Computing Triplet and Quartet Distances

Jens Johansen, 20082212
Morten Kragelund Holt, 20081607

Master's Thesis, Computer Science

June 2013

Advisor: Gerth Stølting Brodal

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

In this thesis we present an implementation of, improvements of and experiments on the two algorithms presented by Brodal *et al.* [1].

The algorithms calculate the triplet and quartet distances on two respectively rooted and unrooted trees, of arbitrary degree. Both the triplet and the quartet distances are measures for comparing the similarity between two trees.

The triplet distance calculation operates on sets of three leaves (*triplets*). The quartet distance calculation operates on sets of four leaves (*quartets*). The distances are defined as the number of triplets or quartets with different structures in the two trees.

The triplet distance calculation algorithm presented in [1] runs in time $O(n \cdot \lg n)$. The space usage of the algorithm is $O(n \cdot \min(d_1, \lg n))$, where d_1 is the degree of the first input tree, T_1.

The quartet distance calculation algorithm runs in time $O(\max(d_1, d_2) \cdot n \cdot \lg n)$ where d_2 is the degree of the second input tree, T_2. The space usage of the algorithm is $O(\max(d_1, d_2) \cdot n \cdot \min(\max(d_1, d_2), \lg n))$.

We improve the quartet distance calculation algorithm, reducing the runtime to $O(\min(d_1, d_2) \cdot n \cdot \lg n)$. This improvement furthermore decreases the space usage to $O(\min(d_1, d_2) \cdot n \cdot \min(d_1, d_2, \lg n))$.

We furthermore significantly reduce the number of calculations needed to calculate the quartet distance.

Through experiments we provide empirical evidence that both the algorithms presented in [1], as well as our improvements, are feasible and perform well in practice.

Work Method

The thesis, as well as the implementation it describes, is the product of an equal share collaboration between the two authors.

The implementation was written via pair programming, and the text of the thesis was written via “pair writing”. Virtually no sentence in this thesis is the product of one author alone.

Resumé

I dette speciale præsenterer vi en implementering af, forbedringer af og eksperimenter på de to algoritmer præsenteret af Brodal *et al.* [1].

Algoritmerne beregner triplet- og kvartetdistancerne på to henholdsvis rodede og urodede træer af vilkårlig grad. Både triplet- og kvartetdistancerne er mål for hvor ens to træer er.

Tripletdistanceberegningen arbejder på mængder af tre blade (*tripletter*). Kvartetdistanceberegningen arbejder på mængder af fire blade (*kvartetter*). Distancen er defineret som antallet af tripletter, eller kvartetter, med forskellig struktur i de to træer.

Algoritmen til tripletdistanceberening præsenteret i [1] kører i tiden $O(n \cdot \lg n)$. Algoritmens pladsforbrug er $O(n \cdot \min(d_1, \lg n))$, hvor d_1 er graden af det første inputtræ, T_1.

Algoritmen til kvartetdistanceberegning kører i tiden $O(\max(d_1, d_2) \cdot n \cdot \lg n)$ hvor d_2 er graden af det andet inputtræ, T_2. Algoritmens pladsforbrug er $O(\max(d_1, d_2) \cdot n \cdot \min(\max(d_1, d_2), \lg n))$.

Vi forbedrer algoritmen til kvartetdistanceberegning. Dette reducerer kørselstiden til $O(\min(d_1, d_2) \cdot n \cdot \lg n)$. Ligeledes formindsker denne forbedring pladsforbruget til $O(\min(d_1, d_2) \cdot n \cdot \min(d_1, d_2, \lg n))$.

Vi reducerer dernæst markant antallet af beregninger, der er nødvendige for at udregne kvartetdistancen.

Igennem eksperimenter fremviser vi empirisk evidens for at implementeringen af både algoritmerne i [1], samt vores forbedringer, er teknisk mulige og yder godt i praksis.

Arbejdsform

Dette speciale, og implementeringen beskrevet heri, er produktet af et ligeligt samarbejde imellem de to forfattere.

Implementeringen blev skrevet ved hjælp af parprogrammering, og specialeteksten blev skrevet ved hjælp af “parskrivning”. Der er praktisk talt ingen sætninger i dette speciale, der er produktet af en forfatter alene.

Acknowledgements

We would like to thank our advisor, Gerth Stølting Brodal, for invaluable insights, moral support, constantly pushing us to do better and some proofreading of this thesis.

Additionally, we would like to thank Andreas Sand, for providing us with a copy of his implementation of his $O(n \cdot \lg^2 n)$ time algorithm for calculating the triplet distance, as well as an initial set of input trees.

Finally, we would like to thank MADALGO for allowing us access to their facilities.

Jens Johansen & Morten Holt,
Aarhus, June 2013.

Contents

Chapter 1

Introduction

Trees appear in many branches of science. One such branch is 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. Given such trees, it is useful to have a distance measure to compare how similar the constructed trees are [2].

Distance measures can be quite natural in some settings, e.g. euclidean distance between two points. There is, however, no natural way to compare two trees. For this reason, a number of distance measures have been proposed for both rooted and unrooted trees.

For rooted trees these distance measures include for instance the triplet distance [3].

For unrooted trees these distance measures include for instance the Robinson-Foulds distance [4], the nearest-neighbor interchange metric [5] and the quartet distance [6].

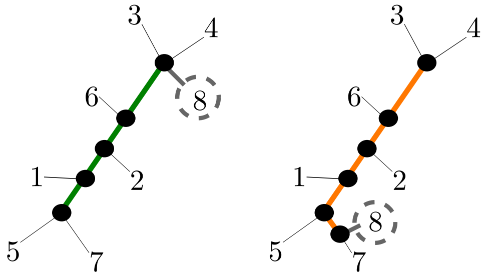
The Robinson-Foulds distance measure is defined for two unrooted trees with the same set of unique leaf-labels. It enumerates the number of non-common splits in the two trees. A split is defined as the removal of an edge in a tree, effectively splitting the tree into two, represented by two sets of leaf-labels. The number of non-common splits is the number of splits that can be performed in one tree, but not in the other.

The Robinson-Foulds distance measure can be computed in linear time [7], but is sensitive to outliers. For instance, changes to a few leaves might significantly influence the output of the algorithm. As an example, see Figure 1.1, where moving a single leaf results in two trees that are 100% different.

The nearest-neighbor interchange metric,

> [...] essentially counts the minimum number of nearest neighbor interchanges required to change one tree to another.
>
> *Waterman and Smith [5]*

It does, however, not distinguish between changes that affect the relationship between many leaves, and changes that affect only a few leaves [8]. Furthermore



(a) T_1 (b) T_2

Common	Only T_1	Only T_2
(none)	12567|348	125678|34
	1257|3468	12578|346
	157|23468	1578|2346
	57|123468	578|12346
		78|123456

(c) Splits

Figure 1.1: An example of the Robinson-Foulds distance measure on two unrooted trees. There are no common splits, and the distance is thus 100%. If the leaf labeled 8 is removed, the distance is 0%.

it has been shown to be NP-Complete [9], meaning that it is unknown whether or not polynomial-time algorithms exist.

The triplet and quartet distance measures do not suffer from the drawbacks outlined above [8]. They work by enumerating all subsets of leaves of size three and four, respectively, and counts the number of different induced topologies.

There are $\binom{n}{3}$ and $\binom{n}{4}$ different subsets for triplet and quartets, respectively, for a tree with n leaves. Simply enumerating all of the subsets, as is the case for naive algorithms, will take time at least $O(n^3)$ or $O(n^4)$, and is thus not practical for large input.

1.1 Triplets & Quartets

In this thesis we consider rooted and unrooted trees with n uniquely labeled leaves. Given three leaves labeled a, b and c in a rooted tree, the subtree induced by these leaves is denoted a *triplet*. Given four leaves labeled a, b, c and d in an unrooted tree, the subtree induced by these leaves is denoted a *quartet*.

We say that a triplet is *resolved* if two leaves have a least common ancestor which is not shared with the remaining leaf. Reversely a triplet, where the least common ancestor of all pairs of leaves contained in the triplet is the same, is denoted as *unresolved*. For three leaves labeled a, b, and c, all four possible configurations are enumerated in Figure 1.2. Note that unresolved triplet configurations only occur in non-binary rooted trees.

As for triplets, quartet can also be either resolved or unresolved, and all four possible configurations are enumerated in Figure 1.2. Note that unresolved quartet configurations only occur in unrooted trees of degree larger then three. Unrooted trees of degree three will henceforth also be denoted as *binary*.

The two configurations, resolved and unresolved, give rise to four combinations for a pair of input trees, T_1 and T_2. This is depicted in Figure 1.3.

If the induced topology of a triplet or quartet is the same in T_1 and T_2, it is denoted as *agreeing*. Reversely, if the induced topology of a triplet or quartet differ in T_1 and T_2, it is denoted as *disagreeing*. Thus, unresolved-unresolved topologies always agree, whereas resolved-unresolved and unresolved-resolved

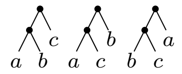
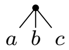
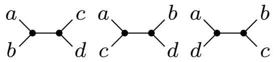
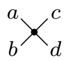
	Resolved	Unresolved
Triplet (rooted)		
Quartet (unrooted)		

Figure 1.2: All configurations for triplets and quartets.

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

Figure 1.3: Cases for topologies in the two trees.

topologies always disagree. Resolved-resolved topologies can either agree or disagree.

Having established the terminology, we note that the triplet distance between two rooted trees with the same set of leaf-labels, counts the number of triplets having disagreeing topologies in the two trees. Equivalently, the quartet distance between two unrooted trees with the same set of leaf-labels, counts the number of quartets having disagreeing topologies in the two trees. In both cases, this corresponds to the value $B + C + D$ in Figure 1.3.

For binary trees the values C, D and E will equal zero since unresolved triplets and quartets do not occur. This reduces the problem to finding B. This problem is obviously simpler, and likely explains the large number of algorithms only operating on binary trees (see Section 1.2).

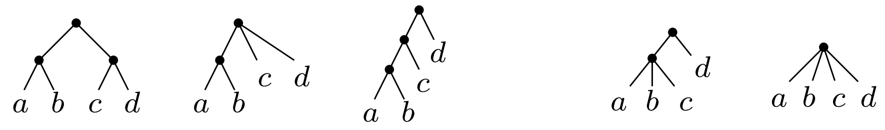
One way to calculate the triplet or the quartet distance would be to find A and E and subtract these numbers from $\binom{n}{3}$ for triplets and $\binom{n}{4}$ for quartets. Another way could be to find B, C and D directly. In fact, a mixture of these approaches are used in the algorithms presented in [1] (see Chapter 2).

To further illustrate, and give an intuition of, the triplet and quartet calculation, a naive algorithm is introduced in Section 1.1.1.

1.1.1 A Naive Algorithm

Naive algorithms work by enumerating all of the $\binom{n}{3}$ triplets or $\binom{n}{4}$ quartets in the two input trees. They then find the induced topologies and compare them.

For triplets this can be done by simply starting a walk in each of the three leaves of a triplet. Once two or more walks join, the induced topology has been found. If two, and not three, walks join, the topology is resolved. If, on the



(a) Resolved cases.

(b) Unresolved cases.

Figure 1.4: Cases for quartets in a tree rooted in an arbitrary internal node.

other hand, all three walks join at the same node, the topology is unresolved and will add to the distance.

Since the input tree can be arbitrarily unbalanced, finding the topology of a triplet can take linear time, and as such the total time for enumerating all triplets and finding their topologies is bounded by $O(n^4)$.

The process is repeated for both input trees, and A and E is found by counting the number of agreeing topologies in the two trees. This can for instance be done by representing the set of topologies as lists, sorting the two lists and doing a linear scan over them. This takes time $O(n^3 \cdot \lg n)$ for a total runtime for this naive triplet distance calculation algorithm of $O(n^4)$.

As an example of how large such a list is, consider a tree with 2,000 leaves. Such a tree will have $\binom{2000}{3}$ triplets, which all needs to be saved in the above mentioned list. Under the very optimistic assumption that each triplet can be represented by one byte, such a list will use at least 1.2GB of space.

The quartet distance can be calculated in the same way, but using four leaves instead of three. However, since the quartet distance is measured on unrooted trees, each of the two trees are first rooted in an arbitrary internal node. This can be done as described in Section 1.5.1. Again, if only two walks join, the quartet is resolved (see Figure 1.4a), whereas if three or four walks join, the quartet is unresolved (see Figure 1.4b). As a tree has $\binom{n}{4}$ quartets this naive algorithm for calculating the quartet distance, by the same argument as above, runs in time $O(n^5)$.

Again let us consider how large such a list is by looking at a tree with 2,000 leaves. Such a tree will have $\binom{2000}{4}$ quartets, which all needs to be saved in a list. Under the same optimistic assumption as before, such a list will use at least 619GB of space.

As we have seen, the naive approach is not feasible for large n. As such, other approaches are required, if we wish to calculate the triplet and quartet distances for large input. The following section outlines some of the research in this area.

1.2 Previous Work

In this section we outline some of the advancements in the area of triplet and quartet distance calculation algorithms. This outline is also presented in Tables 1.1 and 1.2.

Year	Reference	Runtime	Binary	Arbitrary
	Naive algorithm	$O(n^5)$	✓	✓
1993	Steel and Penny [2]	$O(n^3)$	✓	
2000	Bryant *et al.* [8]	$O(n^2)$	✓	
2001	Brodal *et al.* [10]	$O(n \cdot \lg^2 n)$	✓	
2004	Brodal *et al.* [11]	$O(n \cdot \lg n)$	✓	
2007	Stissing *et al.* [12]	$O(d^9 \cdot n \lg n)$	✓	✓
2011	Nielsen *et al.* [13]	$O(n^{2.688})$	✓	✓
2013	Brodal *et al.* [1]	$O(\max(d_1, d_2) \cdot n \cdot \lg n)$	✓	✓
2013	This thesis	$O(\min(d_1, d_2) \cdot n \cdot \lg n)$	✓	✓

Table 1.1: Quartet distance calculation algorithms.

In 1993, an algorithm calculating the quartet distance in time $O(n^3)$ was reported by Steel and Penny [2]. It is unclear whether or not the algorithm operates on trees of arbitrary degree.

Around the turn of the millennium this was improved to $O(n^2)$ by Bryant *et al.* [8], 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. The following year, in 2001, Brodal *et al.* [10] improved the quartet distance calculation runtime to $O(n \cdot \lg^2 n)$ for binary trees. In 2004 this was further improved to $O(n \cdot \lg n)$ by the same authors [11].

The problem of calculating the quartet distance between trees of arbitrary degree was, in 2007, addressed by Stissing *et al.* [12]. The authors gave an algorithm for calculating the quartet distance between arbitrary degree trees in time $O(d^9 \cdot n \cdot \lg n)$, where d is the maximum degree of any node in the two trees. A few years later, in 2011, Nielsen *et al.* [13] introduced an algorithm that does not depend on the degree of the input; calculating the quartet distance between arbitrary degree trees in time $O(n^{2.688})$. The somewhat atypical asymptotic runtime for this algorithm is due to the usage of matrix multiplication.

Recently, in 2013, Brodal *et al.* [1] gave an algorithm for calculating the quartet distance in time $O(d \cdot n \cdot \lg n)$.

In regards to the triplet distance calculation, an algorithm for binary trees running in time $O(n^2)$ was given by Critchlow *et al.* [3] in 1996. More than a decade later, the problem of arbitrary degree trees was addressed when Bansal *et al.* [14] in 2011 introduced an algorithm for calculating the triplet distance of arbitrary degree trees, also in time $O(n^2)$.

In 2013, Sand *et al.* [15] gave an $O(n \cdot \lg^2 n)$ time algorithm for calculating the triplet distance for binary trees. The same year, Brodal *et al.* [1] gave an algorithm for calculating the triplet distance of two trees of arbitrary degree in time $O(n \cdot \lg n)$.

It is the algorithms for calculating triplet and quartet distances presented by Brodal *et al.* [1] that are the subject of this thesis.

Year	Reference	Runtime	Binary	Arbitrary
	Naive algorithm	$O(n^4)$	✓	✓
1996	Critchlow *et al.* [3]	$O(n^2)$	✓	
2011	Bansal *et al.* [14]	$O(n^2)$	✓	✓
2013	Sand *et al.* [15]	$O(n \cdot \lg^2 n)$	✓	
2013	Brodal *et al.* [1]	$O(n \cdot \lg n)$	✓	✓

Table 1.2: Triplet distance calculation algorithms.

1.3 Existing Implementations

A number of the above mentioned algorithms have been implemented. These include, but are not necessarily limited to, the following.

Mailund and Pedersen [16] implemented the algorithm in [10], calculating the quartet distance for binary trees in time $O(n \cdot \lg^2 n)$. The authors furthermore stated that the algorithm is useful in practice. The source-code for the implementation is available for download[1].

In 2011, Nielsen *et al.* [13] documented the implementation of their algorithm, calculating the quartet distance for trees of arbitrary degree in time $O(n^{2.688})$. The authors stated that it is "[...] the fastest algorithm so far for computing the quartet distance between general trees". The source-code has been made available for download[2].

Sand *et al.* [15], in 2013, presented an algorithm, and implementation, for calculating the triplet distance between two binary trees in time $O(n \cdot \lg^2 n)$. The authors concluded that the algorithm was useful in practice. While the source code is not generally available at the time of writing, a copy has been provided to us by the authors.

In Section 4.3.6 we present experiments on these algorithms, comparing them to the algorithms described in this thesis.

1.4 Our Results & Overview of the Thesis

In this thesis we present our implementation (Chapter 3), evaluation (Chapter 4) and improvements (Section 2.7 and Chapter 5) of the algorithms presented by Brodal *et al.* [1].

The algorithms in [1] calculate the triplet distance between two trees of arbitrary degree in time $O(n \cdot \lg n)$ and space $O(n \cdot \min(d_1, \lg n))$. This will be the main topic of Chapter 2, and runtime and memory usage results will be presented in Sections 4.3.3 to 4.3.6.

The algorithms in [1] furthermore calculate the quartet distance between two trees of arbitrary degree in time $O(\max(d_1, d_2) \cdot n \cdot \lg n)$ and space $O(\max(d_1,$

[1] `http://cs.au.dk/~mailund/qdist.html`

[2] `http://birc.au.dk/software/qdist/`

$d_2) \cdot n \cdot \min(\max(d_1, d_2), \lg n))$ where d_i is the maximum degree of any node in the tree T_i.

By extending the algorithm in [1] we improve both the runtime and memory usage bounds for the quartet distance calculation between two trees of arbitrary degree. The new bounds are $O(\min(d_1, d_2) \cdot n \cdot \lg n)$ and $O(\min(d_1, d_2) \cdot n \cdot \min(d_1, d_2, \lg n))$ for time and space, respectively. These changes from a maximum to a minimum can be very significant for input consisting of one tree with large degree and one tree with small degree. The quartet distance calculation algorithm, as well as our asymptotic improvement, will be the topic of Sections 2.7 and 2.8. Furthermore, runtime and memory usage results will be presented in Sections 4.3.1, 4.3.2, 4.3.5 and 4.3.6.

From Figure 1.3 we note that given either A or B and any of the other values, both the triplet distance and the quartet distance can be calculated in linear time (see Section 2.1).

Having reduced the asymptotic runtime and memory usage of the quartet distance calculation, we furthermore decrease both the runtime and the memory usage by a constant factor. This is achieved by calculating A and E, instead of A and B as done by Brodal *et al.* [1] (see Chapter 5).

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 Section 4.1.

To our knowledge, the results presented in this thesis are the current state of the art.

1.5 Preliminaries

In this thesis we use a number of terms. These will be summarized below. The terms are also explained when they first occur. This list can be used as a reference, should the need arise.

- Triplets are induced subtrees of three leaves from a rooted tree.
- Quartets are induced subtrees of four leaves from an unrooted tree.
- T_1 and T_2 is used to denote the two input trees.
- d_1 and d_2 is used to denote the degree of T_1 and T_2, respectively.
- n is the number of leaves in each of the two trees, T_1 and T_2.
- Topology of a subtree denotes the structure of this subtree.
- Resolved denotes a triplet or quartet where all internal nodes in the induced topology are binary or has degree $\leq$ three for triplets or quartets, respectively.
- Unresolved denotes a triplet or quartet that is not resolved.
- Agreeing denotes the situation where two triplets or quartets have the same induced topology in T_1 and T_2.
- Disagreeing denotes the situation where two triplets or quartets have different induced topologies in T_1 and T_2.

