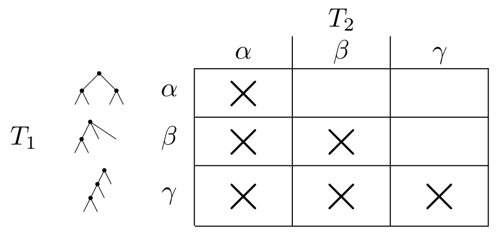


Figure 2: Counted resolved-resolved quartet configuration combinations for A and B in [4].

We improve this by handling the previously unhandled cases directly (details in [8, Appendix B]). By using the tree with the smallest maximum degree as T_1 we achieve running time $O(dn \lg n)$ and space usage $O(dn \lg \min(d, n))$, where $d = \min(d_1, d_2)$ and d_i is the maximum degree of any node in tree T_i. The optimization also removes (some of) the obvious overhead of running the algorithm twice, but increases the space required by our variation by a constant factor (see Sect. 5). To add support for the computation of E for quartets, we handle directly all the additional cases (details in [8, Appendix C]). All these extensions follow the counting framework introduced in [4].

It should be mentioned that all the additional variables make a substantial increase in the number of variables to be handled. In the original solution [4] 107 different types of topologies were counted. Our improvements added 53 additional types of topologies to be handled.

4 Implementation

We have implemented the triplet and quartet distance calculation algorithms as presented in [4] as well as our variations. The implementation was done in plain C++ with cross-platform compatibility in mind, although non-standard gcc features can be enabled at compile-time (see Sect. 4.4). The implementation has been tested on both Windows and Linux machines.

The purpose of our implementation was to address the following questions:

- The quartet distance calculation algorithm in [4] seems to have very large constants, e.g. we need to calculate up to $2d^2 + 79d + 22$ variables for each component in the HDT. For a binary input this is up to 188 variables per HDT component. This leads to the question of whether or not [4] has any practical value.

- The previous related implementations for binary trees [10, 13] use only one static HDT for T_2, i.e. the implementations do not try to contract the HDT of T_2 during the recursion. Theoretically this should lead to a $\lg n$ factor overhead in [10, 13] compared to [4] (and [3]). This leads to the question of whether or not the added work of contracting the HDT during recursion outweigh the saved $\lg n$ factor in practice.

- Our asymptotic improvement discussed in Sect. 3.2 adds up to $5d^2 + 18d + 7$ variables to each HDT component, for a total of $7d^2 + 97d + 29$ variables per HDT component. For a binary input this becomes up to 251 variables per HDT component. This leads to the question of whether or not using this variation compared to the algorithm presented in [4] has any practical advantages.

Before attempting to answer these questions we will go through some of the details of the implementation.

4.1 Representation of counters in the HDT.

To store the variables in each HDT node, a natural first step is to use a number of arrays, each of size d. This, however, is not a good idea. In the worst case the degree, d, can equal n. Thus, if T_1 is a single node with n leaves directly below the root, each of the $O(n)$ nodes in the HDT of T_2 will have arrays of size n yielding at least a quadratic space usage for the triplet distance calculation. In addition, the time-analysis of the algorithm only holds when the updating of nodes only handles the colors actually in use below it. When using arrays, this is not possible and what should have been constant time is now instead linear time in d. The solution we adopted was to use linked lists so that only the colors in use are actually handled. This results in a guarantee of the claimed runtime and space usage.

4.2 Extract & Contract.

One of the cornerstones of the algorithm is the two functions `extract` and `contract`. The process of extracting and contracting is three-step. The `extract` function creates a copy of the input HDT with non-marked sub-trees replaced with 0-leaves representing one or more leaves that have been cut off. The extracted copy of the HDT is converted into the tree it represents, and finally the `contract` function takes this input and returns the minimal tree with the same induced topologies for the given coloring. To reduce the overhead of multiple traversals of the HDT, we have combined the `extract` and `goBack` functions into a single function, which, given an HDT with marked leaves, outputs the extracted and converted HDT as a rooted tree. This effectively turns the three-step process into a two-step process.

In our implementation, extracting and contracting can be enabled or disabled at compile-time. If disabled, the asymptotic runtime will incur a $\lg n$ factor penalty. If enabled, the non-largest children are always extracted and contracted. The largest child, however, is only extracted and contracted when the size of this child is at most some fraction of the size of the current HDT. The fraction can be varied for different results. We have experimentally found the implementation to run faster, when the denominator of the fraction, herein named Q, is around 20,000. The value can be modified at compile-time.