- v is generally used to denote the node in T_1 that is currently being processed.
- d is generally used to denote the degree of the input, but can also be used to denote the degree of v. It will be clear from the context.
- $c_1, \ldots, c_k$ denotes the k children of v.
- Compatible is used to denote the triplets or quartets contributing to A, B or E, depending on the context.
- HDT, Hierarchical Decomposition Tree, is a locally balanced tree built atop of a rooted tree.
- **C**, **G**, **I** components are the components (i.e. nodes) of the HDT.
- Super root, for a **G** component, denotes the LCA of all leaves in the subtree rooted at the **G** component.
- External path, of a **C** component, denotes the direct path in the original tree, through the **C** component.
- 0-leaf, a single leaf, representing a contracted subtree where all nodes have the color 0. The number of leaves in the subtree are saved as an integer in the 0-leaf.
- Q denotes the denominator of the fraction used to determine if the largest subtree should be extracted and contracted.
- Fully balanced trees are perfectly balanced, i.e. all leaves are at roughly the same level in the tree.
- $x\%$ 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.
- Random describes the tree where all leaf-labels have been randomly permuted.
- Leaf moved describes the input for which T_1 is a random tree and T_2 is T_1 with the leaf-labels 1 and 2 having switched places.
- α, β, γ, δ and ϵ are different configurations for quartet topologies. The first three are defined in [1].

In all pseudo-code given, we use a C++-like syntax. This was chosen to stay close to the actual implementation.

We furthermore note that the quartet distance is defined for unrooted trees, but that the algorithm in [1] works on rooted trees. The first step of the quartet distance calculation algorithm is therefore to root the tree.

1.5.1 Rooting an Unrooted Tree

An unrooted tree can be rooted by choosing an arbitrary non-leaf node as the root of the tree. Assume that the trees consist of at least three nodes. Anything less than four does not make sense in this context. Rooting can then be done as follows.

To find an arbitrary non-leaf node, pick an arbitrary node, and check if it is a leaf. If the node is not a leaf, a non-leaf node has been found and we can continue. If the node is a leaf, it has an edge to another node. Since the tree has at least three nodes, this other node cannot be a leaf. A non-leaf node has been found and we can continue.

Once a non-leaf node has been found, this can be used as the root, and the entire tree can, in linear time, be rooted by a depth-first traversal of the tree.

Chapter 2

Theory

In this chapter we describe the algorithms presented by Brodal *et al.* [1], as well as our first variation of the quartet distance calculation algorithm. This variation improves the asymptotic runtime for the quartet distance calculation from $O(\max(d_1, d_2) \cdot n \cdot \lg n)$ to $O(\min(d_1, d_2) \cdot n \cdot \lg n)$.

In the naive algorithm sketched in Section 1.1.1, all triplets and quartets where enumerated. This is obviously inefficient, and the algorithm presented in [1] takes a different approach. Using the observation that we are not interested in which triplets and quartets differ, but only how many there are, we can ignore the specific triplets and quartets involved.

Recall that the goal of the algorithm is to calculate the value $B + C + D$ in Figure 1.3. This can be done in several different ways, depending on which combination of the values A, B, C, D and E have been computed. First note that $A + B + C + D + E = \binom{n}{3}$ and $\binom{n}{4}$ for triplets and quartets, respectively. Secondly, note that $A + B + C$ can be calculated by looking only at T_1. This sum is the number of triplets or quartets that are resolved in T_1, no matter how they occur in T_2. Equivalently, $A + B + D$ can be calculated by looking only at T_2. Both sums can be calculated in linear time using dynamic programming as described in Section 2.1.

Having calculated the above mentioned sums, the problem of finding $B + C + D$ is reduced to, for example, finding either A and B or A and E. Knowing A and E, the calculation in Equation (2.1) can be used to calculate the triplet distance. This is the approach taken in [1].

For the quartet distance, Equation (2.1) can be used with $\binom{n}{3}$ replaced by $\binom{n}{4}$.

Knowing A and B the calculation in Equation (2.2) can be performed for both the triplet and quartet distance. For the quartet distance, this is the approach taken in [1].

Note that the dynamic programming step is only necessary when calculating A and B, since the above mentioned sums are not used in Equation (2.1).

$$B + C + D = \binom{n}{3} - A - E \tag{2.1}$$

$$B + C + D = (A + B + D) - A + (A + B + C) - A - B \tag{2.2}$$

In the following sections we present the steps necessary for creating the algorithms, such that they run in time $O(n \cdot \lg n)$ and $O(\min(d_1, d_2) \cdot n \cdot \lg n)$ for the triplet and quartet distance calculation, respectively, as claimed in Section 1.4. All time analysis up to, but excluding, Section 2.6, considers only the triplet distance calculation. The effect on the algorithm by introducing quartets, as presented in [1], as well as our first variation, is discussed in Sections 2.6 and 2.7. The memory usage is analyzed in Section 2.8.

2.1 Dynamic Programming

As mentioned above, the first part of the algorithm is to determine the sums $A+B+C$ and $A+B+D$, corresponding to the number of triplets or quartets that are resolved in T_1, no matter how they occur in T_2 and vice versa. This is done using dynamic programming, and can be calculated in linear time using a post-order depth-first traversal of the two trees, one at a time.

Given a rooted tree T, the basic idea is to count the number of unresolved triplets and quartets rooted at a node v of degree d. For quartets we first root the tree arbitrarily in an internal node as described in Section 1.5.1. For a node v, we let n^v denote the number of leaves in the subtree rooted at v, and let n_i^v denote the number of leaves in the subtree rooted at the ith child of v.

To count the number of unresolved triplets and quartets rooted at a node v, we calculate the values s_i^v, p_i^v, t_i^v and q_i^v. Here s_i^v designates the number of leaves in the first i subtrees below the node v. Similarly p_i^v, t_i^v and q_i^v designate the number of sets of two, three and four leaves, respectively, in the first i subtrees of v.

The values s_i^v, p_i^v, t_i^v and q_i^v are computed as follows:

$$s_i^v = \begin{cases} n_1^v & \text{if } i = 1\ , \\ s_{i-1}^v + n_i^v & \text{otherwise.} \end{cases} \tag{2.3}$$

$$p_i^v = \begin{cases} 0 & \text{if } i = 1\ , \\ p_{i-1}^v + n_i^v \cdot s_{i-1}^v & \text{otherwise.} \end{cases} \tag{2.4}$$

$$t_i^v = \begin{cases} 0 & \text{if } i = 1\ , \\ t_{i-1}^v + n_i^v \cdot p_{i-1}^v & \text{otherwise.} \end{cases} \tag{2.5}$$

$$q_i^v = \begin{cases} 0 & \text{if } i = 1\ , \\ q_{i-1}^v + n_i^v \cdot t_{i-1}^v & \text{otherwise.} \end{cases} \tag{2.6}$$

Having calculated these values, the number of unresolved triplets rooted at v is the value t_d^v. Additionally, the number of unresolved quartets rooted at v is the value $q_d^v + t_d^v \cdot (n - n^v)$. Here q_d^v is the number of unresolved quartets where all four leaves are distinct subtrees below v (see Figure 2.1a) and $t_d^v \cdot (n - n^v)$ is the number of unresolved quartets where one of the four leaves is not in the subtree of v (see Figure 2.1b). For each node, v, these two values are propagated towards the root.

Having calculating the above on T_1, the algorithm can read off the value $D + E$ in the root node in constant time. The values read are the propagated

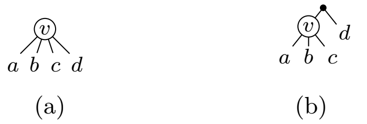


Figure 2.1: The two types of unresolved quartets counted by the dynamic programming.

values t_d^v or $q_d^v + t_d^v \cdot (n - n^v)$, for triplets or quartets respectively. Subtracting this from either $\binom{n}{3}$ or $\binom{n}{4}$ yields the sum $A + B + C$ for triplets or quartets, respectively. Equivalently, calculating the above on T_2, the sum $A + B + D$ can be found.

2.2 Coloring

Having calculated $A + B + C$ and $A + B + D$, [1] approaches the problem by finding either A and E or A and B, for triplets or quartets, respectively. Note that while the description below mentions only triplets, the same approach is applied when calculating the quartet distance.

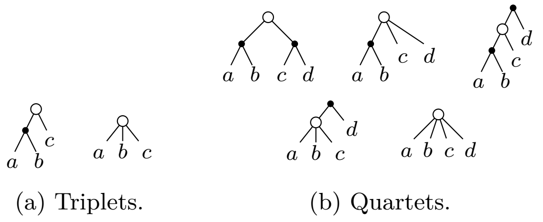
The approach taken is to see a triplet as anchored at a node v in T_1 (see Figure 2.2a). By looking at all triplets in T_1 rooted at v, and locating their counterparts in T_2 given by the same set of leaf-labels, the triplets can be categorized as A, B, C, D or E. As all triplets are anchored in some node, traversing all inner nodes of T_1 will find the anchor points of all triplets.

This can be done by a recursive coloring of the trees, using the colors $0, 1, 2, \ldots, d$.

The first step towards making the approach work, is to link the leaves of T_1 and T_2 with bidirectional pointers, a with a, b with b etc. Using the bidirectional pointers, the coloring of T_2 can be updated in constant time when recursively coloring T_1.

The algorithm maintains the following invariant during a recursive traversal of T_1:

> *When entering a node v* [in T_1], *all leaves in the subtree of v have the color 1, and all leaves not in the subtree of v have the color 0. When exiting v, all leaves in T_1 have the color 0.*
>
> *Brodal* et al. *[1]*



(a) Triplets.

(b) Quartets.

Figure 2.2: Different anchorings of triplets and quartets. The anchor nodes are white.

```
void recursiveColoring(RootedTree v)
{
  if (v->isLeaf())
    v->color(0);
  else
  {
    v->makeFirstChildLargest(); // Let $c_1$ be the child of v with the largest subtree, and let $c_2,\ldots,c_k$ be its remaining children
    for (int i=2; i <= k; i++)
      $c_i$->color(i);
    // Leaves are now colored according to v

    findNumberOfNodesCompatibleWithColoring(); // Ignored for now

    for (int i=2; i <= k; i++)
      $c_i$->color(0);

    recursiveColoring($c_1$);

    for (int i=2; i <= k; i++)
    {
      $c_i$->color(1);
      recursiveColoring($c_i$);
    }
  }
}
```

Code 2.1: Pseudo-code for the recursive coloring. Adapted from [1].

We will argue that this holds when describing the algorithm below. To initially satisfy the invariant, all nodes are colored 1 in the beginning of the algorithm.

The base-case of the recursion is when the visited node is a leaf. In this case the node is colored 0 and the recursion returns. This satisfies the last part of the invariant.

For a recursive step, when visiting a node v in T_1, let $c_1, \ldots, c_k$, be the children of v, where c_1 is the root of the largest subtree. Leaves in a subtree c_i, $i > 1$, are recolored by the color i. Note that, by the invariant, the leaves of c_1 are already colored 1. At this point, another method is called to find the number of triplets, categorized as A or E, contributed by the current coloring. These are henceforth denoted as *compatible* with the coloring. For now we ignore this step.

After this, all leaves in subtrees c_i, $i > 1$, are recolored with the color 0 and a recursive call on c_1 is performed. Note that this satisfies the invariant that upon entering c_1, all leaves in c_1 are colored 1 and all other leaves are colored 0.

Once returned from this call, the entire tree is colored 0 per the invariant. Each child c_i, $i > 1$, can now have their entire subtree colored 1 and be the input to a recursive call, again satisfying the invariant. The pseudo-code for this algorithm can be seen in Code 2.1.

Ignoring how to find the values A and E, the recursive coloring takes time $O(n \cdot \lg n)$. This is easily seen as a leaf is only recolored for a child c_i of a node v, if c_i is not the largest child of v. As such, a leaf can only be recolored once for each ancestor not being the root of the largest subtree of v, each of which having size at most $\frac{|v|}{2}$. This effectively at least halves the number of leaves rooted at each of these ancestors, leading to at most $O(\lg n)$ ancestors, and thus $O(\lg n)$ recolorings, per leaf. With n leaves this yields a total recoloring charge of $O(n \cdot \lg n)$.

In the following sections we introduce the Hierarchical Decomposition Tree. We furthermore show how to use this tree to find the number of triplets compatible with the coloring of a given node v in T_1.

2.3 Hierarchical Decomposition Tree (HDT)

To find the number of triplets or quartets compatible with a coloring, [1] maintains a number of counters (see Section 2.5).

A naive approach to this counting scheme would be to count directly in T_2. As we shall see, however, this will not result in the desired asymptotic runtime.

Since T_2 is given as input to the algorithm, we can make no assumptions as to how balanced the tree is. A perfectly unbalanced tree (i.e. a very long chain) is indeed valid input.

All counters in internal nodes are based on counters from their children. As such, changing the color of one or more leaves, requires the counters of all ancestors for these leaves to be updated. This means that even if just a single leaf is recolored in T_1, we might have to visit all nodes in T_2. At least one leaf is recolored in each step of the coloring algorithm, i.e. for each internal node of T_1. Since there are $O(n)$ internal nodes, the algorithm takes $O(n^2)$ time if operating directly on T_2. Note that this is under the assumption that each node, given updated information from their children, can be updated in constant time.

The problem above is that T_2 can have height $O(n)$. If we can guarantee a smaller height, we can also obtain a faster runtime.

Let a locally balanced tree be a tree which is balanced in all internal nodes, i.e. every subtree rooted at a node v has height $O(\lg |v|)$, where $|v|$ is the number of leaves in the subtree of v. A solution to the problem above is thus to construct such a locally balanced version of T_2. In [1], this is done using a Hierarchical Decomposition Tree, or HDT, of T_2. This is explained below.

2.3.1 Basic Idea

As stated above, an HDT is used to construct a locally balanced tree on top of an unbalanced tree, T, of arbitrary degree. The basic idea is to let each component in the HDT correspond to a set of nodes in the input tree, T, where the leaves correspond directly to nodes in T. For this purpose [1] defines four different types of components in an HDT (see Figure 2.3):

L A leaf in T.
I An internal node in T.
C A connected subset of nodes in T.
G A set of subtrees with roots being siblings in T.

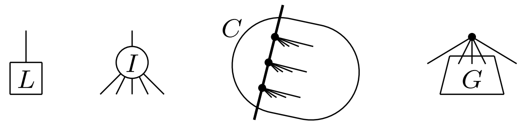


Figure 2.3: Different types of nodes of an HDT (courtesy of [1]).

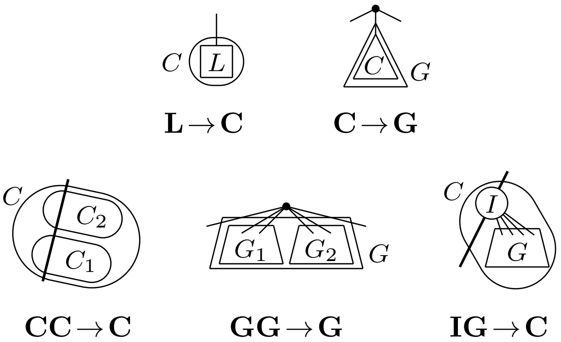


Figure 2.4: Transformations and compositions of HDT nodes (courtesy of [1]).

While an **L** component corresponds to a leaf in T, these are only needed during the initial construction of the tree, and will not occur in the final tree.

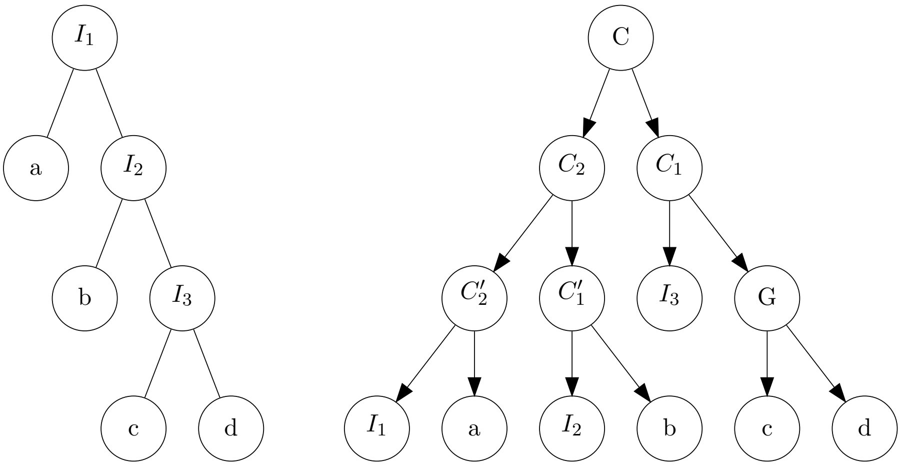
For **C** components we let the direct path in the original tree, T, going through the **C** component be denoted as the *external path*. **C** components are at most allowed to have two edges in T, crossing the boundary of the component; one going upwards and one going downwards. Neither of these edges are required.

For **G** components we let the LCA of all leaves in the subtree of the HDT rooted at this component be denoted as the *super root*. **G** components are downwards closed, meaning that there is no edge in T from a node inside the component to a child, which crosses the boundary of the component.

Additionally, [1] defines five transformations and compositions over these components, helping to ensure the structure of the HDT as well as enforcing that the HDT is locally balanced. These are depicted in Figure 2.4. Since each composition merges two components into one, we can view the merged component as the parent of the two original components, giving rise to the binary property of the HDT.

It should be noted, that during the construction of the HDT (see Section 2.3.2), we operate over two different types of children, namely the original relationship in T, and the new relationship between compositions. Additionally, some of the invariants of a fully constructed HDT are allowed to be broken during the construction.

An example of a rooted tree, and the HDT constructed based on the tree can be seen in Figure 2.5.



(a) The original tree.

(b) The result, an HDT.

Figure 2.5: Example of creating an HDT for a tree.

```
HDT preFirstRound(RootedTree t)
{
  if (t->isLeaf())
  {
    // Construct a new G component & mark as C -> G.
    HDT hdt = new HDT(G);
    setTypeGConvertedFromC(hdt);
    return hdt;
  }

  // Inner node (i.e. an I component)
  HDT component = new HDT(I);
  // Loop over the children of t, and convert them to HDT components
  for($c_i$ in t->children)
    component->addChild(preFirstRound($c_i$));
  return component;
}
```

Code 2.2: Pseudo-code for the `preFirstRound` function of the HDT construction.

2.3.2 Construction

We know from Lemma 4.1 in [1] that the HDT can be constructed in linear time in the size of the input tree and yields a locally balanced tree. The algorithm is described below.

The HDT construction works by calling the function `preFirstRound` (see Code 2.2) followed by calls, with a geometrically decreasing set of components as input, to the function `round` (see Code 2.3).

The `preFirstRound` function converts the rooted input tree into an, illegal, HDT. This is done by a depth-first traversal of the input tree. During this traversal, all internal nodes are converted to **I** components, and all leaf nodes are converted to **G** components which are marked as converted from **C** components. This is in contrast to the algorithm presented in [1], which states that leaves should be constructed as **L** components. However, the first thing done after the `preFirstRound` function in [1] is to convert all **L** components into **C** components. Furthermore, the first step in each round is to convert all downwards closed **C** components being children of **I** components into **G** components. Thus all **L** components would be converted to **C** components, which, being downwards closed children of **I** components, would then be converted into **G** components. As such we can safely convert all leaf nodes in the input tree to childless **G** components marked as being converted from **C** components. This removes the need for the **L** component.

After converting applicable **C** components, the `round` function proceeds to perform 3 types of non-overlapping compositions on the current HDT.

Composition 1 (GG → G). For an **I** component, with at least two **G** components as children, the **G** components are paired arbitrarily, and a **GG → G** composition is performed on each pair.

Composition 2 (IG → C). For an **I** component having at most one downwards open child, in [1] denoted non-forking, and where only one child is a **G** component, an **IG → C** composition is performed.

Composition 3 (CC → C). A subpath of consecutive **C** components is paired, the top-most with the second, the third with the fourth, etc., and a **CC → C** composition is performed on each pair, resulting in a path of half the length.

Recall that the construction of the HDT operates over two different types of children, namely the original relationship in T, and the new relationship between compositions. The three compositions described above are performed as long as the root of the HDT has children from the original relationship in T.

Adding the HDT to the pseudo-code in Code 2.1 we end up with the pseudo-code in Code 2.4.

2.3.3 The Cost of Recoloring

We now consider the cost of recoloring a set of leaves rooted at the node v (see Section 2.2). From Lemma 2 in [11] we know that the union of k root-to-leaf paths in a locally balanced, rooted, binary tree with n leaves contains at most $O(k + k \cdot \lg \frac{n}{k})$ nodes.

Thus, if each leaf rooted at a child $c_i,\ i > 1$, of v pay

$$O\left(\frac{|c_i| + |c_i| \cdot \lg \frac{n}{|c_i|}}{|c_i|}\right) = O(1 + \lg \frac{n}{|c_i|}) \leq O(\lg n) \tag{2.7}$$

per recoloring, any component in the HDT with x colors below it has implicitly already paid x and can thus update its $O(x)$ counters for free, i.e. in constant time.

Thus, if we pay $O(\lg n)$ per leaf-recoloring for updating the HDT, i.e. a $\lg n$ factor for all $O(n \cdot \lg n)$ recolorings, we have paid enough, yielding a total runtime for the algorithm of $O(n \cdot \lg^2 n)$.

```
HDT round()
{
  // Composition 3: CC -> C
  if (type == C && childCount() == 1 && firstChild->type == C)
  {
    HDT newC = new HDT(C, this, firstChild);
    // If there are children, there is only 1 (because of IG -> C).
    // We recurse on that child and add the result to our child-list
    if (firstChild->hasChildren())
      newC->addChild(firstChild->getFirstChild()->round());
    return newC;
  }

  for(i in children)
  {
    // Transform all I-child downwards closed C comp. into G comp.
    if (type == I && i->type == C && i->isDownwardsClosed())
      setTypeGConvertedFromC(i);

    // Composition 1: GG -> G
    if (i->type == G) {
      if (foundOneG()) mergeThisWithPreviousGAndSetFoundOneGToFalse();
      else setFoundOneG(true);
      continue;
    }

    i->round(); // Recurse on the child
  }

  // Composition 2: IG -> C
  if (type == I && downwardsOpenChildren < 2 && gChildren == 1)
  {
    HDT newC = new HDT(C, this, lastG);
    for(i != lastG in children) newC->addChild(i);
    return newC;
  }

  return this;
}
```

Code 2.3: Pseudo-code for the `round` function of the HDT construction. Note that `children` represents a list of, possibly merged, edges from the input tree, i.e. distinct from HDT children of which there are always two for internal components and zero for leaves.

While this is better than without the HDT, we are still a $\lg n$ factor from the claimed runtime. Removing this additional $\lg n$ factor is the subject of the next section.

```
void count(RootedTree v)
{
  if (v->isLeaf())
    v->color(0);
  else
  {
    v->makeFirstChildLargest(); // Let $c_1$ be the child of v with the largest subtree, and let $c_2, \ldots, c_k$ be its remaining children
    for (int i=2; i <= k; i++)
      $c_i$->color(i);
    // Leaves are now colored according to v

    hdt->query(); // Query the HDT for the number of triplets/quartets in $T_2$ compatible with the coloring
    addHDTNumbersToGlobalCounter();

    for (int i=2; i <= k; i++)
      $c_i$->color(0);

    count($c_1$);

    for (int i=2; i <= k; i++)
    {
      $c_i$->color(1);
      count($c_i$);
    }
  }
}
```

Code 2.4: Pseudo-code for the main-algorithm. Adapted from [1].

2.4 Extract & Contract

Per the analysis in Section 2.3.3, the runtime, when using one static HDT, becomes $O(n \cdot \lg^2 n)$. This can, however, be improved to $O(n \cdot \lg n)$ by ensuring that the HDT is always of size $O(|v|)$ when entering a node v on a call `count(v)`.

Let us first assume that the HDT has size $O(|v|)$ to see that this statement holds true. In this case, every leaf can be charged

$$O\left(\sum_{j=1}^{k} 1 + 1 + \lg \frac{|v_j|}{|v_{j+1}|}\right) \;, \tag{2.8}$$

where we sum over all the ancestors where a leaf is recolored. The nodes v_j and v_{j+1} are two ancestors where recoloring occurred, possibly separated by nodes where recoloring did not occur. The initial 1 is for the recoloring and $1 + \lg \frac{|v_j|}{|v_{j+1}|}$ is for the $|v_{j+1}|$ root-to-leaf paths in a tree of size $|v_j|$ traversed when counting.

As, per Section 2.2, there is at most $O(\lg n)$ terms in the sum, and as

$$\begin{aligned} O\left(\sum_{j=1}^{k} \lg \frac{|v_j|}{|v_{j+1}|}\right) &= O\left(\sum_{j=1}^{k} \lg |v_j| - \lg |v_{j+1}|\right) & (2.9)\\ &= O(\lg |v_1| - \lg |v_2| + \lg |v_2| - \lg |v_3| \ldots - \lg |v_{k+1}|) & (2.10)\\ &= O(\lg |v_1| - \lg |v_{k+1}|) & (2.11)\\ &\leq O(\lg n) \;, & (2.12)\end{aligned}$$

the total runtime becomes $O(n \cdot \lg n)$.

To achieve the needed $O(|v|)$ size of the HDT when entering any node v in a call to `count(v)`, two extra functions, `extract` and `contract`, are added. The function `extract`, given a color i, extracts a partial HDT consisting of the leaves with this color, as well as additional information regarding the parts of the HDT not otherwise included (see Section 2.4.1). The function `contract` takes a tree that was previously extracted and, without changing the induced topology of any real leaves, creates a smaller copy of this tree (see Section 2.4.3).

Before the recursive call `count(`c_1`)`, the tree is extracted and contracted if the size of c_1 is less than some constant fraction of the current HDT.

This can be done in $O(|v|)$ time, and since it is only done if the size of c_1 is less than some constant fraction of the current HDT, every subsequent call is on something of a fraction of the size (i.e. geometrically decreasing). The first call will thus pay for all subsequent calls. As such, the next parts can safely ignore the time this takes.

Just after the query to the HDT the `extract` and `contract` functions are applied to all subtrees $c_i,\ i > 1$, to produce a new HDT for each subtree, each of size $O(|c_i|)$.

First an HDT is extracted in time $O(|c_i| + |c_i| \cdot \lg \frac{|v|}{|c_i|})$ (see Section 2.4.1). This produces an HDT of size $O(|c_i| + |c_i| \cdot \lg \frac{|v|}{|c_i|})$.

This HDT is then converted into the tree it represents (see Section 2.4.2), in the same time, yielding a tree of the same size. This tree is then contracted (see Section 2.4.3) to yield a tree of size $O(|c_i|)$, also in time $O(|c_i| + |c_i| \cdot \lg \frac{|v|}{|c_i|})$. Finally an HDT of size $O(|c_i|)$ can be constructed in time $O(|c_i|)$.

All of these charges are bounded by the number of nodes in the union of $|c_i|$ root-to-leaf paths in a locally balanced, rooted, binary tree with $|v|$ leaves. As such, all of this is paid for by charging $O(1 + \lg \frac{|v|}{|c_i|})$ from each leaf (see Equation (2.7)).

This gives us the needed $O(|v|)$ size of the HDT when entering any node v in a call to `count(v)`, and we thus finally achieve the claimed runtime of $O(n \cdot \lg n)$.

The pseudo-code of the main algorithm, after the addition of `extract` and `contract`, is given in Code 2.5.

The details of how to extract, convert an HDT into the tree it represents and contract are given below.

2.4.1 Extracting

Extracting is done by, in the HDT, marking the $O(|c_i| + |c_i| \cdot \lg \frac{|v|}{|c_i|})$ root-to-leaf paths of the $|c_i|$ leaves in the subtree c_i of T_1. Then, via a top-down traversal, a modified copy of the HDT is created. In this copy, all unmarked subtrees are replaced by new subtrees of size $O(1)$. Each replacement-subtree represents a subtree of the same size as the subtree replaced, where all leaves have been colored 0. Each of these replacement-subtrees will later be converted into what we call 0-leaves, which, as an integer, saves the number of leaves colored 0 in

```
void count(RootedTree v)
{
  if (v->isLeaf())
    v->color(0);
  else
  {
    v->makeFirstChildLargest(); // Let $c_1$ be the child of v with the largest subtree, and let $c_2,\ldots,c_k$ be its remaining children
    for (int i=2; i <= k; i++)
      $c_i$->color(i);
    // Leaves are now colored according to v

    hdt->query(); // Query the HDT for the number of triplets/quartets in $T_2$ compatible with the coloring
    addHDTNumbersToGlobalCounter();

    for (int i=2; i <= k; i++)
      $hdt_i$ = constructHDT(hdt->extract(i)->goBack()->contract());

    for (int i=2; i <= k; i++)
      $c_i$->color(0);

    if (hdt->tooLarge($c_1$))
      hdt = constructHDT(hdt->extract(1)->goBack()->contract());

    count($c_1$);

    for (int i=2; i <= k; i++)
    {
      hdt = $hdt_i$;
      $c_i$->color(1);
      count($c_i$);
    }
  }
}
```

Code 2.5: Pseudo-code for the main-algorithm, extended with extract and contract. Adapted from [1].

the represented subtree. The replacement-subtrees should be created in such a way, that the resulting HDT is valid, i.e. a **CC** → **C** composition should still have two **C** children, etc. This is, however, not a problem as any subtree can be replaced by a valid subtree of size $O(1)$. For instance a **C** component can be replaced by an **IG** → **C** composition with a new **I** component and a new leaf component.

All this is done in time $O(|c_i| + |c_i| \cdot \lg \frac{|v|}{|c_i|})$ and produces an HDT of size $O(|c_i| + |c_i| \cdot \lg \frac{|v|}{|c_i|})$.

2.4.2 HDT to Rooted Tree

An HDT represents a regular tree in a locally balanced and binary way. It is possible to convert an HDT back into the tree it represents. We will in the following section present how to perform this conversion.

The overall idea is to convert the HDT via a depth-first traversal. As each component in the HDT is in the following only visited once, the runtime is linear in the size of the input. As the output tree consists of all leaves of the input tree, the asymptotic size of the output tree is equal to that of the HDT.

Before continuing we note that there are only two relevant component types: **C** and **G** components. As the **I** components are leaves in the HDT and will not be recursed upon, they are not relevant here.

To facilitate the algorithm we define the following invariant for the two component types.

> Recursions on **C** components get no input. The recursion returns two values, namely the uppermost node and bottommost node on the external path of the component.
>
> Recursions on **G** components get as input the super root of the component, to which children add themselves. The recursion returns nothing.

An HDT has a limited number of compositions. We will here go through all compositions and explain how to interpret that compositions as a regular tree. Because of the complicated inner workings of the HDT, this section is quite technical. In an attempt to make things more clear, a small example of an HDT is given in Figure 2.6a and the result of the conversion is given in Figure 2.6b.

Leaf. When reaching the leaf level, the component can simply be converted to a regular tree leaf, and be added to the child-list of the parent given as input. As a leaf is a **G** component, the input is given and the invariant is satisfied.

This step is illustrated in Figures 2.6a and 2.6b where the leaves labeled a, b, c and d are present in both trees, having been simply converted.

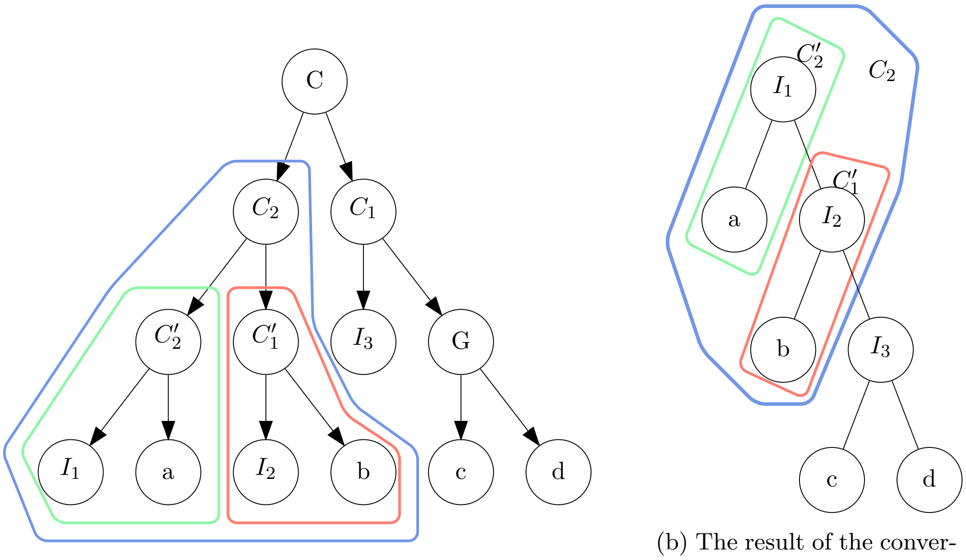
GG → G. In this case we simply recurse on both children, giving both the same input as this composition, and return nothing. This is correct as, per the invariant, **G** components add themselves to the child-list of the input.

An example of this is the leaves labeled c and d which have added themselves to the internal node created by the **I** component. These are all non-marked in Figures 2.6a and 2.6b.

IG → C (possibly then converted to a G component). The **I** component represents an internal node in the original tree, and as such a new tree-node is created to represent this. We then recurse on the **G** component with this new node as the input. This satisfies the invariant that recursions on **G** components are being called with the super root of the component as the input.

Returning from the recursion, if the current component is a **C** component, we return the new node as both the uppermost node and the bottommost node. If, on the other hand, the current component is a **G** component, the new node is added as a child to the input and we return nothing. In both cases, the invariant is satisfied.

In Figure 2.6a the **IG → C** composition C_2', the left- and bottommost subtree consisting of three components, has been marked with green. The result of the



(a) A small HDT.

(b) The result of the conversion.

Figure 2.6: Example of converting an HDT into the tree it represents.

conversion, the root and the leaf labeled a, has also been marked with green in Figure 2.6b. Another example, C'_1, of the same composition is marked with red in both trees.

CC → C (possibly then converted to a G component). Recall the structure of the **CC** → **C** composition from Figure 2.4. We first recurse on both children. Neither is given any input, satisfying the invariant that recursions on **C** components are called without input. Note that there will be an **IG** → **C** component below each **C** component, responsible for creating new internal nodes.

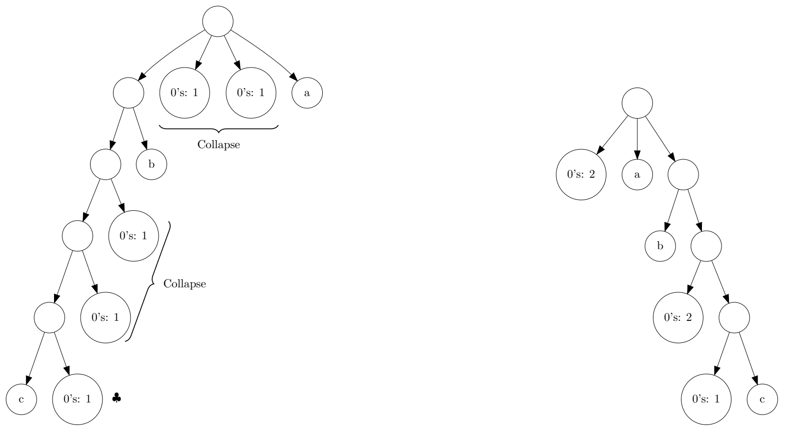
The uppermost return value from the recursive call on C_1 is added as a child to the bottommost return value from the recursive call on C_2. This is due to C_1 being below C_2 in its entirety.

If the current component is a **C** component we return the uppermost return value from C_2 and the bottommost return value from C_1 as the uppermost and bottommost return value, respectively.

If, on the other hand, the current component is a **G** component, we add the uppermost return value from C_2 as a child to the input and return nothing. In both cases, the invariant is satisfied.

In Figure 2.6a, the **CC** → **C** composition C_2, the entire left subtree below the root, has been marked with blue. Both children of this subtree are recursed upon. The uppermost return value from the C'_1 child, marked with red, is added as a child to the bottommost return value from the C'_2 child, marked with green. The uppermost return value becomes the uppermost return value from C'_2 and the bottommost return value becomes the bottommost return value from C'_1.

As can be seen from Figure 2.6b, everything in the subtree C_1 has to go below everything in the subtree C'_1. This is achieved as, after a recursive call in



(a) Before contracting the tree.

(b) After contracting the tree.

Figure 2.7: Contracting the tree. Notice that 0-siblings have been merged, binary chains have been collapsed, and the 0-node marked by ♣ has not been collapsed as this would change the induced topology.

C on both C_1 and C_2, the uppermost return value from C_1 is added as a child to the bottommost return value from C_2.

Using the above we convert an HDT into the tree it represents. This allows us to contract the tree.

2.4.3 Contracting

The `contract` function continuously merges nodes, without changing the induced topology of the real leaves. This means that for all nodes, 0-leaf-siblings are merged by replacing them with a single leaf, storing the total number of leaves represented by the merged 0-leaves.

After merging siblings, all binary chains where each node in the chain has a 0-leaf child and a non-leaf child, are replaced by a single node representing the chain. This node has two children, namely the real subtree of the bottom-most node, and a single 0-leaf representing the sum of all previous 0-leaves in the chain.

An example of an extracted rooted tree, and its contracted counterpart can be seen in Figure 2.7.

2.5 Counters & Notation

To perform the counting in the HDT, a large number of counters and sums are used. Each counter represents the number of sets of leaves with a specific topology in the HDT-subtree rooted at the given HDT component. These counters are then used in sums to find the values, A, B or E, needed for the triplet or quartet distance calculation.

The counters presented in [1], as well as the sums utilizing these counters, with a few errors corrected, are included in Appendix A. Furthermore, recall from Section 1.4 that we have improved the runtime of the quartet distance calculation algorithm. This is done using additional counters and sums. The motivation behind these is presented in Section 2.7, and the counters and sums are included in Appendix B.

Counters appear as e.g. n_i^X where X is the component type in which the counter appears. When the component type is specified as X, the counter is used for both **C** and **G** components. The subscript indicates the coloring and topology of the set.

Each counter with an i in the subscript, is indexed by this color. As such, the counter from before, n_i^X, counts the number of leaves with the color i in the HDT-subtree rooted at the component. As this counter is indexed, with x colors in use below a component, this single counter actually represents x counters, one for each color. This can be generalized to e.g. $n_\bullet^X$ where $\bullet$ means any color other than 0. This counter thus describes the number of non-0 colored leaves below the component. As such, it can be viewed as an aggregate counter. As this counter is not indexed, only one exists per HDT component.

The two can also be combined, e.g. $n_{i\bullet}^X$ describes the number of pairs of nodes, where one node is colored i, and the other is colored different from 0 and i. For a **C** component, the nodes must furthermore be anchored on the external path of the **C** component. For a **G** component, the nodes must furthermore be in two distinct subtrees of the super root of the **G** component.

The descriptor $\square$ is used like $\bullet$ but also excludes the value taken by $\bullet$. For instance, $n_{\bullet\square}^X$ describes the number of pairs of nodes, where both nodes have a color different from 0, and the two colors are distinct. As such, the descriptor $\square$ will never occur without the $\bullet$ descriptor. As for $n_{i\bullet}^X$, the nodes must be anchored on the external path or be in two distinct subtrees of the super root, for a **C** or **G** component, respectively.

The color 0 can also be used directly as a descriptor, and the counter n_{0i}^X is thus the number of pairs anchored as above, where one node in the pair has the color 0, whereas the other has the color i.

For **C** components the notation $\uparrow$ is also used, e.g. $n_{i\uparrow\bullet}^C$. Here it is required that leaves colored i appear below leaves colored different from 0 and i (recall the structure of the **CC**→**C** composition, see Figure 2.4).

Additionally, parenthesis are used to indicate that the nodes are not allowed to be anchored on the external path, or in two distinct subtrees of the super root, for a **C** or **G** component, respectively. Finally, in [1], brackets are used when there are no requirements regarding the anchoring of the leaves.

All of the previous was introduced in [1]. We have further extended this notation with the notation $n_{\bullet-i-j}^X$ as a shorthand for the calculation $n_\bullet^X - n_i^X - n_j^X$. This particular calculation results in the number of leaves in the subtree rooted at this component, colored different from 0, i and j.

Some counters and sums have the symbol $\star$ next to them. This indicates that they are slower to calculate than the unmarked counters and sums. With x colors in use below a component, these counters and sums will require the processing of $O(x^2)$ counters.