4.2.1 Different Values for Q. We have tested the implementation on a randomly generated binary input consisting of two trees with 100,000 leaves each. The input was run with different values of Q on two different systems. System 1 is the same system as the one used in the experiments, and is outlined in Sec. 5.1. The runtimes for this system are depicted in Fig. 3a. System 2 is a Windows 7 system, with a quad-core 3.3 GHz 64-bit Intel Core i5 2500K processor and 16 GB of RAM. The runtimes for this system are depicted in Fig. 3b.

In Fig. 3a the runtimes are lower with contract enabled compared to having contract disabled, even with Q set to 10. This is not the case in Fig. 3b. As such, we note that the effects of different values of Q are somewhat machine dependent. However, a clear tendency is evident in both plots, and even though the system in Fig. 3b is generally slower than the system in Fig. 3a, the optimal value of Q seems to be approximately 20,000 in both cases.

As a result of the above, all experiments in Sec. 5 have been run with Q set to 20,000.

4.3 Optimizations. Since the initial implementation of the algorithm we have identified a number of optimizations. For the following, memory usage is based on polling 10 times per second. This implies that the reported value for the memory usage is subject to a degree of uncertainty, especially on small input. All measurements were taken on runs for the non-extended quartet distance calculation algorithm, as described in [4], with Q set to 10. This value was chosen arbitrarily at the time. Note that, as stated in Sec. 4.2, increasing Q to 20,000 decreases the runtime further.

The algorithm hints at creating contracted copies of the HDT rather early. To create a contracted copy, we convert the extracted HDT back to the tree it represents, and contract this tree before constructing a new HDT. This allows us to extract and contract early in the process, but postpone the construction of the updated HDT until it is needed. Since the HDT uses more memory than the tree it represents, this reduces the memory usage. We observed a reduction in memory usage of 25-50%, with approximately 50% as a relatively stable reduction on large input. As an added effect, the runtime was decreased by 4-10%, less on large input. This optimization is documented in Columns 2 and 3 in Tables 3 and 4.

Initially we used the standard C++ data structure `vector` to hold child pointers. As random-access is not needed, we could replace this by a purpose-built linked list. In doing so, we observed a 6-9% increase in the speed of the implementation when tested on binary trees. The memory usage also decreased slightly. This optimization is documented in Columns 3 and 4 in Tables 3 and 4.

Our final optimization was a more clever memory allocation. The basic idea was to allocate each datatype in a large pool and subsequently releasing memory back to this pool. This reduced the number of allocations

	Initial	Optimizations		
# Leaves	measure	1	1 & 2	1, 2 & 3
$1.0 \cdot 10^3$	17.29	10.49	7.39	11.09
$1.6 \cdot 10^3$	21.75	15.91	10.90	22.69
$2.5 \cdot 10^3$	41.69	24.27	19.31	31.80
$4.0 \cdot 10^3$	69.19	37.62	35.42	46.24
$6.3 \cdot 10^3$	107.20	58.79	56.40	67.57
$1.0 \cdot 10^4$	185.52	92.14	87.43	102.83
$1.6 \cdot 10^4$	292.91	144.37	132.21	158.46
$2.5 \cdot 10^4$	462.94	228.70	206.70	246.37
$4.0 \cdot 10^4$	735.15	362.10	354.03	390.33
$6.3 \cdot 10^4$	1,162.43	575.43	544.87	613.91

Table 3: Memory usage in MB with different levels of optimization.

	Initial	Optimizations		
# Leaves	measure	1	1 & 2	1, 2 & 3
$1.0 \cdot 10^3$	0.18	0.17	0.15	0.12
$1.6 \cdot 10^3$	0.33	0.30	0.28	0.21
$2.5 \cdot 10^3$	0.59	0.54	0.51	0.38
$4.0 \cdot 10^3$	1.06	0.98	0.93	0.70
$6.3 \cdot 10^3$	1.89	1.77	1.68	1.28
$1.0 \cdot 10^4$	3.35	3.14	2.98	2.31
$1.6 \cdot 10^4$	5.91	5.60	5.28	4.14
$2.5 \cdot 10^4$	10.33	9.84	9.27	7.36
$4.0 \cdot 10^4$	17.92	17.18	16.14	13.03
$6.3 \cdot 10^4$	31.15	29.83	27.89	22.87

Table 4: Runtime in seconds with different levels of optimization.

needed, giving an 18-25% increase in the speed of the program. The memory usage, however, increased with 10-20% on large input, and by more than 100% on small input. We expect at least some of this to be due to the reported memory usage being based on polling. When releasing memory back to the operating system, which was the case before this optimization, the polling might by chance measure between peaks. This is in contrast to after the optimization, where releasing memory merely releases it back to the pool. As such, the memory is still allocated by the program, and polling is thus more likely to measure the peak-usage. This optimization is documented in Columns 4 and 5 in Tables 3 and 4.