The $\star$ occurs when the counter has two indexes or we need to sum over such a counter. For instance n_{ij} has two indexes, i and j. The need to sum over such counters occurs in the sums as well as counters where we need to synchronize across component-boundaries. An example is the counter $n^{X}_{\bullet(i\square)}$, where the two aggregates $\bullet$ and $\square$ must be different and in two distinct subtrees. For instance, in the $\mathbf{GG} \to \mathbf{G}$ composition, the counter needs to be rewritten as the sum, over all colors j, of $(n_\bullet - n_i - n_j) \cdot (n_{(ij)})$, where each parenthesis is from a different child of $\mathbf{G}$.

Put differently, synchronizing across component-boundaries yields an implicit double-index.

We note that none of the counters and sums used by the triplet distance calculation (marked $\dagger$), are marked with the symbol $\star$. Thus, if x colors are in use below a component, only $O(x)$ counters require processing and the runtime of the triplet distance calculation becomes $O(n \cdot \lg n)$ (see Sections 2.3.3 and 2.4).

2.6 Quartets

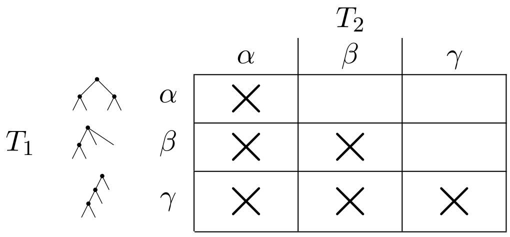
As per Sections 2.4 and 2.5 the triplet distance for both binary and arbitrary degree trees can be calculated in time $O(n \cdot \lg n)$. Calculating the quartet distance instead follows the same principle, but requires more counters and sums. Some of these counters have two indexes (be it explicit or implicit, see Section 2.5).

With two indexes on a counter, we need to process up to $O(d^2)$ counters instead of only up to $O(d)$ counters for single index counters. This yields an additional d factor. These counters are the ones marked with the symbol $\star$ in Appendices A to C.

The d factor is given by the degree of the node in T_1 everything is currently colored according to. As each internal node in T_1 is, at some point, the node everything is colored according to, d becomes the maximum degree of any node in T_1.

When counting resolved quartet topologies, they occur in three configurations, in [1] named α, β and γ (see Figure 2.8). This gives rise to nine different combinations that must be handled when calculating A and B. The algorithm in [1] reduces this to six by handling only the cases marked in Figure 2.8, swapping T_1 and T_2 and redoing the calculation for the three missing symmetric values. In this way, both input trees serve their turn as T_1 and thus d becomes the maximum degree for any node in the two trees.

The runtime for the quartet distance calculation, as presented in [1], thus becomes $O(\max(d_1, d_2) \cdot n \cdot \lg n)$. For binary trees this can be viewed as $O(n \cdot \lg n)$ as the degree is fixed, i.e. $d_1 = d_2 = 2$.



			T_2	
		α	β	γ
	α	X		
T_1	β	X	X	
	γ	X	X	X

Figure 2.8: Counted quartet configuration combinations for A and B in [1].

2.7 Improving the Bound

Recall that the quartet distance calculation algorithm between arbitrary degree trees presented in [1] runs in time $O(\max(d_1, d_2) \cdot n \cdot \lg n)$. Also recall that max is caused by swapping the two input trees. Furthermore recall that swapping is only necessary as not all α, β and γ combinations are handled directly by the algorithm presented in [1].

We improve the bound of the algorithm by handling the previously unhandled cases directly. In doing so we remove the need for swapping the input trees. Using the tree with the smallest maximum degree as T_1 we thus achieve a bound of $O(\min(d_1, d_2) \cdot n \cdot \lg n)$. The change from a maximum to a minimum can, as we shall see in Section 2.8 and Figures 4.5 and 4.11b, be very beneficial for both the running time and the memory usage. In regards to the memory usage, however, note that when $d_1 = d_2$, the memory usage of this improvement actually increases by a constant factor (see Figure 4.9b). It does however, even in such a case, remove some of the obvious runtime overhead of running the algorithm twice (see Figure 4.1a).

The counters and sums needed to handle the previously unhandled cases are contained in Appendix B.

2.8 Memory Usage

Here we analyze the asymptotic memory usage of the algorithms presented in this thesis.

Lemma 1. *The space usage of the quartet distance calculation algorithm of both [1] and our variations is $O(d \cdot n \cdot \min(d, \lg n))$. For [1], $d = \max(d_1, d_2)$, whereas for our variations, $d = \min(d_1, d_2)$.*

Proof. Two upper-bounds can be found for the space usage.

We first note that the HDT has $O(\lg n)$ height. As such, each leaf can contribute its color only at each of the $O(\lg n)$ ancestors. At each such ancestor it can, at most, be paired with d colors, totaling $O(\lg n \cdot d)$ per leaf. As there are n leaves, this yields an upper-bound of

$$O(d \cdot n \cdot \lg n) \; . \tag{2.13}$$

On the other hand, d is always an upper bound for the number of colors in use in a component. There are $O(n)$ components in an HDT built atop of a tree of size n, yielding an upper-bound of

$$O(d^2 \cdot n) \ . \tag{2.14}$$

Combining these two upper-bounds yields a single upper-bound:

$$O(d \cdot n \cdot \min(d, \lg n)) \ . \tag{2.15}$$

Because of the `extract` and `contract` operations, additional, smaller, copies can reside in memory. These copies are however only created when the size of the copy will be some fraction of the original size, and only one top-to-bottom-chain will exist at any one time, i.e. these copies are exponentially decreasing in size and do not change the bound.

For [1], $d = \max(d_1, d_2)$, because the two input trees are swapped during the algorithm. For our variations, $d = \min(d_1, d_2)$, as no swapping occurs, and the tree with the smallest degree is forced as T_1. □

Corollary 1. *The space usage of the triplet distance calculation algorithm of [1] is $O(n \cdot \min(d, \lg n))$, where $d = d_1$.*

Proof. The proof follows the principle from Lemma 1. As colors are not paired for the triplet distance calculation, there is no d factor in Equation (2.13), making the first bound $O(n \cdot \lg n)$. Equivalently, squaring does not occur in Equation (2.14), and since the number of colors is only determined by T_1, the bound per node is $O(d_1)$. This yields a second bound of $O(d_1 \cdot n)$.

Combining the first and second bounds yields the stated bound. □

We note that using the input tree with the smallest degree d as T_1, the memory usage for the triplet distance calculation algorithm is $O(n \cdot \min(d_1, d_2, \lg n))$.

2.9 Similarities with Other Algorithms

Recall that the algorithm presented in [1] boils down to coloring the trees, building HDTs, counting in these and using extract and contract.

The algorithms presented in [10, 11] for calculating the quartet distance of binary trees in time $O(n \cdot \lg^2 n)$ and $O(n \cdot \lg n)$ respectively, are also based on these concepts. It should be noted that the former, however, does not extract and contract, which accounts for the additional $\lg n$ factor. Note also that both algorithms use counting based on polynomials.

The algorithm presented in [15] for calculating the triplet distance for binary trees in time $O(n \cdot \lg^2 n)$ is also based on these concepts, although excluding extract and contract, accounting for the additional $\lg n$ factor.

Per the above, the algorithm in [1], as well as our improvements, can be seen as a generalization and evolution of the algorithms in [10, 11, 15].

Chapter 3

Implementation

We have implemented the triplet and quartet distance calculation algorithms as presented in [1] as well as our variations (see Section 2.7 and Chapter 5). 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 Section 3.5). 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 [1] 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 [1] has any practical value?

- The previous related implementations for binary trees [15, 16] only use 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 [15, 16] compared to [1] (and [11]). Does the added work of contracting the HDT during recursion outweigh the saved $\lg n$ factor in practice?

- Our asymptotic improvement, as discussed in Section 2.7, 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 [1] has any practical advantages?

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

3.1 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.

3.2 Extract & Contract

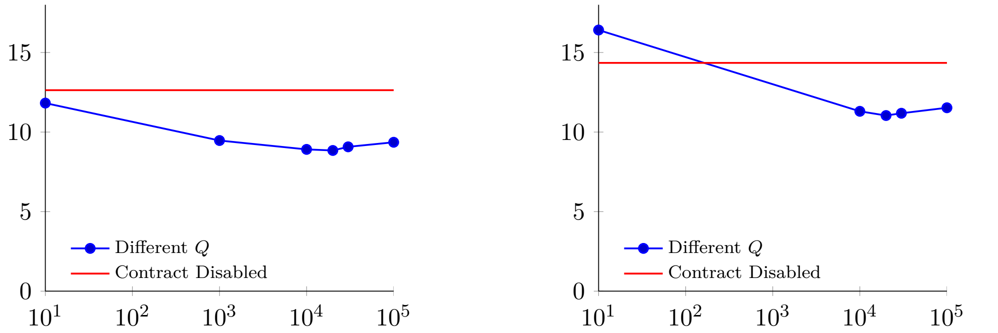
As mentioned in Section 2.4, one of the cornerstones of the algorithm is the two functions `extract` and `contract`. The process of extracting and contracting is three-step. The `extract` function creates a copy of the input HDT with non-marked sub-trees replaced with 0-leaves representing one or more 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 (see Sections 2.3 and 2.4). 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.

3.2.1 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 Section 4.1. The runtimes for this system are depicted in Figure 3.1a. System 2 is a Windows 7 system, with a quad-core 3.3GHz 64-bit Intel Core i5 2500K processor and 16GB of RAM. The runtimes for this system are depicted in Figure 3.1b.

In Figure 3.1a the runtimes are lower with contract enabled compared to having contract disabled, even with Q set to 10. This is not the case in Figure 3.1b. 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



(a) Run on same, Linux, machine as the experiments in Chapters 4 and 5.

(b) Run on a different, Windows, machine than the one used in Chapters 4 and 5.

Figure 3.1: Runtimes with different values of Q as discussed in Section 3.2.1. 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.

even though the system in Figure 3.1b is generally slower than the system in Figure 3.1a, the optimal value of Q seems to be approximately 20,000 in both cases.

As a result of the above, all experiments in Chapters 4 and 5 have been run with Q set to 20,000.

3.2.2 Not Always Contracting the Non-Largest Children

Setting Q quite high, i.e. contracting the largest child infrequently, performed better than setting Q low. As such, an obvious test would be to find out if limiting the frequency at which the non-largest children are contracted yields a similar result.

Per the algorithm in [1], the non-largest children are contracted on every recursion. As an example, if the algorithm visits a binary node where each child has the same size, one will be contracted whereas the other, due to the value of Q, might not.

This does have a few benefits. The HDT of the currently visited node, can be deleted as soon as the largest child is contracted. This is not the case when not contracting the non-largest children, as the HDT is then potentially in use above this node.

In the HDT constructed after the `contract` function, the counters associated with leaves outside of the subtree for which the HDT was created, do not have to be updated when recoloring of these leaves occur. When not always contracting the non-largest children, the non-contracted HDT associated with such a child, is, however, affected by recolorings outside of the subtree. As such, a large number of the counters of this, larger, HDT will have to be updated.

As it is unclear if the approach will benefit the runtime or not, we partially implemented this variation. We soon found it to be significantly slower though, and therefore discarded the changes.

3.3 Debugging

For any non-trivial software implementation, errors are unavoidable. The implementation documented here was no exception, and, as such, we utilized a number of techniques that helped us in the process of resolving such errors.

The first, and perhaps most used, technique was that of automatic testing. In our case we early on implemented a naive algorithm, and generated a large number of random inputs. We then wrote a test system, which ran the two implementations, i.e. the naive algorithms and the algorithms from [1], against each other. Recall, however, that the naive algorithms runs in time $O(n^4)$ and $O(n^5)$ for triplets and quartets respectively. As such the generated trees had to be rather small; the largest trees we tested using the naive algorithm had 80 leaves. As we also had previous implementations available (see Section 1.3), these helped test larger trees. This allowed us to quickly realize if a change in the implementation introduced errors.

In extension of the above, we found it to be too tedious, if not impossible, to analyze an error by hand when running on input larger than around 10 leaves. We therefore wrote a system which continuously generated small random input, compared the output of the two algorithms, and reported an error if the outputs differed. This allowed us to quickly identify small test cases, where an error presented itself.

This left the question of finding the bug in the code. We eased this process somewhat by extending the naive algorithm to also report data relevant to the algorithm described herein.

As explained in Section 2.6, when counting resolved quartet topologies, they occur in three configurations, in [1] named α, β and γ. As these three configurations are counted in different calculations, it would be useful to know how many of each combination should be counted at specific points, during the execution of the implementation.

We extended our naive algorithm to, when outputting all quartets, also output the configuration (α, β or γ) of the quartet, as well as the anchor node of the particular configuration. With this data available, we wrote a script that, given one such list for each tree, outputs the agreeing (i.e. A and E) and disagreeing (i.e. B) combinations, and the labels of their anchor nodes.

With this information, it became possible to easily compare the values at each recursion step to these expected results. If a discrepancy occurred we had found the anchor of a bug-occurrence. With the configuration information we also knew which result each configuration-combination sum should return, easing the debugging further.

Lastly, working with trees can be quite abstract, and being able to visualize a tree can be very useful when debugging. For this reason we wrote functions that allowed us to print trees as files, which could then be visualized by external software. These functions have also been used to output trees for this thesis, for instance Figures 2.5 and 2.7.

In total, using the above mentioned methods, it was possible to pin-point when and where a bug presented itself, easing debugging significantly.

3.4 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 [1], with Q set to 10. This value was chosen arbitrarily at the time. Note that, as stated in Section 3.2, increasing Q to 20,000 decreases the runtime further. Note also that a bug in the `contract` function, affecting the speed, but not the correctness, of this function, was found and fixed after this step. We do not expect this to have influenced the overall trend of the optimizations. If anything, the optimizations would likely have given larger speed boosts. For instance, an improvement of τ seconds with the bug would likely give the same τ seconds improvement without the bug. As the `contract` function became faster by fixing the bug, the overall runtime of the program would also decrease, increasing the speed boost percentage when decreasing the runtime with τ seconds.

The algorithm hints at creating contracted copies of the HDT rather early. To create a contracted copy, as described in Section 2.4, 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 3.1 and 3.2.

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 3.1 and 3.2.

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 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

# Leaves	Initial measure	Opt. 1	Opt. 1 & 2	Opt. 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 3.1: Memory usage in MB with different levels of optimization.

# Leaves	Initial measure	Opt. 1	Opt. 1 & 2	Opt. 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 3.2: Runtime in seconds with different levels of optimization.

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 3.1 and 3.2.

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 3.1 and 3.2. Please note, however, that these measurements were performed on a development machine, not the machine used for the experiments in Chapters 4 and 5.

3.5 Limitations

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

3.5.1 Integer representation

As part of the algorithm, $\binom{n}{3}$ is calculated for triplets and $\binom{n}{4}$ is calculated for quartets. These numbers, in the order of n^3 and n^4, respectively, 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 n^4. 128-bit integers are used for these numbers in the experiments below.

With signed 128-bit integers, $\binom{n}{4}$ will not overflow before $n > 3{,}600{,}000{,}000$. Still being signed 64-bit integers, counters containing n^3 will, however, still overflow at $n \approx 2{,}000{,}000$. As we, in our experiments, have not calculated distances on trees with more than 1,000,000 leaves, we have not changed the n^3 counters to 128-bit integers. Doing so would postpone the problem to $n \approx 3{,}000{,}000{,}000$ where n^2 would overflow the 64-bit signed integers.

Another approach to this problem would be to use an arbitrary precision library. Doing this, there would be no limit to the size of the numbers we could represent, except for the available memory. We have chosen not to do this for two reasons:

- Arbitrary precision libraries are slower than build-in types. See below.
- The available memory will, for all intents and purposes, become a problem before reaching the current 2,000,000 node limit. If not, changing the n^3 counters to 128-bit integers as well, one would definitely run into memory deficiencies on the computers of today, before the next limit of 3,000,000,000 nodes would be reached.

In regards to the arbitrary precision libraries being slower than build-in types, we calculated 3^{50} by means of $3 \cdot 3 \cdot \ldots \cdot 3$, 1,000,000 times. This was done on a 5 year old laptop, running Windows where not otherwise noted. The runtimes where:

$\approx$ 150 ms with `long long` (i.e. 64-bit integers).
$\approx$ 210 ms with `__int128` (i.e. 128-bit integers).
$\approx$ 1740 ms with `mpz_class` usage of `gmp`[1] under Arch Linux.
$\approx$ 11300 ms with a low-level (`mpn`) usage of `mini-gmp`.

From this experiment we conclude that calculations will be at least 8 times slower using an arbitrary precision library than using build-in types.

3.5.2 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 input which include a tree consisting

[1] The GNU Multiple Precision Arithmetic Library. For details about `gmp` and `mini-gmp` see `gmplib.org`.

of a very long chain, we have experimentally found that the program fails when $n \approx 4{,}000$ on Windows and $n \approx 48{,}000$ on Fedora Linux.

The stack overflow already happens when parsing the tree. Even if the parser was rewritten to have less recursive calls (or be entirely iterative), recursion is still used in the main-algorithm as presented in Chapter 2. For this reason, even with a changed parser, there would be a limit to the height of the tree. This limit could, however, be at a larger n. While one could likely get around this problem as well, as most trees of height even just 4,000 will likely be very large in practice, we have chosen not to change our implementation.

Chapter 4

Experiments

Using our implementation of the algorithms presented in [1], and the variation presented in Section 2.7, we have performed a number of experiments presented in Section 4.3.

4.1 Setup

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

Runtime-values are averages of three runs, measured externally. This results in a slight startup overhead, but gives a good indication of the actual wall-time runtime of our implementation. All runtimes depicted in this chapter, as well as more input variations, are presented in tabular form in Appendix D.

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.

All plots have a logarithmic x-axis, but only some have a logarithmic y-axis. The non-logarithmic y-axis better illustrates the actual runtime or memory usage, whereas the logarithmic y-axis can sometimes better illustrate growth and ease comparison for small input.

Memory usage is, as noted previously, based on polling 10 times per second. The reported numbers are the peak values.

Additionally, we have compared our implementation to a number of previous implementations for both triplet and quartet distance calculation (see Section 1.3). These are presented in Section 4.3.6.

4.2 Test Input

Since we are primarily interested in the scalability of our implementation we have only performed tests on randomly generated data. We utilize three types of trees: *Fully balanced trees*, *75% left-biased trees* and *99% left-biased trees.*

Fully balanced trees are, as the name implies, perfectly balanced, i.e. all leaves are at the same level in the tree (or as close to this as the number of leaves allow). 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.

We can furthermore describe trees as *random* and input as *leaf moved*. Random describes the situation where all leaf-labels have been randomly permuted. Two random trees will likely have a very large distance. 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.

Furthermore we define trees as binary or with a specific d. Binary are equivalent to $d = 2$. A specific d specifies the approximate degree in all nodes, but, especially on small input, this is not a hard guarantee. For instance on 99% left-biased trees with $n = 100$, 99% of the leaves (i.e. 99 leaves) go into the first subtree, and the remaining 1% (i.e. 1 leaf) goes into the remaining subtree(s). No matter the d, there will for this node clearly only be one extra child, yielding a binary node. For larger n and other types of trees this discrepancy gets smaller.

4.3 Results

In this section we present the results from having run our implementation variations of the quartet distance calculation and our implementation of the triplet distance calculation on a number of inputs. We also compare our implementation to the implementations running in time $O(n \cdot \lg^2 n)$ [15, 16] and $O(n^{2.688})$ [13]. The comparisons between quartet distance calculation algorithms, have only been performed on input of up to 10,000 leaves.

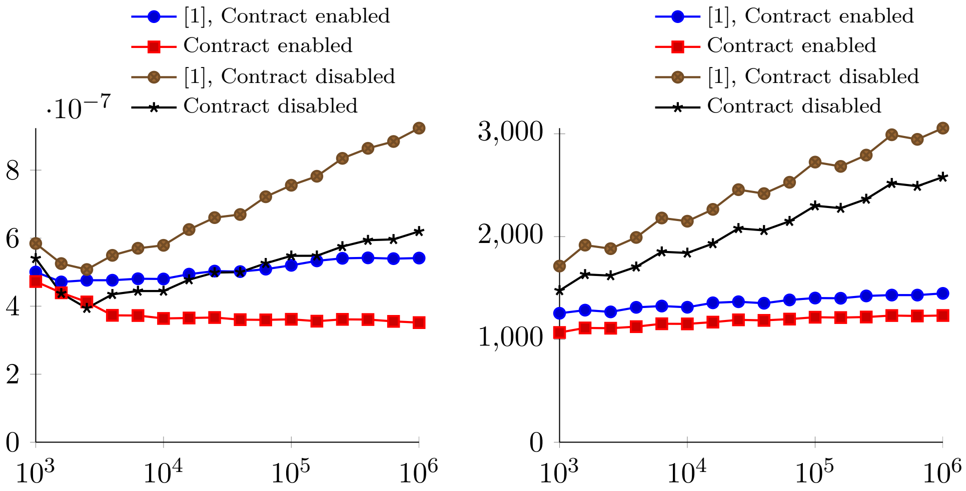
4.3.1 Quartet Distance, Binary Trees

From Figure 4.1a 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 Figure 4.1b, 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” [17].