In total, on inputs larger than 10,000 leaves, these optimizations increased the speed by approximately 25% and decreased the memory usage by approximately 45%. The raw data is available in Tables 3 and 4.

4.4 Limitations. During the development, we have identified a number of limitations. These are discussed below.

4.4.1 Integer representation. As part of the algorithm, $\binom{n}{3}$ is calculated for triplets and $\binom{n}{4}$ for quartets. These numbers increase rapidly and representing them is therefore a problem. We generally use 64-bit signed integers and will therefore run into overflows at $n \approx 2,000,000$ for triplets and $n \approx 55,000$ for quartets. For this reason, we have made it a compile-time option to use the non-standard type `__int128` available in `gcc` for variables that can potentially contain $O(n^4)$. 128-bit integers are used in the experiments below.

4.4.2 Recursion depth. The underlying operating system imposes a limit on the number of recursions a program can perform. Since the implementation is written in a recursive manner it will not work for very high trees. On inputs which include a tree consisting of a very long chain, the program fails when $n \approx 4,000$ on Windows and $n \approx 48,000$ on Linux.

5 Experiments

Using our implementation we have performed a number of experiments presented below.

5.1 Setup. The experiments have been performed on a computer running Ubuntu Linux Server 12.04, with a quad-core 3.4 GHz 64-bit Intel Core i7-3770 processor and 31.2 GB of RAM.

Runtime-values are averages of wall-time runtimes across three runs, measured externally. This results in a slight startup overhead, but gives a better indication of the real-world runtime of the algorithm. Memory usage is based on polling. The reported number is the peak value. In addition to actual runtime-values we have instrumented the code with a global counter of recursive calls and loop rounds. This provides us with a stable look at the work done by the implementation, unaffected by other processes running on the test-system. We, however, do not try to weigh some operations more than others, and doing constant work is thus recorded as such.

We have compared our implementation to a number of previous implementations for both triplet and quartet calculations.

5.2 Test input. Since we are primarily interested in the scalability of our implementations we have only performed tests on randomly generated data. In the following section we utilize random, fully balanced trees, i.e. all leaves are at the same level in the tree (or as close to this as the number of leaves allow) and all leaf-labels have been randomly permuted. Two random trees will likely have a very large distance.

5.3 Results. In this section we have run our implementation of the quartet distance calculation (under different configurations) and our implementation of the triplet distance calculation on a number of trees. The implementations running in time $O(n \lg^2 n)$ [10, 13] and $O(n^{2.688})$ [11] have also been run on the input, although these have not been tested for inputs larger than $n =$ 10,000.

5.3.1 Quartet Distance. From Fig. 5a we observe that all three configurations of our implementation can easily compute the difference between two binary trees of size up to one million leaves. The implementation of [4] without additions is slowest at a still respectable $\approx$ 213 seconds. Faster then is the implementation calculating the three missing symmetric values directly and fastest where we instead of calculating A and B calculate A and E ($\approx$ 140 seconds and $\approx$ 105 seconds, respectively).

From Fig. 5b we observe, however, that calculating the missing symmetric values directly, not surprisingly, requires more memory, because of additional counters and sums. Again, however, calculating A and E instead is better, using the least memory. For one million leaves, the variations use $\approx$ 11.3 GB, $\approx$ 9.3 GB and $\approx$ 8.5 GB, respectively. While this is a lot of memory, it is not more than some desktops or even laptops would be able to handle, even at 1,000,000 leaves.

Comparing our fastest variation to previous algorithms, our implementation is a clear winner, spending less than half a second on 10,000 leaves, whereas

[10] spends more than 29 seconds, and [11] spends more than 62 seconds, meaning that it is more than 140 times slower than our result. In all observed cases our variation, calculating A and E is faster in practice than anything else.

Going from binary to non-binary trees, as seen in Fig. 6a-c, the story is more or less unchanged. Calculating A and E is still fastest overall, and can calculate the quartet distance between two trees with one million leaves, at least with $d = 256$ as the case in Fig. 6a-b. Our variation, calculating A and B and the symmetric values directly, however, use too much memory at one million leaves to be able to run on our test system. The plot for this run thus tops out at $10^{5.8}$ leaves.