In all observed cases our variation is faster in practice than the algorithm in [1].

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.

From Figures 4.2a and 4.2c 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.



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

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

Figure 4.1: Quartet distance calculation. Random balanced binary tree against random balanced binary tree. The x axis denotes number of leaves.

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.

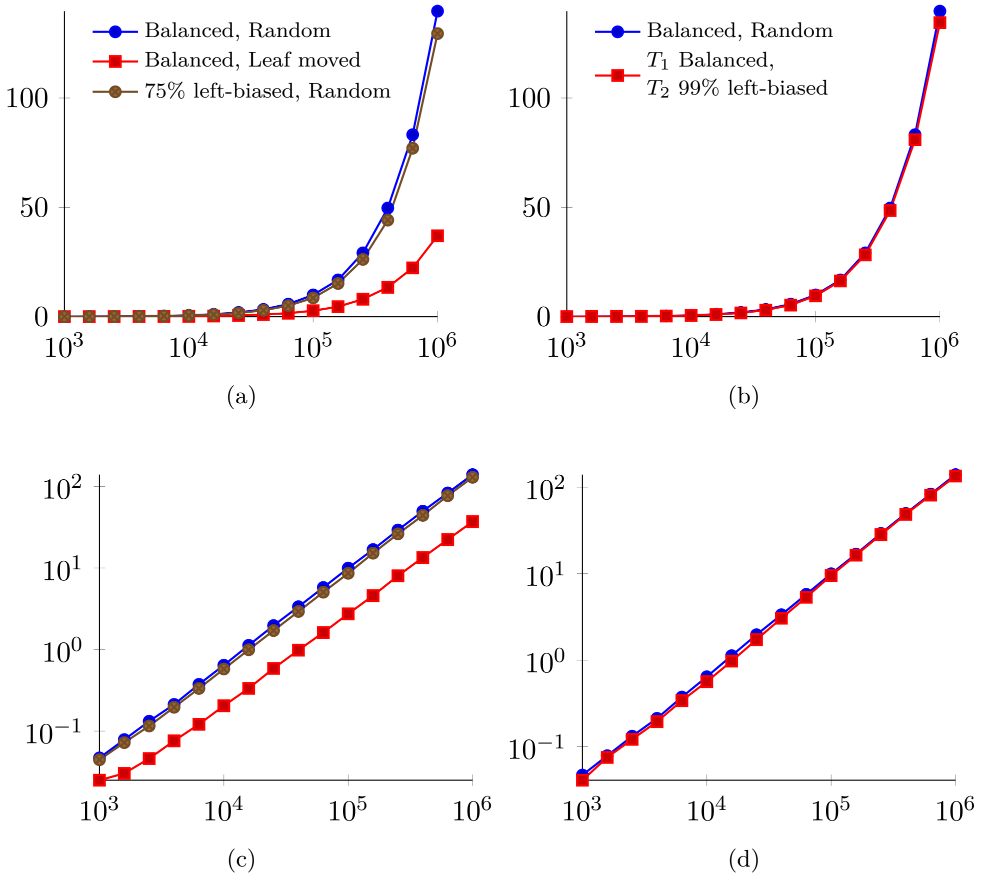
From Figures 4.2b and 4.2d we observe that with T_1 being fully balanced, the type of T_2 does not seem to have a significant influence on the runtime.

In Figure 4.3 we notice something new: Contract is not always a good thing. In fact, on input that is 99% leaf biased, with contract enabled it takes slightly more than five and a half minutes to process 1,000,000 nodes. With contract disabled, however, the same input is processed in slightly less than two minutes.

The runtime practically reverts to the case of two balanced random trees when setting T_1 to a completely balanced tree and only letting T_2 be 99% left-biased. In that case, the algorithm ticks in at approximately two and a quarter minutes.

A possible explanation for the algorithm with contract disabled being fast in the 99% left-biased case is the following. Assume for simplicity that the algorithm with contract disabled is run on input consisting of a chain, i.e. each internal node has two children where one is a leaf. At each internal node only one leaf is recolored, and it thus takes $O(\lg n)$ time to update the HDT. There are $O(n)$ internal nodes and the runtime thus becomes $O(n \cdot \lg n)$ on this input. While 99% left-biased is not a chain per-say, we believe it to be similar enough to a chain for it to explain these runtimes.

However, that the algorithm, with contract enabled, performs significantly worse with 1,000,000 leaves on 99% left-biased input, than when the input is fully balanced, is quite a puzzle. Something happens around the 100,000 leaves mark, where the runtime suddenly jumps with contract enabled. After the sudden



(a)

(b)

(c)

(d)

Figure 4.2: 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).

jump, the trend of the runtime continues as before. Having contract enabled is faster for trees of size 100,000 leaves or less. We can offer no explanation for this behavior.

We observe that the difference between the two sets of input is whether or not T_1 is a balanced tree. Since this is the primary difference between the two cases, we conclude that the structure of T_1 affects the runtime.

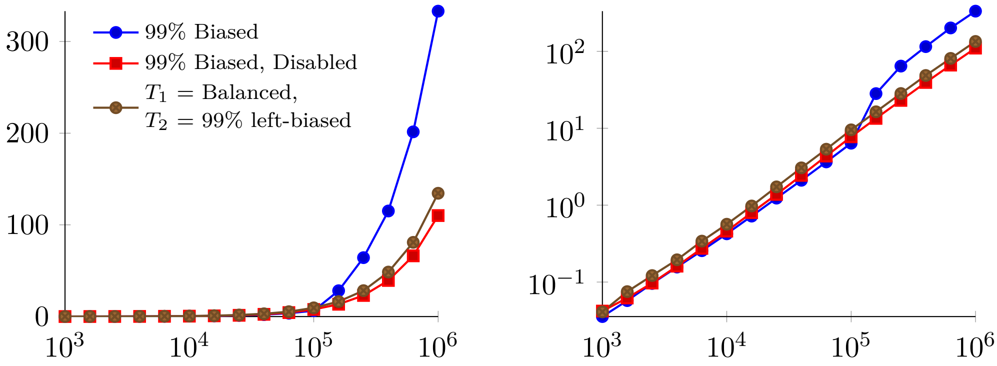


Figure 4.3: Quartet distance calculation. All trees are random, binary and the algorithm is our variation. “Disabled” denotes contract being disabled, if nothing is denoted, contract is enabled. The x and y axis are number of leaves and runtime in seconds respectively. Both plots depict the same data and the legend applies to both.

4.3.2 Quartet Distance, Arbitrary Degree Trees

The asymptotic analysis suggests that a higher value of d implies a longer runtime. As can be observed from Figure 4.4a this is not always the case. The runtime is actually faster with $d = 16$ than with $d = 2$. On input with $d = 16$, the quartet distance for 1,000,000 leaves can be calculated in slightly over a minute.

As can be seen from Figure 4.4b, however, this does not generalize indefinitly. With $d = 128$ the runtime is still slightly faster than with $d = 2$, but with $d = 256$ this does not repeat, and the runtimes gets continuously slower. For larger values of d, we do not have data for up to 1,000,000 leaves. This is due to the memory consumption (see Lemma 1 in Section 2.7).

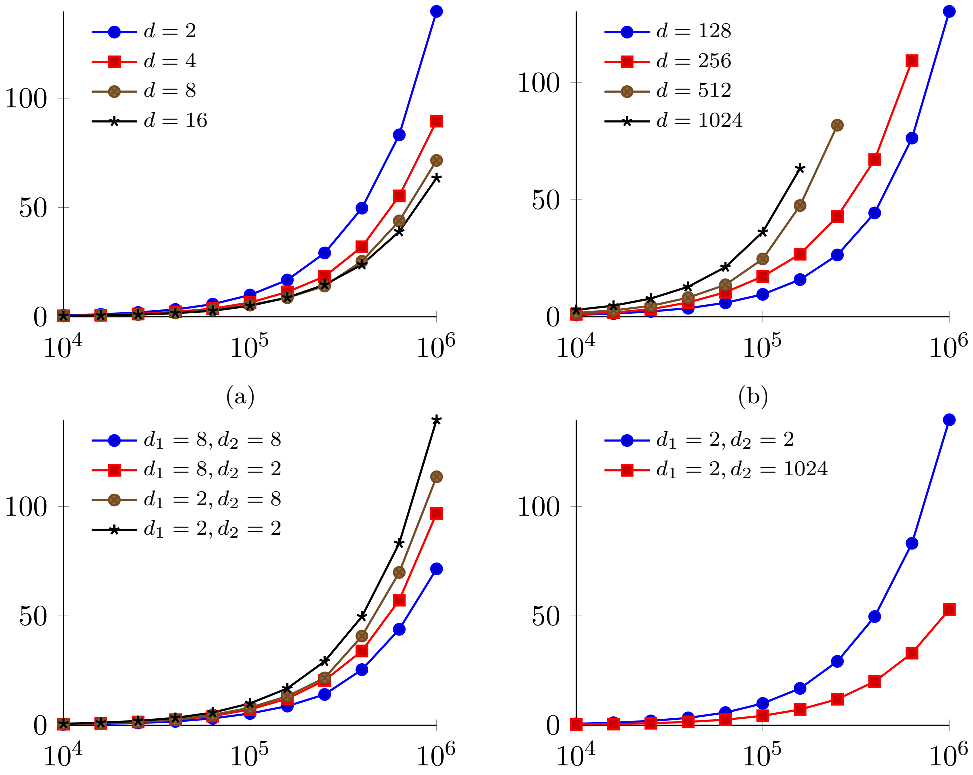
From Figures 4.4c and 4.4d we observe that both the degree of T_1 and of T_2 seems to have an effect on the runtime.

It makes sense that the value of d_1 affects the runtime. A larger value of d_1 implies fewer recolorings. However, a larger value for d_1 also increases the asymptotic runtime.

The value of d_2, however, also seems to change the runtime. One possible explanation for this, is that with a higher degree, there are fewer internal nodes. As such, the HDT has fewer leaves, and the trees that are extracted and contracted are also smaller.

Note that the $d_1 = 8$, $d_2 = 2$ case in Figure 4.4c has been programatically forced to use the larger degree tree as T_1 to facilitate this experiment. Under normal circumstances the implementation will automatically choose the tree with the smallest value of d as T_1 to guarantee the claimed runtime.

Depicted in Figure 4.5 is our variation versus the unmodified algorithm from [1]. Recall that our variation runs in time $O(\min(d_1, d_2) \cdot n \cdot \lg n)$ whereas the algorithm from [1] runs in time $O(\max(d_1, d_2) \cdot n \cdot \lg n)$. This indicates that on input where the degree of one tree is larger than the other, our variation should be faster. This also holds true in practice for large enough difference between d_1 and d_2. As depicted in Figure 4.5 with one tree having degree 2 and



(a) (b) (c) (d)

Figure 4.4: Quartet distance calculation. Varying values for d of the input trees to the algorithm. Trees are random and fully balanced. The x and y axis are number of leaves and runtime in seconds respectively.

the other degree 1024, the unmodified algorithm uses approximately 42 seconds processing 100,000 leaves, whereas our variation processes the same input in approximately 4 seconds.

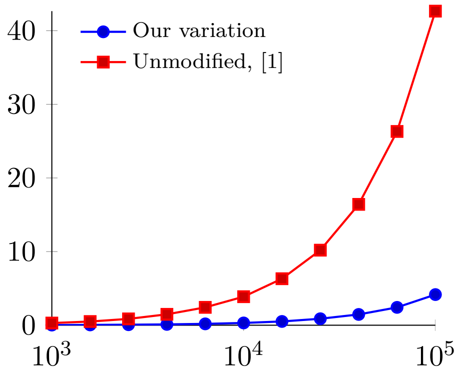


Figure 4.5: Quartet distance calculation. Input has $d_1 = 2$ and $d_2 = 1024$. In both cases contract is enabled and the trees are random and balanced. The x and y axis are number of leaves and runtime in seconds respectively.

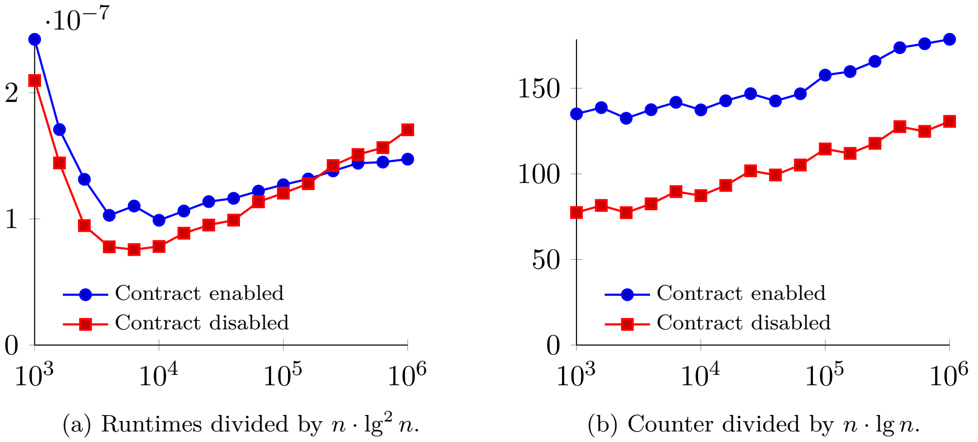
4.3.3 Triplet Distance, Binary Trees

From Figure 4.6a we observe that contracting actually slows things down when n is less than approximately 250,000 on random balanced binary trees. After this, contracting was observed to improve the actual runtime. Neither with contract enabled nor disabled does it look like $O(n \cdot \lg^2 n)$ as was the case with the quartet calculation. The counter value does not seem to resemble $O(n \cdot \lg n)$ either, but as the algorithm is essentially the same as the quartet distance calculation algorithm, although with less counters, we must conclude that it will increase until reaching some maximum constant.

With contract enabled it takes less than a minute to calculate the triplet distance for 1,000,000 leaves on this type of input.

From Figures 4.7a to 4.7d we observe that the story from the quartets repeats itself, i.e. leaf-moved input is faster to process than completely random input. Equally, 99% left-biased trees are processed slower with contract enabled than with contract disabled on 1,000,000 leaves. Again, something happens around the 100,000 leaves mark, for which we can offer no explanation.

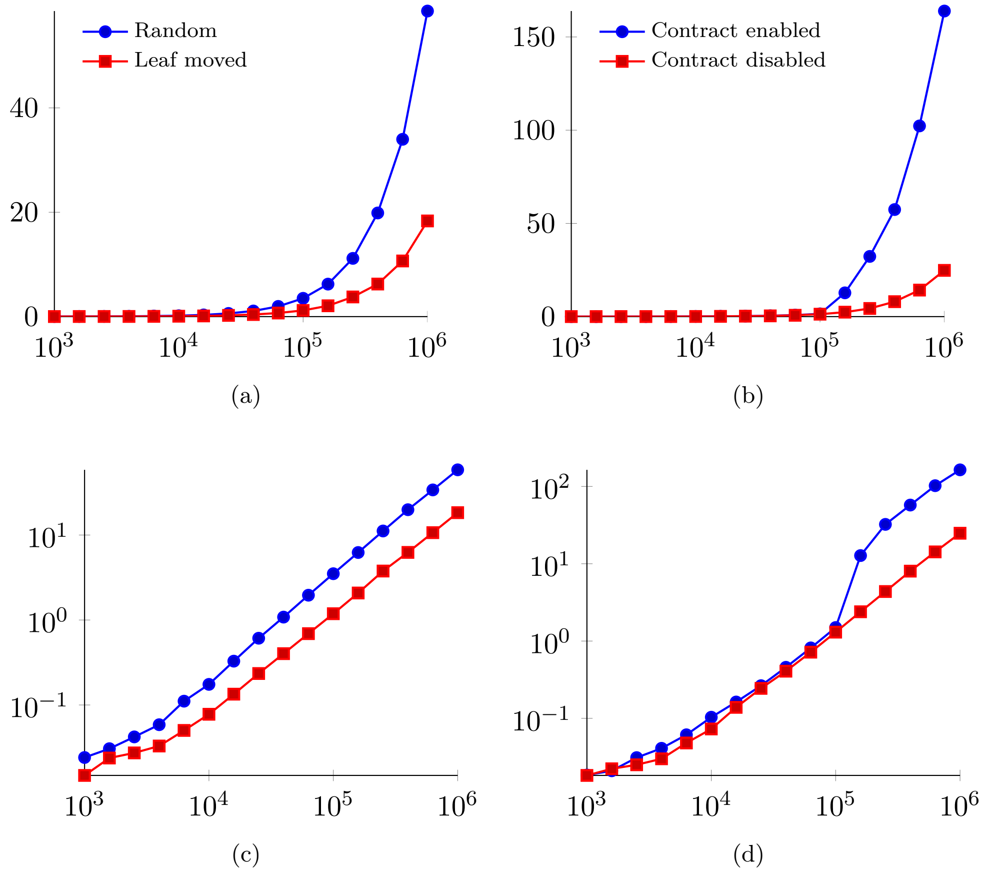
Additionally, we note that for leaf-moved input, contracting is not beneficial, see Appendix D.



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

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

Figure 4.6: Triplet distance calculation. Random balanced binary tree against random balanced binary tree. The x and y axis are number of leaves and runtime in seconds respectively.



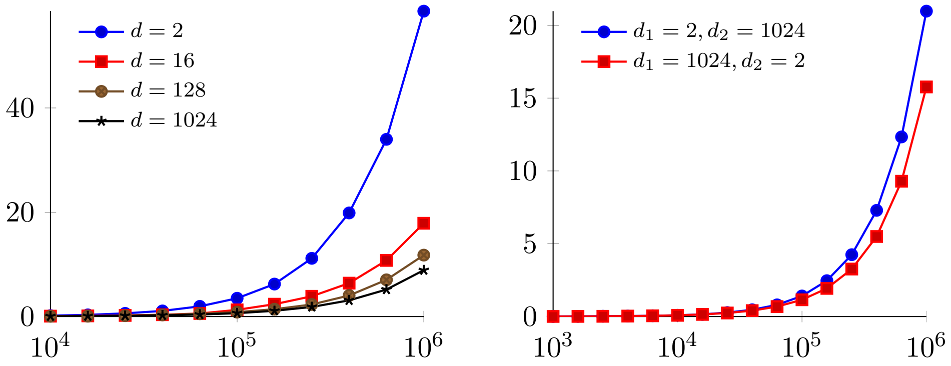
(a)

(b)

(c)

(d)

Figure 4.7: Triplet distance calculation. The x and y axis are number of leaves and runtime in seconds respectively. Input in (a) is binary and balanced and is run on our variation with contract enabled. Input in (b) is 99% left-biased, random and binary, and is run on our variation. 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).



(a)

(b)

Figure 4.8: Triplet distance calculation. Varying values of d for the trees given as input to the algorithm. Trees are random and fully balanced. The x and y axis are number of leaves and runtime in seconds respectively.