With $d = 1024$ [11] is actually faster than our implementation on small instances, but once the instance gets large enough, the time spend increases rapidly, and our implementation is more than twice as fast a the 10,000 leaves mark. Additionally, our implementation uses significantly less memory, even at large instances.

In all observed cases our variation is faster in practice than the algorithm in [4]. With contract enabled the quartet distance for the binary balanced trees with 1,000,000 leaves can be calculated in less than two and a half minutes using our variation.

The importance of contraction is studied in Fig. 4a and 4b. In Fig. 4a we observe that with contract enabled the actual runtime appears to be $O(n \cdot \lg^2 n)$, an additional $\lg n$ factor compared to the theoretical timebound (d is the constant 2 here). The counter value, as can be seen from Fig. 4b, however, appears to be $O(n \cdot \lg n)$, and thus agrees with the theory. In both cases disabling contract results in an additional $\lg n$ factor as the theory predicts. An explanation for the additional $\lg n$ factor in the actual runtime could be "the cost of address translation" [9], but as the focus of this paper has been on scalability, this has not been explored in depth.

In Fig. 8 two other types of input are studied. *Leaf moved* describes the situation where T_1 is a random tree and T_2 is merely T_1 where the leaves labeled 1 and 2 have switched places. This results in a relatively small distance. The $x\%$ *left-biased* trees are trees, where a node with n leaves below it has $x\%$ of these leaves in the first child, and the rest evenly distributed among the rest of the children. From Figs. 8a and c we observe that leaf moved input is processed much faster than two random trees. We believe this is because the leaves being recolored are close to each other in the HDT. This will result in a lower number of internal nodes that require updating than if both input trees were random. From the same figures we observe that 75% left-biased trees are actually processed slightly faster than completely balanced trees. We believe this is because of the number of times a leaf is recolored during the course of the algorithm. As only the non-largest children are recolored, leaves are recolored more in balanced trees than in unbalanced trees. From Figs. 8b and d we observe that with T_1 being fully balanced, the type of T_2 does not seem to have a significant influence on the runtime.

5.3.2 Triplet Distance.

In Fig. 7a-b, our implementation, running in time $O(n \lg n)$ is plotted against [13] running in time $O(n \lg^2 n)$. Despite the theoretical advantage, our implementation is both slower and uses significantly more memory. Moving away from binary trees (which is the only trees [13] supports), the runtime of our implementation gets faster, most likely because of fewer internal nodes that needs updating.

When comparing the time and memory usage of the triplet distance calculation to the memory usage of the quartet distance calculation, however, the triplet distance calculation is seen to be faster and less memory-heavy. Our triplet distance calculation can calculate the triplet distance for 1,000,000 leaves in less than a minute. While this is slower than [13] we believe that this it is still very much acceptable.

6 Conclusion and future work

We have improved upon the algorithm in [4]. The first variation improves the runtime, although at the cost of a constant factor memory-increase. The second variation both improves the runtime and decreases the memory usage. We have found the algorithm, in all variations, to be useful in practice.

For practical purposes the amount of memory is likely going to be a limiting factor before time-usage is. With 1,000,000 nodes the memory consumption of our algorithm, calculating A and B, is approximately 11.5 GB. Calculating A and E instead reduces this number to approximately 8.5 GB.

Using our first variation, we are able to calculate the quartet distance for 1,000,000 leaves in less than two and a half minutes. The second variation reduces this to under two minutes. The triplet calculation is faster than this, although an implementation for binary trees which is even faster in practice does exist [13].

6.1 Future work.

As the memory consumption appears to be the first limiting factor, an obvious question is if the amount of memory used can be decreased. For instance, would it be possible to decrease the number of counters further than we did by calculating E instead of B? Or could the number of nodes in the generated

HDT somehow be decreased? Alternatively, if a precise result is not strictly necessary, could 32-bit floats be used to decrease the memory usage while still being correct within a small margin of error of a few percent?

Another question is whether or not the asymptotic run time can be reduced. Can triplets or quartets (binary or of arbitrary degree) for instance be solved in linear time instead? Can the d factor be removed?

Since most processors today have multiple cores, it would be interesting to see, if the algorithm can be multithreaded. A possible approach for this, might be to split T_1 in two, extracting and contracting the relevant parts of T_2, and delegating each subproblem to different threads. If this approach is feasible, it could perhaps be extended to more cores by, for example, repeating the process on each subproblem. Furthermore, if this technique is feasible, it might make it possible to split the problem into smaller chunks, solving one at a time for a smaller memory footprint.

Going in an entirely different direction, can the distances be approximated faster instead? Would it for instance be possible to create an approximation algorithm that produce a correct result within a margin of error of a few percent, while running in linear or sublinear time?