4.3.4 Triplet Distance, Arbitrary Degree Trees

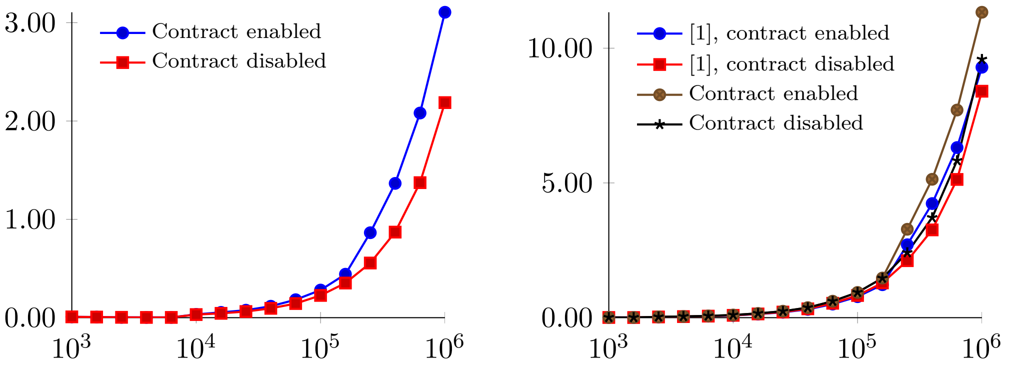
Unlike the runtime for the quartet distance calculation, the runtime for the triplet distance calculation, when running on input with large degree, gets continuously smaller (see Figure 4.8a). On a balanced input with trees of size 1,000,000, we observe that it takes ≈ 59 seconds when $d = 2$, ≈ 18 seconds when $d = 16$, ≈ 12 seconds when $d = 128$, and ≈ 9 seconds when $d = 1024$. This is consistent with the runtime of the triplet distance calculation not depending on d, and the runtimes of the quartet distance calculation decreasing, up to a point, for larger values of d. The results in Figure 4.8b are also consistent with the results from the quartet distance calculation, i.e. both d_1 and d_2 affect the runtime of the implementation. Specifically both cases in Figure 4.8b are faster than the case when $d = 2$ in Figure 4.8a.

4.3.5 Memory Usage

In Figures 4.9a and 4.9b the memory usage of the triplet distance calculation algorithm and the quartet distance calculation algorithm, respectively, is depicted. Both figures depict memory usage when running on random balanced binary trees.

Quite a lot of memory is consumed, although not more than some desktops or even laptops would be able to handle, even at 1,000,000 leaves, at least for binary trees. The quartet distance calculation algorithms use more memory than the triplet distance calculation algorithm because of significantly more counters. Having contract enabled increases the memory usage because of several HDTs. Our quartet distance calculation algorithm generally uses more memory than the algorithm in [1] because of more counters still. The difference for binary trees is approximately 20%.

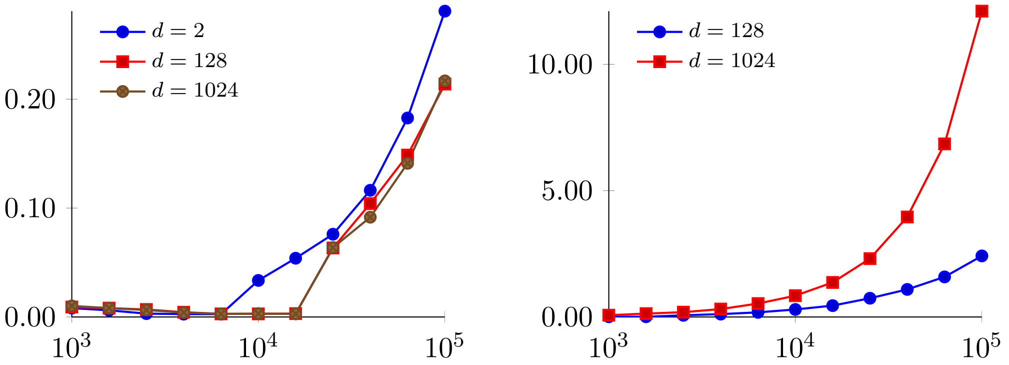
An exception to the latter is the case where the two values of d are different, see Figure 4.11b. As for the runtime (see Figure 4.5), our variation benefits from $d = \min(d_1, d_2)$ compared to $d = \max(d_1, d_2)$ for the unmodified algorithm.



(a) Triplet distance calculation.

(b) Quartet distance calculation.

Figure 4.9: Memory usage in GB plotted against the number of leaves, when running on random, binary and balanced trees using our variation for quartets.



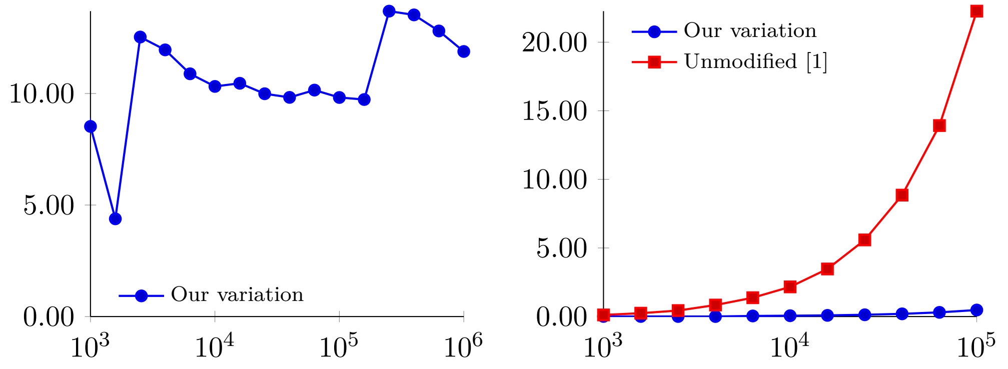
(a) Triplet distance calculation.

(b) Quartet distance calculation.

Figure 4.10: Memory usage in GB plotted against the number of leaves, when running on random balanced trees of large degree. Contract enabled, and using our variation for quartets. Note that this figure only depicts memory usage of up to 100,000 leaves, whereas Figure 4.9 depicts memory usage of up to 1,000,000 leaves.

With this in mind one can, to some degree, trade time for space. With input consisting of two trees with the same degree, one can reduce the memory usage and increase the runtime by using the algorithm in [1] with contract disabled instead of our variation with contract enabled. When the input consists of two trees with different degree, and where one of the degrees is large and the other relatively smaller, one can get the best of both worlds by using our variation with contract enabled.

Per Lemma 1, the memory usage of the quartet distance calculation will increase with d. This is, per Corollary 1, only the case for the triplet distance calculation for $d \leq \lg n$. That the memory usage does not depend on d, when $d > \lg n$, is observed to hold in practice, see Figure 4.10. The memory usage for triplets, with 100,000 leaves is observed to be between 220 and 290 MB for d being 2, 128 and 1024. Interestingly, it is the $d = 2$ case which uses 290 MB.



(a) Memory usage divided by n to indicate if memory usage is in fact linear with a fixed small d.

(b) Memory usage in GB for the quartet distance calculation on input with $d_1 = 2$ and $d_2 = 1024$.

Figure 4.11: Memory usage plotted against the number of leaves, for the quartet distance calculation when input trees are random, balanced and contract is enabled. In (b), curiously, the unmodified algorithm uses more memory than the modified algorithm does in the $d_1 = d_2 = 1024$ case (from Figure 4.10b).

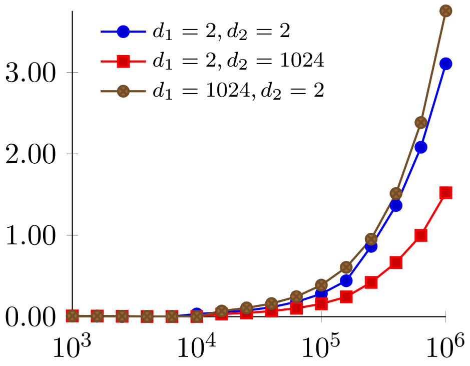


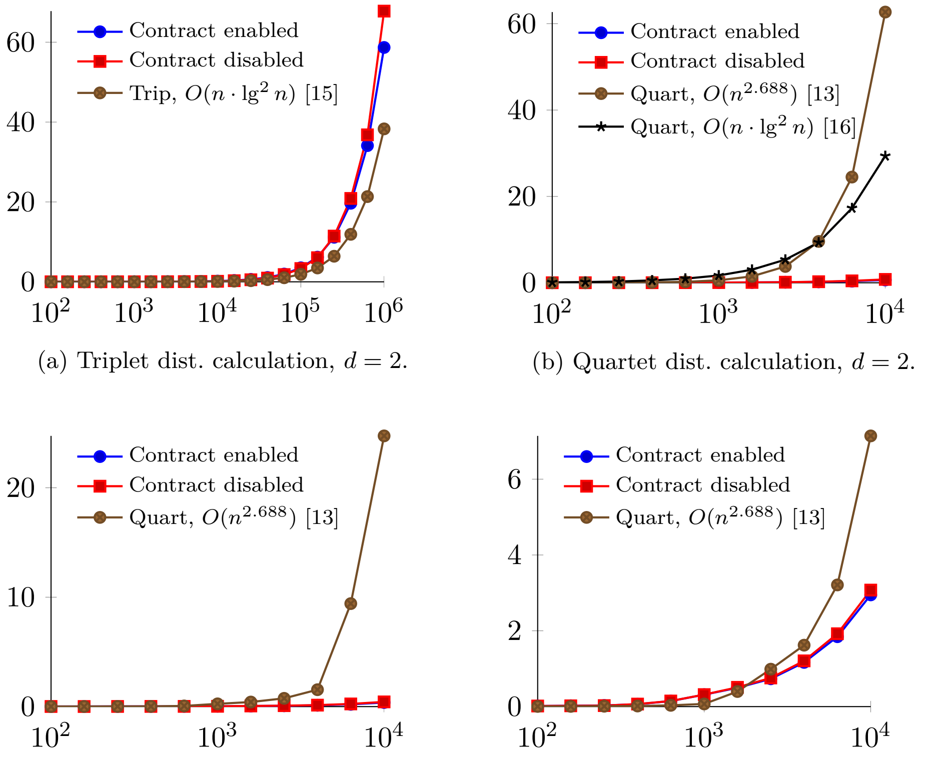
Figure 4.12: Memory usage in GB plotted against the number of leaves, for the triplet distance calculation with different size d as T_1 and T_2. Trees are random and balanced.

This is in contrast to what is predicted by Corollary 1. A possible explanation could be the number of nodes in T_2, decreasing for larger values of d_2.

For quartets, with 100,000 leaves, the memory usage is observed to be ≈ 0.95, 2.4 and 12.1 GB for d being 2, 128 and 1024, respectively.

From Lemma 1 we see that the quartet distance calculation should, for a small fixed d, be linear in the size of the input. This is illustrated in Figure 4.11a. Although the line is less than stable, there does not seem to be a correlation between a larger input and a larger value of memory once divided by the size of the input. Thus, the memory usage, for a small fixed d, does in fact seem to be linear in the size of the input.

From Figure 4.12, we see that the memory usage of the triplet distance calculation increases with a larger degree for T_1 when d_2 is fixed. This is as



(a) Triplet dist. calculation, $d = 2$.

(b) Quartet dist. calculation, $d = 2$.

(c) Quartet dist. calculation, $d = 8$.

(d) Quartet dist. calculation, $d = 1024$.

Figure 4.13: Comparison to other implementations. Input trees are random balanced trees. The x and y axis are number of leaves and runtime in seconds respectively.

predicted by Corollary 1. However, as before, the value of d_2 also has an effect in practice. We again offer the explanation of a decreasing number of nodes in T_2 for larger values of d_2.

4.3.6 Comparison to Other Implementations

We now compare the implementation of our variation to previous implementations. Runtimes are depicted in Figure 4.13 and memory usage is depicted in Figure 4.14.

In Figure 4.13a we observe that the triplet distance calculation algorithm for random binary trees running in time $O(n \cdot \lg^2 n)$, [15], is faster in practice than our implementation of an $O(n \cdot \lg n)$ algorithm. This also holds true for our $O(n \cdot \lg^2 n)$ algorithm, i.e. with contract disabled. Some of this might be contributable to the limitation of [15], i.e. it only handles binary trees. As an additional note, [15] uses 32-bit integers and thus starts to overflow at $n \approx 3{,}000$. Furthermore, we note that on leaf-moved input, our implementation, with contract disabled, is actually faster than [15], see Appendix D.

Comparing the quartet distance calculation algorithms, however, our implementation, both with and without contract enabled, is faster than the other implementations tested. While this is not surprising for the algorithm running

# Leaves	Quart, $O(n^{2.688})$	Contract enabled	Percentage
$1.0 \cdot 10^2$	0.005	0.016	28.64%
$1.6 \cdot 10^2$	0.006	0.023	25.48%
$2.5 \cdot 10^2$	0.010	0.030	32.75%
$4.0 \cdot 10^2$	0.016	0.062	25.28%
$6.3 \cdot 10^2$	0.033	0.146	22.85%
$1.0 \cdot 10^3$	0.070	0.315	22.22%
$1.6 \cdot 10^3$	0.398	0.496	80.23%
$2.5 \cdot 10^3$	0.987	0.734	134.54%
$4.0 \cdot 10^3$	1.620	1.166	138.92%
$6.3 \cdot 10^3$	3.210	1.844	174.08%
$1.0 \cdot 10^4$	7.141	2.947	242.29%

Table 4.1: Runtime comparison between our implementation with contract enabled and [13]. Percentage denotes time spent by [13] compared to our implementation. Input is balanced trees with $d = 1024$.

in time $O(n^{2.688})$ [13], it comes as a welcome surprise that the story from the triplet case does not repeat itself in regards to the $O(n \cdot \lg^2 n)$ time algorithm [16]. An explanation for the latter might be that the quartet distance calculation algorithm implemented in [16] uses a counting scheme based on very general polynomials. This, however, is pure speculation on our part.

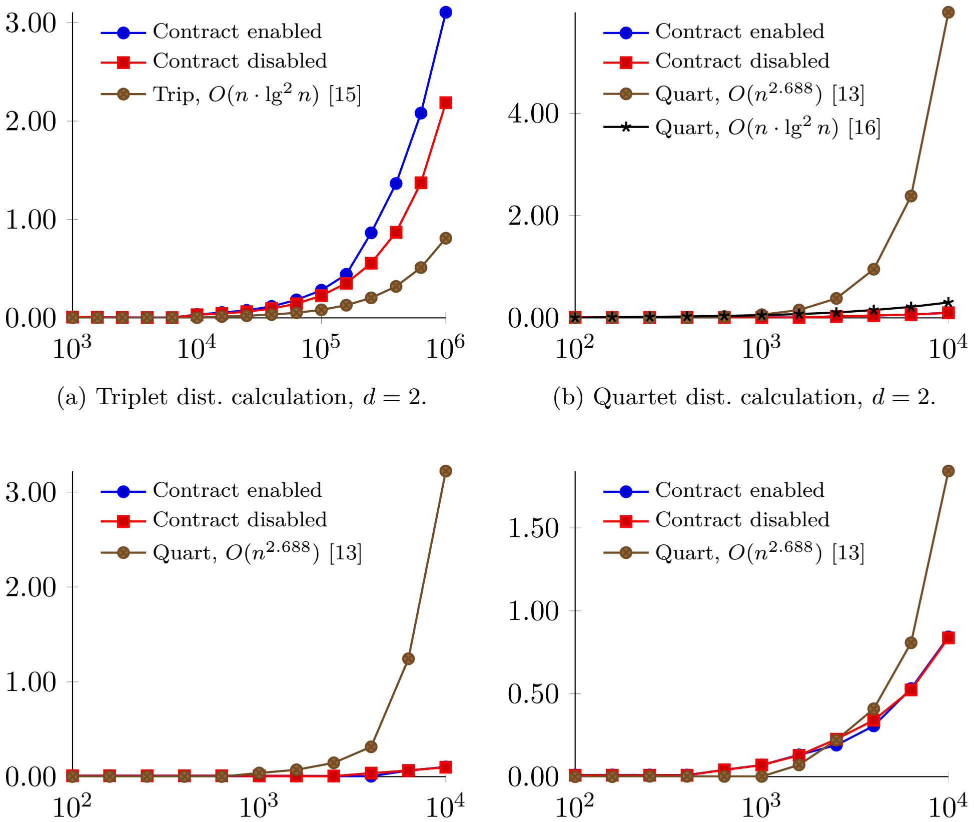
Both our implementation and the implementation from [13] are faster with $d = 8$ than with $d = 2$. Our implementation is, however, in both cases faster than the implementation from [13], see Figures 4.13b and 4.13c.

As can be observed from Figure 4.13d, our implementation is slower than implementation in [13] for trees with less than approximately 2,000 leaves when $d = 1024$. In [13] the authors observed that the runtime of their implementation was, in practice, closer to $O(n^2)$ than the theoretical $O(n^{2.688})$. This means that for input of size 2^{10}, the expected runtime of their implementation, for some constant c, is $c \cdot 2^{20}$. The runtime of our implementation running in time $O(\min(d_1, d_2) \cdot n \cdot \lg n)$, for some constant c', becomes $c' \cdot 2^{10} \cdot 2^{10} \cdot \lg 2^{10} = c' \cdot 10 \cdot 2^{20}$. These numbers give a possible explanation as to why our implementation is slower for small trees with large d, compared to [13].

As can be seen from Table 4.1 our implementation, however, catches up to [13] and becomes increasingly faster for large input. This is also in line with the above.

The memory usage of the different algorithms, compared to ours, is depicted in Figure 4.14. As can be seen from Figure 4.14a, our implementation of the triplet distance calculation, both with contract enabled and disabled, uses more memory than the implementation in [15]. An explanation for this could be the use of 32-bit integers in [15], whereas we utilize 64-bit integers.

From Figures 4.14b to 4.14d we generally observe the same trend for the memory usage as we did for the runtime. One exception to this, is the memory usage of the implementation from [16]. The memory usage of this implemen-



(a) Triplet dist. calculation, $d = 2$.

(b) Quartet dist. calculation, $d = 2$.

(c) Quartet dist. calculation, $d = 8$.

(d) Quartet dist. calculation, $d = 1024$.

Figure 4.14: Comparison to other implementations. Input trees are random balanced trees. The x and y axis are number of leaves and memory usage in GB respectively.

tation increases significantly slower than the runtime. A purely speculative explanation for this is the use of a counting scheme based on very general polynomials, which do not require much space.

4.4 Summary

In Chapter 3 a number of questions were posed. Using the results from this chapter, we are now in a position to answer these questions.

- The first question posed was, whether or not the large number of counters made the algorithm impractical.

  It is certainly true that we need to calculate many variables, and that the constant is very large. As can be seen from Figure 4.1 and Sections 4.3.3 and 4.3.4, however, the usage of the algorithms presented in [1] do, in fact, seem to have practical value.

- The second question posed was, whether or not the overhead of performing the contract operation, would outweigh the benefits of contracting.

As we saw in, e.g., Figure 4.1, it is sometimes beneficial to perform the contract operation. However, as we saw in Figure 4.3, this is not always the case. As such, the benefits of contract must be weighed on a case by case basis.

- The final question posed was, if our variation, as presented in Section 2.7, had any advantage over the algorithm presented in [1].

  As we saw in, e.g., Figures 4.1 and 4.5, our variation is indeed faster. Although, as we saw in Figure 4.9b, this comes at an added cost of memory (approximately 20% for binary trees). However, as we saw in Figure 4.11b, for the right input, this can be more than saved by the memory usage depending on $\min(d_1, d_2)$ instead of $\max(d_1, d_2)$.

Additionally, we observed that the structure (i.e. how balanced the tree is) and degree of the tree operating as T_1 is important for the runtime of the algorithm, and that the degree of T_1 influences the memory usage (see, e.g., Figures 4.3, 4.4a, 4.4b and 4.10b).

We do note that the triplet distance calculation is different from the quartet distance calculation in this regard, as the degree of T_1 does not change the asymptotic runtime, but only the asymptotic memory usage of the algorithm.

We have, however, observed the degree of T_1 to, in practice, affect both the runtime and the memory usage (see Figures 4.8b and 4.12). In practice, the runtime decreases for larger values of d_1. For the memory usage, different values for d_1 only have an effect when $d_1 \leq \lg n$. This is due to the asymptotic memory usage being $O(n \cdot \min(d_1, \lg n))$. As such, for $d_1 > \lg n$, the value of d_1 no longer affects the memory usage (see Figure 4.10a). Thus, in the triplet distance calculation, the tree used as T_1 should be the tree with the largest degree, as long as the needed amount of memory is available.

In addition, the degree of T_2 was seen to influence the runtime (see Figure 4.4d), whereas the structure of T_2 does not seem to be relevant in this regard (see Figures 4.2b and 4.2d).

Comparing our implementation to previous implementations, we have found that our implementation is very competitive in regards to both runtime and memory usage. For triplets, our implementation only appears to be a small constant factor slower than the implementation in [15]. It should be noted that our implementation, in contrast to the implementation in [15], also operates on trees of arbitrary degree and does not overflow at $n \approx 3{,}000$.

For quartets, we observe that for some types and sizes of input, the implementation in [13] is faster than our implementation. Our implementation, however, generally outperforms the implementations in [13, 16].

Chapter 5

Calculating E Instead of B

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 used? Additionally could the problem be solved faster? As can be seen from Figures 5.1a and 5.1b, the majority of the time spent by the implementation is spent on counting in the HDT.

Recall from Figure 1.3 and Section 1.4 that given either A or B and any of the other values, both the triplet distance and the quartet distance can be calculated in linear time (see Section 2.1).

A solution that addresses both of the above mentioned problems, could thus be to find E instead of B when calculating the quartet distance.

To calculate B, [1] dictates the maintenance of sums for 10 configuration combinations. To remove the need for swapping the two trees, our variation adds 4 configuration combinations.

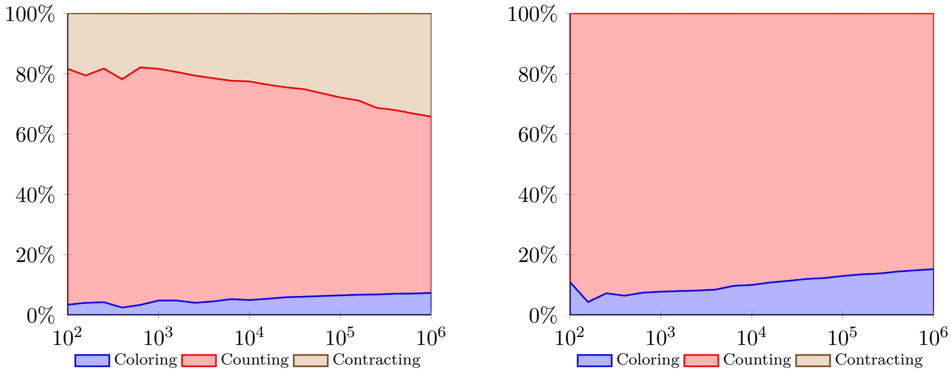
A rudimentary enumeration of the cases used when calculating E, shows that we would need to maintain only 5 different configuration combinations (see Figure 5.2). This indicates that fewer counters could be required, thus using less memory and possibly calculating the result faster.

From the rudimentary enumeration of the cases this approach seemed promising and we thus filled out the details (available in Appendix C) and did an implementation. Note that this introduces a new aggregate descriptor, $\triangle$. This descriptor is to $\square$ as $\square$ is to $\bullet$, see Section 2.5.

Our variation, calculating B, requires the handling of 14 configuration combinations, totaling 92 sums. These sums require up to $5d^2 + 48d + 8$ counters.

In comparison, calculating E instead, only requires the handling of 5 configuration combinations, totaling 21 sums, requiring up to $1d^2 + 12d + 12$ counters (of which $1d^2 + 1d$ also appeared in the B calculation).

Calculating E instead of B affects neither the asymptotic runtime, nor the asymptotic memory usage.



(a) Contract enabled.

(b) Contract disabled.

Figure 5.1: Time spent in different parts of the algorithm. Both figures depict quartet distance on our first variation of the algorithm. Trees are random, balanced and binary.

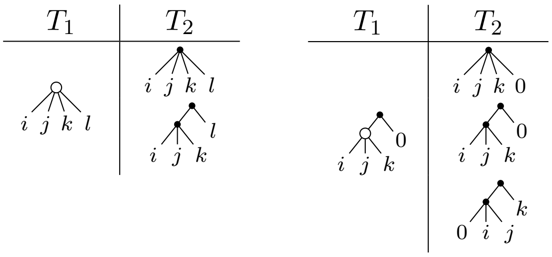
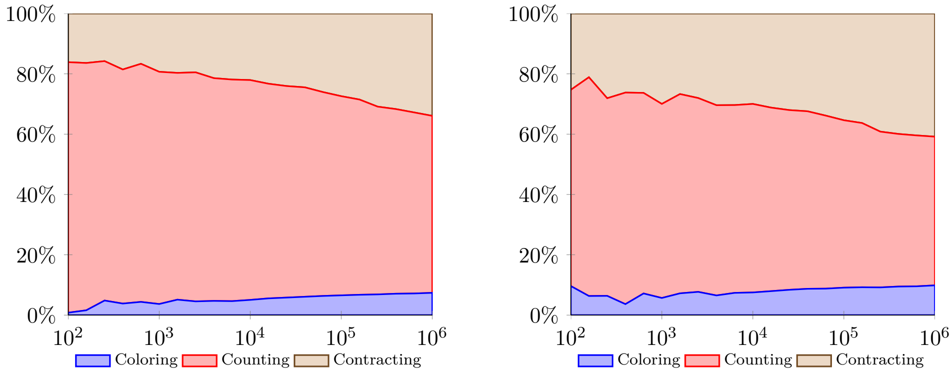


Figure 5.2: Configuration combinations needed when calculating E instead of B.

5.1 Results when Calculating E

As can be seen from Figure 5.3, the amount of time spent in the counting part of the algorithm is relatively smaller when calculating E instead of B. Additionally we note from Figure 5.4 that the actual runtime, when calculating E instead of B, is also less. This is the case both for binary trees and with larger values of d. We further note that the memory usage (see Figure 5.5), when calculating E instead of B, is significantly lower. This allows us to calculate the quartet distance on input consisting of 1,000,000 leaves, when $d = 256$. This was not possible when calculating B, with the amount of memory available on the test system.

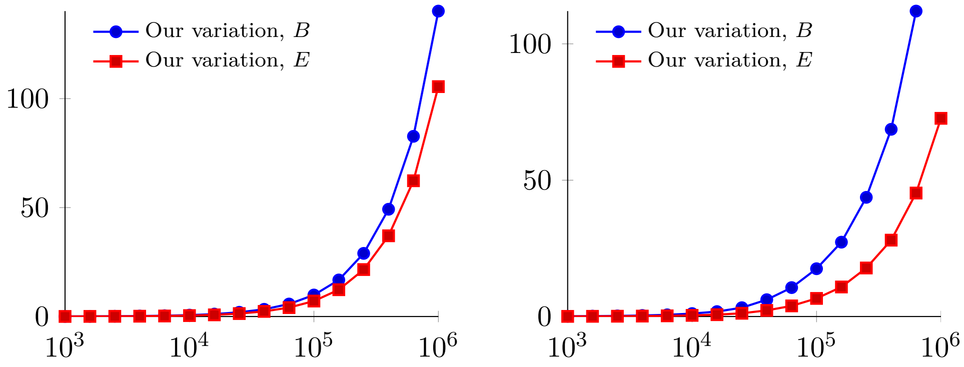
Lastly, from Appendix D we observe that for all tested input, our variation calculating E and with contract disabled, is faster than the algorithm presented in [1] with contract enabled. For some input (for the tested input when $d_1 \geq 16$), our variation calculating E and with contract disabled, is also faster than our variation calculating B with contract enabled. Enabling contract for the E calculation decreases the runtime further. These details indicate that one could write a fast implementation, calculating the quartet distance between two trees of arbitrary degree, without the need for difficult contract-code, by calculating A (with our additions) and E.



(a) Our variation, B.

(b) Our variation, E.

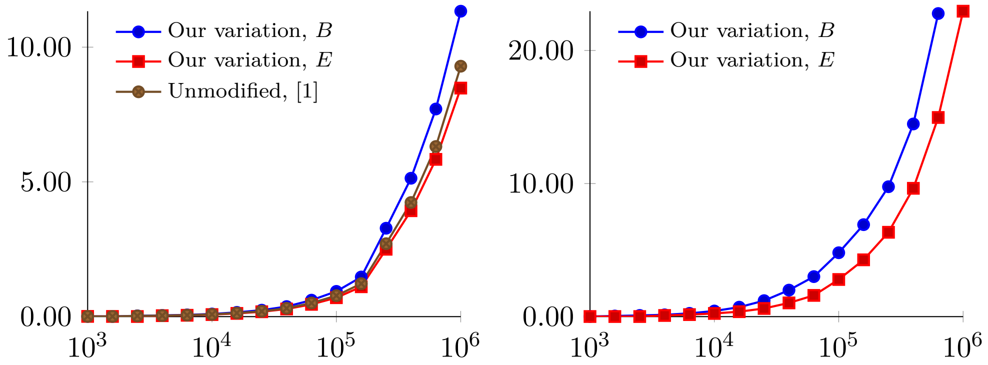
Figure 5.3: Time spent in different parts of the algorithm for calculating B or E. Both figures depict quartet distance calculation with contract enabled. Trees are random, balanced and binary.



(a) Input is binary.

(b) Input has $d = 256$.

Figure 5.4: Runtime in seconds plotted against the number of leaves, when run on random balanced trees with contract enabled.



(a) Input is binary.

(b) Input has $d = 256$.

Figure 5.5: Memory usage in GB plotted against the number of leaves, when run on random balanced trees with contract enabled.

5.2 Summary

As we saw, the number of counters needed was significantly decreased by calculating E instead of B. This did in fact result in a faster runtime (see Figure 5.4), and a smaller memory usage (see Figure 5.5).

In our view, one should, when calculating the quartet distance, always use this variation, i.e. calculating A and E, as it is both faster and uses less memory. Additionally, as this variation requires both less counters and fewer sums, the task of implementing this variation is less daunting. As a side note, by calculating A and E for both the triplet distance calculation and the quartet distance calculation, the combined algorithm is more polished.

As the amount of asymptotic work done, in each HDT node, has not changed, the overall algorithm is still the same. As such, we expect the conclusions regarding the different structures and degrees of the input, as presented in Section 4.4, to remain unchanged.

Chapter 6

Future work

Even with our improvements the memory consumption appears to be the first limiting factor. As such the question of, whether the amount of memory used can be decreased, still remains. Would it be possible, e.g., to decrease the number of counters further than we did by calculating E instead of B? 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 margin of error of a few percent?

Additionally, in our implementation we contract the largest subtree of a given node when the subtree is smaller than a specific fraction of the size of the HDT. This is set for the duration of the program at compile-time. Can this be set dynamically during the running of the program to fine-tune performance for the current state of the program? Alternatively, does another measure than size exist for comparison?

We contract by converting the extracted HDT into the tree it represents, contracting this tree, and constructing a new HDT from the tree. Another question is if contracting can be done directly on the extracted HDT, and if this would result in a smaller overhead when contracting?

Another question in the same ballpark, is whether or not the asymptotic run time can be reduced further. For instance, could the triplet or quartet distance be calculated in linear time instead? Or can the d factor, for quartets, 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, and with a smaller memory footprint, instead? Would it for instance be possible to create an approximation algorithm that produces a correct result within a margin of error of a few percent, while running in linear or sublinear time and space?

Chapter 7

Conclusion

We have implemented and improved upon the algorithm in [1].

One improvement of the quartet distance calculation algorithm reduces the runtime from $O(\max(d_1, d_2) \cdot n \cdot \lg n)$ to $O(\min(d_1, d_2) \cdot n \cdot \lg n)$. The memory usage is also decreased from $O(\max(d_1, d_2) \cdot n \cdot \min(\max(d_1, d_2), \lg n))$ to $O(\min(d_1, d_2) \cdot n \cdot \min(d_1, d_2, \lg n))$. Timewise, the improvement also pays off in practice, even when $d_1 = d_2$, although this is not implied by the asymptotic runtime. For $d_1 = d_2$, the memory usage of the improvement, however, increases by a constant factor. Both with and without this improvement, the algorithm works well in practice.

The quartet distance calculation, as presented in [1], as well as this improvement, calculates the distance by calculating the values A and B.

We have improved the runtime and memory consumption further by instead calculating A and E. This does not change the asymptotic runtime or memory usage, but in practice it has increased the speed and decreased the memory usage, both by a constant factor.

For practical purposes the amount of memory is likely going to be a limiting factor before time-usage is. For binary trees with 1,000,000 leaves as input, the memory consumption of our vaiation of the quartet distance calculation algorithm, calculating A and B, is approximately 11.5GB. The memory usage increases with the degree, and for $d = 256$, the memory usage is more than 22GB for $\approx$ 630,000 leaves. Calculating A and E instead reduces these numbers to approximately 8.5GB and 15GB, respectively.

We are able to calculate the quartet distance for binary trees with 1,000,000 leaves in less than two and a half minutes using our first variation. The second variation reduces this to under two minutes. The triplet distance calculation is faster than this, although an implementation for binary trees which is even faster in practice does exist [15]. This even faster triplet distance calculation implementation, however, overflows at $n \approx$ 3,000.

We have found that contracting the tree can in fact improve the runtime for calculating the quartet distance as well as, for large enough input, the triplet distance. Whether this is beneficial or not, however, depends on the trees given as input.

While the actual runtime, for the quartet distance calculation on binary trees, appears to be $O(n \cdot \lg^2 n)$, a counter-variable, counting the amount of constant work only seems to grow as $O(n \cdot \lg n)$ as the theory predicts.

In conclusion, we have documented the ideas brought forth by Brodal *et al.* [1] to be both feasible and practical, even for very large trees.

Bibliography

Text in parenthesis denotes the parts of the articles that are referenced.

[1] G. S. Brodal, R. Fagerberg, M. Mailund, C. N. S. Pedersen, and A. Sand. Efficient algorithms for computing the triplet and quartet distance between trees of arbitrary degree. In *SODA*, pages 1814–1832. SIAM, 2013.

[2] M. A. Steel and D. Penny. Distributions of tree comparison metrics—some new results (p. 133 first column, last 7 lines). *Systematic Biology*, 42(2): 126–141, 1993.

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

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

[5] M. S. Waterman and T. F. Smith. On the similarity of dendrograms ("nearest neighbor interchange matric", first paragraph). *Journal of Theoretical Biology*, 73(4):789–800, 1978.

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

[7] W. H. E. Day. Optimal algorithms for comparing trees with labeled leaves ("abstract"). *Journal of Classification*, 2(1):7–28, 1985.

[8] D. Bryant, J. Tsang, P. E. Kearney, and M. Li. Computing the quartet distance between evolutionary trees ("introduction"). In *SODA*, pages 285–286. ACM/SIAM, 2000.

[9] B. Dasgupta, X. He, T. Jiang, M. Li, J. Tromp, and L. Zhang. On computing the nearest neighbor interchange distance ("abstract"). In *Proc. DIMACS Workshop on Discrete Problems with Medical Applications*, pages 125–143. Press, 1997.

[10] G. S. Brodal, R. Fagerberg, and C. N. S. Pedersen. Computing the quartet distance between evolutionary trees in time $O(n \log^2 n)$ ("Abstract"). In

Proc. of the 12th International Symposium on Algorithms and Computation, volume 2223 of *Lecture Notes in Computer Science*, pages 731–742. Springer, 2001.

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

[12] 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 (“abstract”). In *Proc. of the 5th Asia-Pacific Bioinformatics Conference (APBC)*, pages 101–110. Imperial College Press, 2007.

[13] 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 (“abstract”, “background”). *Algorithms for Molecular Biology*, 6(1): 15, 2011.

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

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

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

[17] T. Jurkiewicz and K. Mehlhorn. The cost of address translation (“abstract”, “introduction”). In *ALENEX*, pages 148–162. SIAM, 2013.

Appendix A

Counters and Sums from [SODA 2013], Corrected

This appendix is for completeness only and is courtesy of Brodal *et al.* [1]. We have, however, corrected the errors and added the missing counters. The changes have been marked with the symbol $\sharp$. We have also inserted the symbol $\dagger$ which represents the counters used for the triplet distance calculation the few places where it was missing. Equivalently we have inserted the symbol $\star$ which represents bottlenecks of the quartet distance calculation the few places where it was missing.

Additionally, we found that the counter $n^{X}_{(i(i\bullet))}$ from [1] was not necessary. It has been removed from the list.

While we have tried to fix all errors we cannot promise we have not missed some. As we have not found examples of our implementation giving a wrong result, however, we do believe any errors to be localized to these tables.

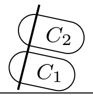
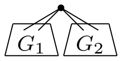
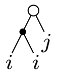
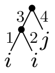
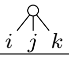
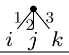
T_1	T_2		$\mathbf{CC} \rightarrow \mathbf{C}$ 	$\mathbf{GG} \rightarrow \mathbf{G}$ 
Counting resolved-resolved triplets				
		1,2	$\sum_{i=1}^{d} n_i^{C_1} \cdot n_{i\uparrow\bullet}^{C_2}$	$\sigma\Big[\sum_{i=1}^{d} n_{(ii)}^{G'} \left(n_{\bullet}^{G''} - n_i^{G''}\right)\Big]$
		3	$\sum_{i=1}^{d} \binom{n_i^{C_1}}{2} \left(n_{\bullet}^{C_2} - n_i^{C_2}\right)$	
		4	$\sum_{i=1}^{d} \left(n_{\bullet}^{C_1} - n_i^{C_1}\right) n_{(ii)}^{C_2}$	
Counting unresolved-unresolved triplets				
		1,2,3	$\sum_{i=1}^{d} n_i^{C_1} \left(n_{\bullet\square}^{C_2} - n_{i\bullet}^{C_2}\right)$	$\sigma\Big[\sum_{i=1}^{d} n_i^{G'} \left(n_{\bullet\square}^{G''} - n_{i\bullet}^{G''}\right)\Big]$

Figure A.1: Sums used for finding A and E for the triplet distance calculation.

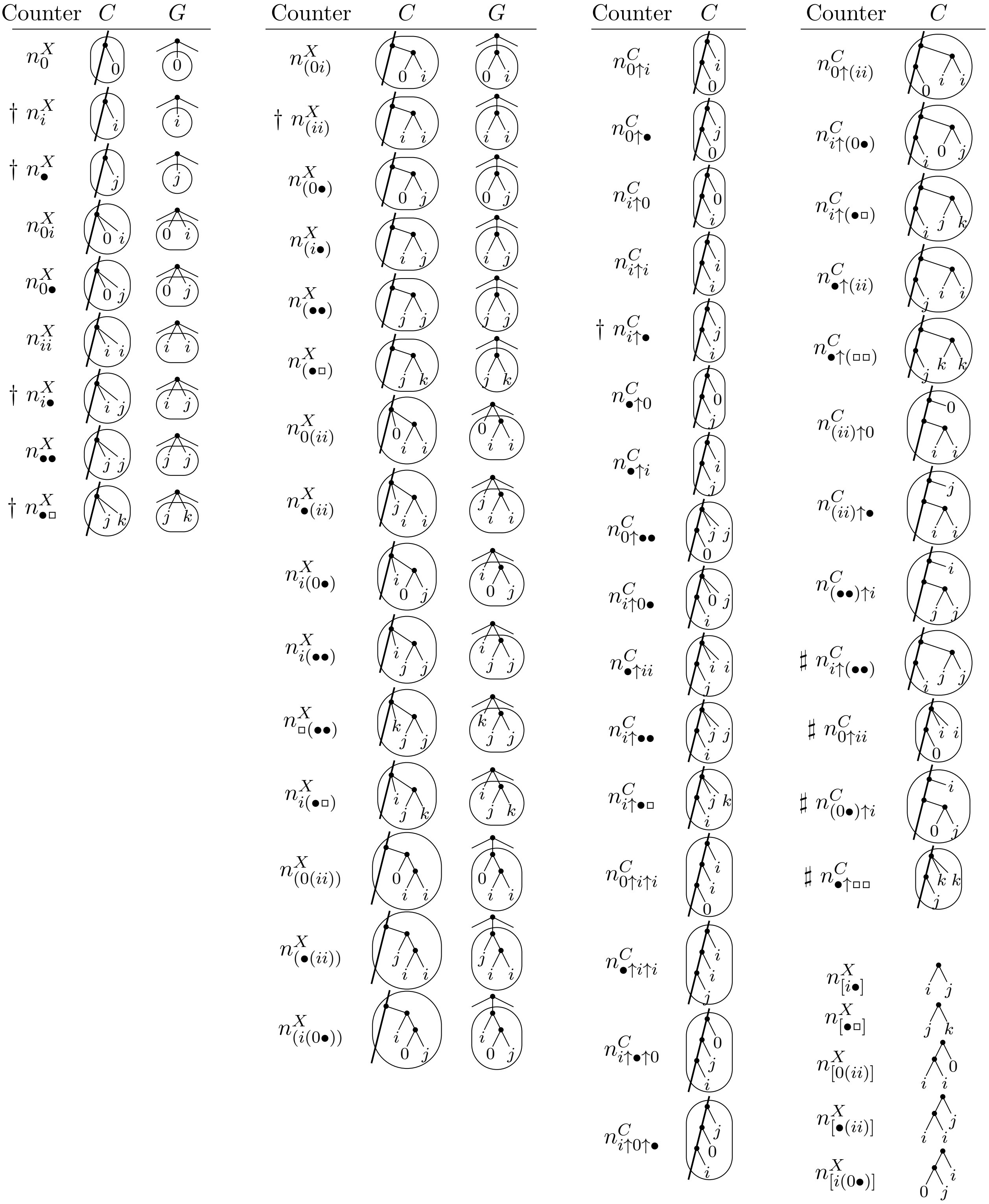
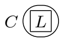
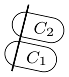
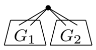
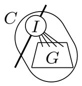