References

[1] M. S. Bansal, J. Dong, and D. Fernández-Baca. Comparing and aggregating partially resolved trees. *Theoretical Computer Science*, 412(48):6634–6652, 2011.

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

[3] G. S. Brodal, R. Fagerberg, and C. N. S. Pedersen. Computing the quartet distance between evolutionary trees in time $O(n \log n)$. *Algorithmica, Special issue on ISAAC 2001*, 38(2):377–395, 2004.

[4] G. S. Brodal, R. Fagerberg, C. N. S. Pedersen, T. Mailund, and A. Sand. Efficient algorithms for computing the triplet and quartet distance between trees of arbitrary degree. In *Proc. 24th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1814–1832, 2013.

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

[6] D. E. Critchlow, D. K. Pearl, and C. L. Qian. The triples distance for rooted bifurcating phylogenetic trees. *Systematic Biology*, 45(3):323–334, 1996.

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

[8] M. K. Holt and J. Johansen. Computing triplet and quartet distances. Master's thesis, Department of Computer Science, Aarhus University, June 2013. www.cs.au.dk/~gerth/advising/thesis/jens-johansen-morten-holt.pdf.

[9] T. Jurkiewicz and K. Mehlhorn. The cost of address translation. In *Proc. 15th Meeting on Algorithm Engineering and Experiments*, pages 148–162. SIAM, 2013.

[10] T. Mailund and C. N. S. Pedersen. QDist–quartet distance between evolutionary trees. *Bioinformatics*, 20(10):1636–1637, 2004.

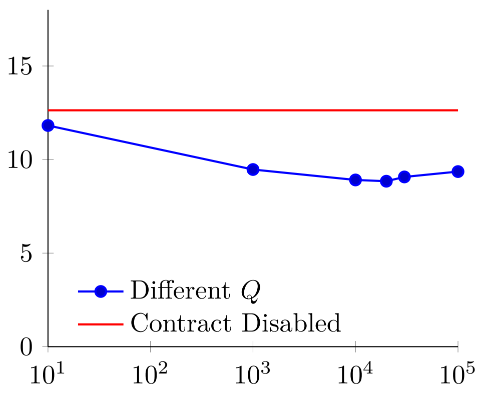
[11] J. Nielsen, A. Kristensen, T. Mailund, and C. N. S. Pedersen. A sub-cubic time algorithm for computing the quartet distance between two general trees. *Algorithms for Molecular Biology*, 6(1):15, 2011.

[12] D. F. Robinson and L. R. Foulds. Comparison of phylogenetic trees. *Mathematical Biosciences*, 53:131–147, 1981.

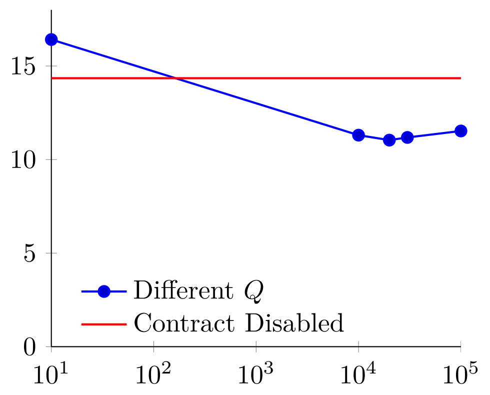
[13] A. Sand, G. S. Brodal, R. Fagerberg, C. N. S. Pedersen, and T. Mailund. A practical $O(n \log^2 n)$ time algorithm for computing the triplet distance on binary trees. *BMC Bioinformatics*, 14(Suppl 2):S18, 2013.

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

[15] M. S. Stissing, C. N. S. Pedersen, T. Mailund, G. S. Brodal, and R. Fagerberg. Computing the quartet distance between evolutionary trees of bounded degree. In *Proc. 5th Asia-Pacific Bioinformatics Conference*, pages 101–110. Imperial College Press, 2007.

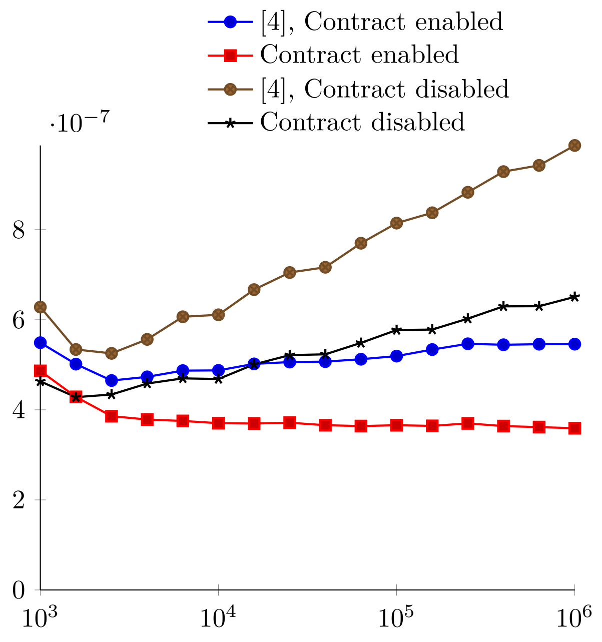


(a) Run on same, Linux, machine as the experiments in Sect. 5.

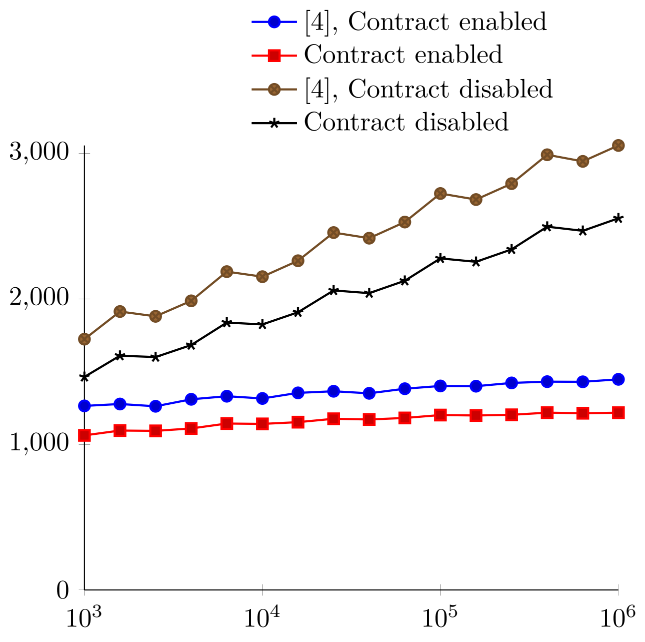


(b) Run on a different, Windows, machine than the one used in Sect. 5.

Figure 3: Runtimes with different values of Q as discussed in Sect. 4.2.1. Input trees had 100,000 leaves. Trees are the same for both runs. The x and y axis are the values of Q and runtime in seconds respectively.

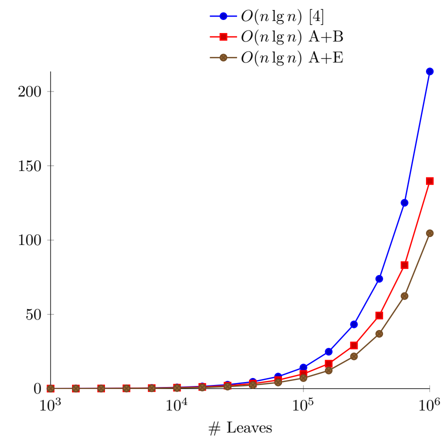


(a) Runtimes divided by $n \cdot \lg^2 n$.

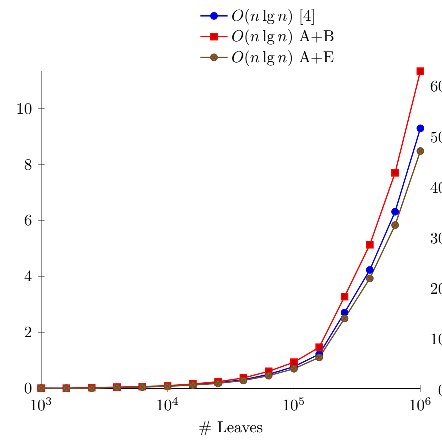


(b) Counter divided by $n \cdot \lg n$.

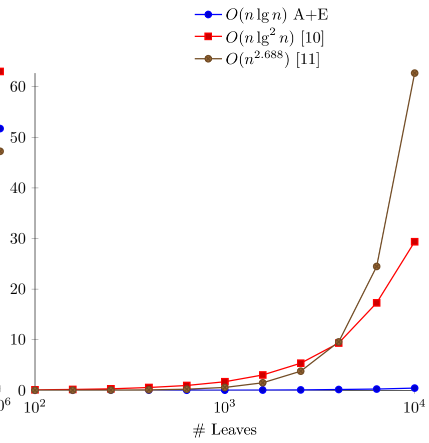
Figure 4: Quartet distance calculation. Random balanced binary tree against random balanced binary tree. The x axis denotes number of leaves.



(a) Runtimes in seconds.

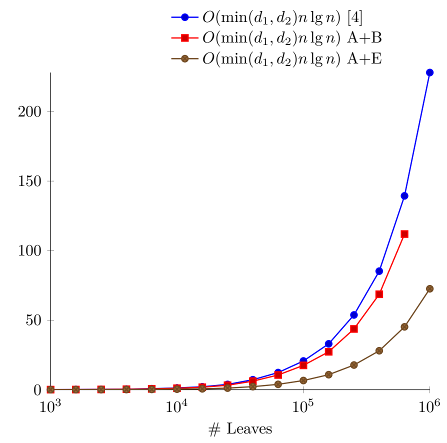


(b) Memory usage in GB.

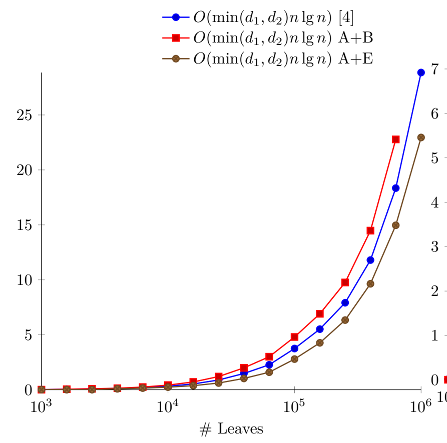


(c) Runtimes in seconds.

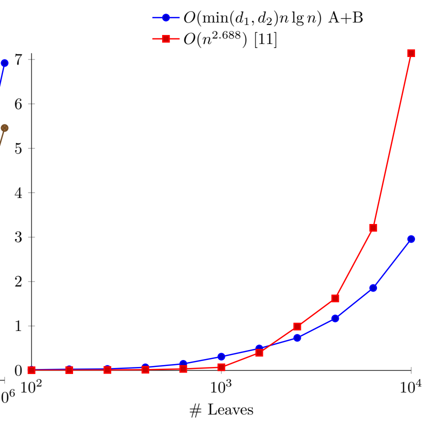
Figure 5: Comparison of algorithms for computing the quartet distance on binary trees.



(a) Runtimes in seconds, $d = 256$.

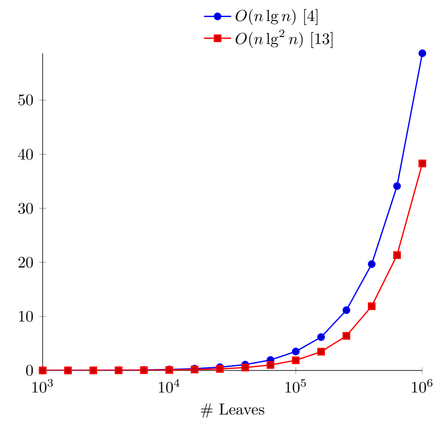


(b) Memory usage in GB, $d = 256$.

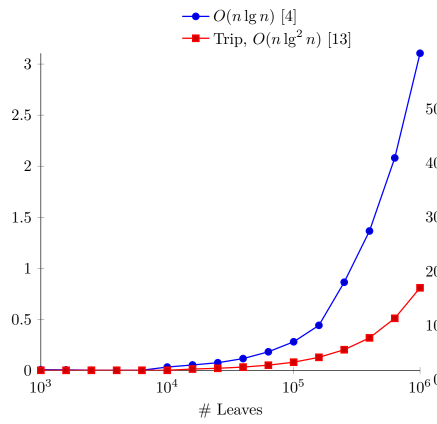


(c) Runtimes in seconds, $d = 1024$.

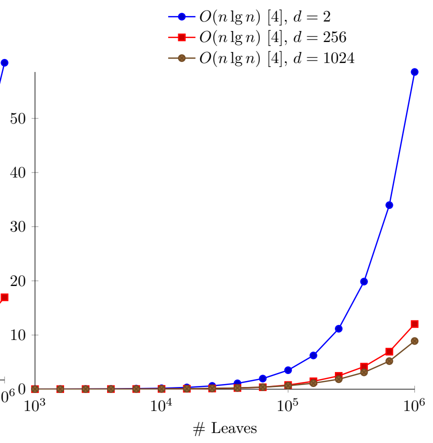
Figure 6: Comparison of algorithms for computing the quartet distance on trees of degree larger than three.



(a) Runtimes in seconds.

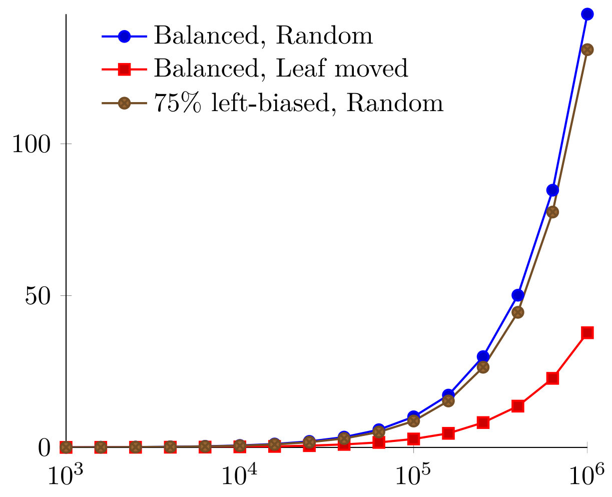


(b) Memory usage in GB.

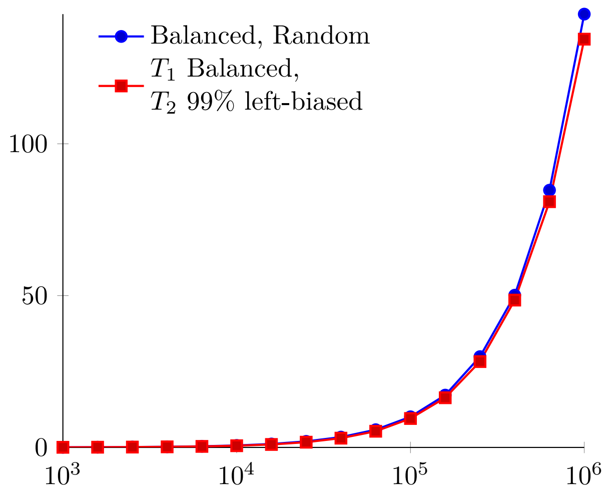


(c) Runtimes in seconds, different d.

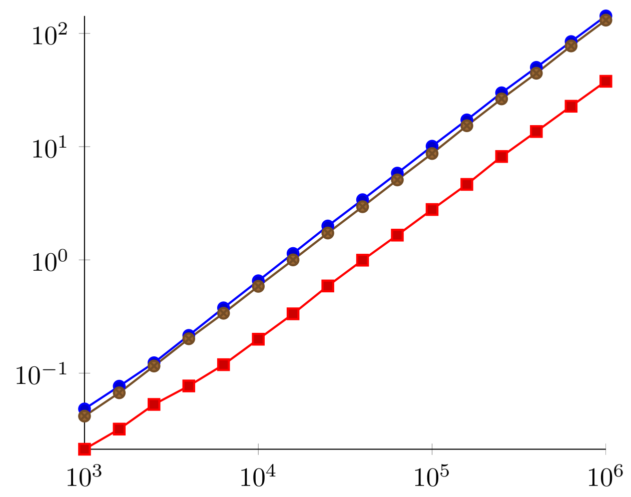
Figure 7: Comparison of algorithms for computing the triplet distance on binary, and high degree trees.



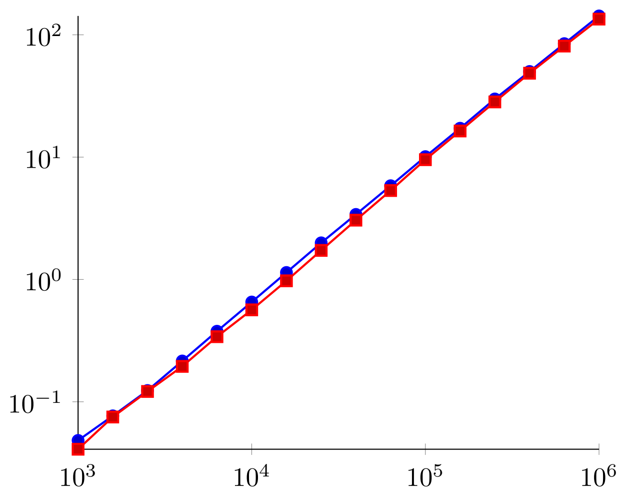
(a)



(b)



(c)



(d)

Figure 8: Quartet distance calculation. All trees are binary, the algorithm is our variation with contract enabled. The x and y axis are number of leaves and runtime in seconds respectively. Note the (a) and (c) depict the same data, although (c) uses a logarithmic y-axis. As such the legend from (a) applies to (c) as well. The same is the case for (b) and (d).