Counter
C
G
n^X_0
† n^X_i
† $n^X_\bullet$
n^X_{0i}
$n^X_{0\bullet}$
n^X_{ii}
† $n^X_{i\bullet}$
$n^X_{\bullet\bullet}$
† $n^X_{\bullet\square}$
$n^X_{(0i)}$
† $n^X_{(ii)}$
$n^X_{(0\bullet)}$
$n^X_{(i\bullet)}$
$n^X_{(\bullet\bullet)}$
$n^X_{(\bullet\square)}$
$n^X_{0(ii)}$
$n^X_{\bullet(ii)}$
$n^X_{i(0\bullet)}$
$n^X_{i(\bullet\bullet)}$
$n^X_{\square(\bullet\bullet)}$
$n^X_{i(\bullet\square)}$
$n^X_{(0(ii))}$
$n^X_{(\bullet(ii))}$
$n^X_{(i(0\bullet))}$
$n^C_{0\uparrow i}$
$n^C_{0\uparrow\bullet}$
$n^C_{i\uparrow 0}$
$n^C_{i\uparrow i}$
† $n^C_{i\uparrow\bullet}$
$n^C_{\bullet\uparrow 0}$
$n^C_{\bullet\uparrow i}$
$n^C_{0\uparrow\bullet\bullet}$
$n^C_{i\uparrow 0\bullet}$
$n^C_{\bullet\uparrow ii}$
$n^C_{i\uparrow\bullet\bullet}$
$n^C_{i\uparrow\bullet\square}$
$n^C_{0\uparrow i\uparrow i}$
$n^C_{\bullet\uparrow i\uparrow i}$
$n^C_{i\uparrow\bullet\uparrow 0}$
$n^C_{i\uparrow 0\uparrow\bullet}$
$n^C_{0\uparrow(ii)}$
$n^C_{i\uparrow(0\bullet)}$
$n^C_{i\uparrow(\bullet\square)}$
$n^C_{\bullet\uparrow(ii)}$
$n^C_{\bullet\uparrow(\square\square)}$
$n^C_{(ii)\uparrow 0}$
$n^C_{(ii)\uparrow\bullet}$
$n^C_{(\bullet\bullet)\uparrow i}$
♯ $n^C_{i\uparrow(\bullet\bullet)}$
♯ $n^C_{0\uparrow ii}$
♯ $n^C_{(0\bullet)\uparrow i}$
♯ $n^C_{\bullet\uparrow\square\square}$
$n^X_{[i\bullet]}$
$n^X_{[\bullet\square]}$
$n^X_{[0(ii)]}$
$n^X_{[\bullet(ii)]}$
$n^X_{[i(0\bullet)]}$

Figure A.2: Illustration of the different counters used to find A and E for the triplet distance calculation (marked with †) and A for the quartet distance calculation.

Counter	 **L → C** n_{-}^{C}	 **CC → C** n_{-}^{C}	 **GG → G** n_{-}^{G}	 **C → G** n_{-}^{G}	 **IG → C** n_{-}^{C}
n_{0}^{X}	$\begin{cases}1 & \text{if color}(L)=0\\ 0 & \text{otherwise}\end{cases}$	$n_{0}^{C_1}+n_{0}^{C_2}$	$n_{0}^{G_1}+n_{0}^{G_2}$	n_{0}^{C}	n_{0}^{G}
† n_{i}^{X}	$\begin{cases}1 & \text{if color}(L)=i\\ 0 & \text{otherwise}\end{cases}$	$n_{i}^{C_1}+n_{i}^{C_2}$	$n_{i}^{G_1}+n_{i}^{G_2}$	n_{i}^{C}	n_{i}^{G}
Common	0	$n_{-}^{C_1}+n_{-}^{C_2}$	$n_{-}^{G_1}+n_{-}^{G_2}+g_{-}(G_1,G_2)+g_{-}(G_2,G_1)$	0	n_{-}^{G}
n_{0i}^{X}			$n_{0}^{G'}n_{i}^{G''}$		
$n_{0\bullet}^{X}$			$n_{0}^{G'}n_{\bullet}^{G''}$		
n_{ii}^{X}			$\frac{1}{2}n_{i}^{G'}n_{i}^{G''}$		
† $n_{i\bullet}^{X}$			$n_{i}^{G'}(n_{\bullet}^{G''}-n_{i}^{G''})$		
$n_{0(ii)}^{X}$			$g_{-}(G',G'') =$ $n_{0}^{G'}n_{(ii)}^{G''}$		
$n_{\bullet(ii)}^{X}$			$(n_{\bullet}^{G'}-n_{i}^{G'})n_{(ii)}^{G''}$		
$n_{i(0\bullet)}^{X}$			$n_{i}^{G'}(n_{(0\bullet)}^{G''}-n_{(0i)}^{G''})$		
$n_{i(\bullet\bullet)}^{X}$			$n_{i}^{G'}(n_{(\bullet\bullet)}^{G''}-n_{(ii)}^{G''})$		
$n_{i(\bullet\square)}^{X}$			$n_{i}^{G'}(n_{(\bullet\square)}^{G''}-n_{(i\bullet)}^{G''})$		
Common	0	$n_{-}^{C_1}+n_{-}^{C_2}$	$n_{-}^{G_1}+n_{-}^{G_2}$		n_{-}^{G}
$n_{(0i)}^{X}$				$n_{0}^{C}n_{i}^{C}$	
† $n_{(ii)}^{X}$				$\binom{n_{i}^{C}}{2}$	
$n_{(0\bullet)}^{X}$				$n_{0}^{C}n_{\bullet}^{C}$	
$n_{(i\bullet)}^{X}$				$n_{i}^{C}(n_{\bullet}^{C}-n_{i}^{C})$	
$n_{(0(ii))}^{X}$				$n_{[0(ii)]}^{C}$	
$n_{(\bullet(ii))}^{X}$				$n_{[\bullet(ii)]}^{C}$	
$n_{(i(0\bullet))}^{X}$				$n_{[i(0\bullet)]}^{C}$	
Common	0	$n_{-}^{C_1}+n_{-}^{C_2}+f_{-}(C_1,C_2)$	$n_{-}^{G_1}+n_{-}^{G_2}+g_{-}(G_1,G_2)+g_{-}(G_2,G_1)$	n_{-}^{C}	n_{-}^{G}
$n_{[0(ii)]}^{X}$		$\binom{n_{i}^{C_1}}{2}n_{0}^{C_2}+n_{0}^{C_1}n_{(ii)}^{C_2}+n_{i}^{C_1}n_{i\uparrow 0}^{C_2}$	$n_{0}^{G'}n_{(ii)}^{G''}$		
$n_{[\bullet(ii)]}^{X}$		$\binom{n_{i}^{C_1}}{2}(n_{\bullet}^{C_2}-n_{i}^{C_2})+(n_{\bullet}^{C_1}-n_{i}^{C_1})n_{(ii)}^{C_2}+n_{i}^{C_1}n_{i\uparrow\bullet}^{C_2}$	$(n_{\bullet}^{G'}-n_{i}^{G'})n_{(ii)}^{G''}$	$= f_{-}(C_1,C_2)$ $g_{-}(G',G'')$	
$n_{[i(0\bullet)]}^{X}$		$n_{0}^{C_1}(n_{\bullet}^{C_1}-n_{i}^{C_1})n_{i}^{C_2}+n_{i}^{C_1}(n_{(0\bullet)}^{C_2}-n_{(0i)}^{C_2})+n_{0}^{C_1}n_{\bullet\uparrow i}^{C_2}+(n_{\bullet}^{C_1}-n_{i}^{C_1})n_{0\uparrow i}^{C_2}$	$n_{i}^{G'}(n_{(0\bullet)}^{G''}-n_{(0i)}^{G''})$		
Common	0	$n_{-}^{C_1}+n_{-}^{C_2}+f_{-}(C_1,C_2)$	not defined	not defined	0
$n_{0\uparrow i}^{C}$		$n_{0}^{C_1}n_{i}^{C_2}$			
$n_{0\uparrow\bullet}^{C}$		$n_{0}^{C_1}n_{\bullet}^{C_2}$			
$n_{i\uparrow 0}^{C}$		$n_{i}^{C_1}n_{0}^{C_2}$			
$n_{i\uparrow i}^{C}$		$n_{i}^{C_1}n_{i}^{C_2}$			
† $n_{i\uparrow\bullet}^{C}$		$n_{i}^{C_1}(n_{\bullet}^{C_2}-n_{i}^{C_2})$			
$n_{\bullet\uparrow 0}^{C}$		$n_{\bullet}^{C_1}n_{0}^{C_2}$			
$n_{\bullet\uparrow i}^{C}$		$(n_{\bullet}^{C_1}-n_{i}^{C_1})n_{i}^{C_2}$			
$n_{0\uparrow(ii)}^{C}$		$n_{0}^{C_1}n_{(ii)}^{C_2}$			
♯ $n_{i\uparrow(0\bullet)}^{C}$		$n_{i}^{C_1}(n_{(0\bullet)}^{C_2}-n_{(0i)}^{C_2})$			
$n_{i\uparrow(\bullet\square)}^{C}$		$n_{i}^{C_1}(n_{(\bullet\square)}^{C_2}-n_{(i\bullet)}^{C_2})$			
$n_{\bullet\uparrow(ii)}^{C}$		$(n_{\bullet}^{C_1}-n_{i}^{C_1})n_{(ii)}^{C_2}$			
$n_{0\uparrow\bullet\bullet}^{C}$		$n_{0}^{C_1}n_{\bullet\bullet}^{C_2}$			
$n_{i\uparrow 0\bullet}^{C}$		$n_{i}^{C_1}(n_{0\bullet}^{C_2}-n_{0i}^{C_2})$			
$n_{i\uparrow\bullet\bullet}^{C}$	$f_{-}(C_1,C_2) =$	$n_{i}^{C_1}(n_{\bullet\bullet}^{C_2}-n_{ii}^{C_2})$			
$n_{i\uparrow\bullet\square}^{C}$		$n_{i}^{C_1}(n_{\bullet\square}^{C_2}-n_{i\bullet}^{C_2})$			
$n_{\bullet\uparrow ii}^{C}$		$(n_{\bullet}^{C_1}-n_{i}^{C_1})n_{ii}^{C_2}$			
$n_{(ii)\uparrow 0}^{C}$		$n_{(ii)}^{C_1}n_{0}^{C_2}$			
$n_{(ii)\uparrow\bullet}^{C}$		$n_{(ii)}^{C_1}(n_{\bullet}^{C_2}-n_{i}^{C_2})$			
$n_{(\bullet\bullet)\uparrow i}^{C}$		$(n_{(\bullet\bullet)}^{C_1}-n_{(ii)}^{C_1})n_{i}^{C_2}$			
$n_{0\uparrow i\uparrow i}^{C}$		$n_{0}^{C_1}n_{i\uparrow i}^{C_2}+n_{0\uparrow i}^{C_1}n_{i}^{C_2}$			
$n_{i\uparrow\bullet\uparrow 0}^{C}$		$n_{i}^{C_1}(n_{\bullet\uparrow 0}^{C_2}-n_{i\uparrow 0}^{C_2})+n_{i\uparrow\bullet}^{C_1}n_{0}^{C_2}$			
$n_{i\uparrow 0\uparrow\bullet}^{C}$		$n_{i}^{C_1}(n_{0\uparrow\bullet}^{C_2}-n_{0\uparrow i}^{C_2})+n_{i\uparrow 0}^{C_1}(n_{\bullet}^{C_2}-n_{i}^{C_2})$			
$n_{\bullet\uparrow i\uparrow i}^{C}$		$(n_{\bullet}^{C_1}-n_{i}^{C_1})n_{i\uparrow i}^{C_2}+n_{\bullet\uparrow i}^{C_1}n_{i}^{C_2}$			
♯ $n_{i\uparrow(\bullet\bullet)}^{C}$		$n_{i}^{C_1}(n_{(\bullet\bullet)}^{C_2}-n_{(ii)}^{C_2})$			
♯ $n_{0\uparrow ii}^{C}$		$n_{0}^{C_1}n_{ii}^{C_2}$			
♯ $n_{(0\bullet)\uparrow i}^{C}$		$(n_{(0\bullet)}^{C_1}-n_{(0i)}^{C_1})n_{i}^{C_2}$			

† $n_{\bullet}^{X}$	=	$\sum_{i=1}^{d} n_{i}^{X}$
$n_{\bullet\bullet}^{X}$	=	$\sum_{i=1}^{d} n_{ii}^{X}$
† $n_{\bullet\square}^{X}$	=	$\frac{1}{2}\sum_{i=1}^{d} n_{i\bullet}^{X}$
$n_{\square(\bullet\bullet)}^{X}$	=	$\sum_{i=1}^{d} n_{i(\bullet\bullet)}^{X}$
$n_{(\bullet\bullet)}^{X}$	=	$\sum_{i=1}^{d} n_{(ii)}^{X}$
$n_{(\bullet\square)}^{X}$	=	$\frac{1}{2}\sum_{i=1}^{d} n_{(i\bullet)}^{X}$
♯ $n_{\bullet\uparrow(\square\square)}^{C}$	=	$\sum_{i=1}^{d} n_{\bullet\uparrow(ii)}^{C}$
$n_{[i\bullet]}^{X}$	=	$n_{i}^{X}(n_{\bullet}^{X}-n_{i}^{X})$
♯ $n_{[\bullet\square]}^{X}$	=	$\frac{1}{2}\sum_{i=1}^{d} n_{[i\bullet]}^{X}$
♯ $n_{\bullet\uparrow\square\square}^{C}$	=	$\sum_{i=1}^{d} n_{i\uparrow\bullet\bullet}^{C}$

Figure A.3: Counters used for the *A* and *E* calculation for the triplet distance calculation (marked with †) and counters used for the *A* calculation of the quartet distance calculation.

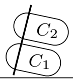
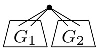
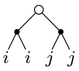
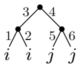
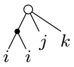
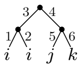
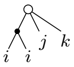
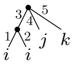
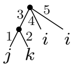
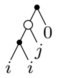
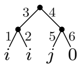
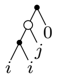
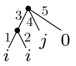
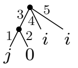
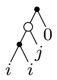
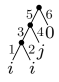
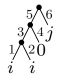
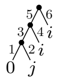
Case	T_1	T_2		**CC→C** 		**GG→G** 
$\alpha\alpha$			1,2,5,6	$\sum_{i=1}^{d} n_i^{C_1} \cdot n_{i\uparrow(\bullet\bullet)}^{C_2}$	3:4	$\sum_{i=1}^{d} n_{(ii)}^{G_1} \left(n_{(\bullet\bullet)}^{G_2} - n_{(ii)}^{G_2}\right)$
			3,4	$\sum_{i=1}^{d} \binom{n_i^{C_1}}{2} \left(n_{(\bullet\bullet)}^{C_2} - n_{(ii)}^{C_2}\right)$		
$\beta\alpha$			1,2	$\sum_{i=1}^{d} n_i^{C_1} \cdot n_{i\uparrow(\bullet\square)}^{C_2}$	3:4	$\sigma\Big[\sum_{i=1}^{d} n_{(ii)}^{G'} \left(n_{(\bullet\square)}^{G''} - n_{(i\bullet)}^{G''}\right)\Big]$
			3	$\sum_{i=1}^{d} \binom{n_i^{C_1}}{2} \left(n_{(\bullet\square)}^{C_2} - n_{(i\bullet)}^{C_2}\right)$		
			4	$\sum_{i=1}^{d} \left(n_{[\bullet\square]}^{C_1} - n_{[i\bullet]}^{C_1}\right) n_{(ii)}^{C_2}$		
			5,6$\sharp$	$\sum_{k=1}^{d} n_k^{C_1} \left(n_{\bullet\uparrow(\square\square)}^{C_2} - n_{k\uparrow(\bullet\bullet)}^{C_2} - n_{\bullet\uparrow(kk)}^{C_2}\right)$		
$\beta\beta$			1,2	$\sum_{i=1}^{d} n_i^{C_1} \cdot n_{i\uparrow\bullet\square}^{C_2}$	3:45	$\sigma\Big[\sum_{i=1}^{d} n_{(ii)}^{G'} \left(n_{\bullet\square}^{G''} - n_{i\bullet}^{G''}\right)\Big]$
			3	$\sum_{i=1}^{d} \binom{n_i^{C_1}}{2} \left(n_{\bullet\square}^{C_2} - n_{i\bullet}^{C_2}\right)$	34:5,35:4	$\sigma\Big[\sum_{k=1}^{d} n_k^{G'} \left(n_{\square(\bullet\bullet)}^{G''} - n_{k(\bullet\bullet)}^{G''} - n_{\bullet(kk)}^{G''}\right)\Big]$
			4,5	$\sum_{k=1}^{d} n_k^{C_1} \left(n_{\square(\bullet\bullet)}^{C_2} - n_{k(\bullet\bullet)}^{C_2} - n_{\bullet(kk)}^{C_2}\right)$		
			1,2$\sharp$	$\sum_{k=1}^{d} n_k^{C_1} \left(n_{\bullet\uparrow\square\square}^{C_2} - n_{k\uparrow\bullet\bullet}^{C_2} - n_{\bullet\uparrow kk}^{C_2}\right)$	3:45	$\sigma\Big[\sum_{i=1}^{d} n_{ii}^{G'} \left(n_{(\bullet\square)}^{G''} - n_{(i\bullet)}^{G''}\right)\Big]$
			3	$\sum_{i=1}^{d} \left(n_{[\bullet\square]}^{C_1} - n_{[i\bullet]}^{C_1}\right) n_{ii}^{C_2}$	34:5,35:4	$\sigma\Big[\sum_{i=1}^{d} n_i^{G'} \cdot n_{i(\bullet\square)}^{G''}\Big]$
			4,5	$\sum_{i=1}^{d} n_i^{C_1} \cdot n_{i(\bullet\square)}^{C_2}$		
$\gamma\alpha$			1,2	$\sum_{i=1}^{d} n_i^{C_1} \cdot n_{i\uparrow(0\bullet)}^{C_2}$	3:4	$\sigma\Big[\sum_{i=1}^{d} n_{(ii)}^{G'} \left(n_{(0\bullet)}^{G''} - n_{(0i)}^{G''}\right)\Big]$
			3:45	$\sum_{i=1}^{d} \binom{n_i^{C_1}}{2} \left(n_{(0\bullet)}^{C_2} - n_{(0i)}^{C_2}\right)$		
			4	$\sum_{i=1}^{d} n_0^{C_1} \left(n_\bullet^{C_1} - n_i^{C_1}\right) n_{(ii)}^{C_2}$		
			5	$\sum_{i=1}^{d} \left(n_\bullet^{C_1} - n_i^{C_1}\right) n_{0\uparrow(ii)}^{C_2}$		
			6	$\sum_{i=1}^{d} n_0^{C_1} \cdot n_{\bullet\uparrow(ii)}^{C_2}$		
$\gamma\beta$			3	$\sum_{i=1}^{d} \binom{n_i^{C_1}}{2} \left(n_{0\bullet}^{C_2} - n_{0i}^{C_2}\right)$	3:45	$\sigma\Big[\sum_{i=1}^{d} n_{(ii)}^{G'} \left(n_{0\bullet}^{G''} - n_{0i}^{G''}\right)\Big]$
			1,2	$\sum_{i=1}^{d} n_i^{C_1} \cdot n_{i\uparrow 0\bullet}^{C_2}$	34:5	$\sigma\Big[n_0^{G'} \cdot n_{\square(\bullet\bullet)}^{G''}\Big]$
			5	$\sum_{i=1}^{d} n_0^{C_1} \cdot n_{\bullet(ii)}^{C_2}$	35:4	$\sigma\Big[\sum_{i=1}^{d} \left(n_\bullet^{G'} - n_i^{G'}\right) n_{0(ii)}^{G''}\Big]$
			4	$\sum_{i=1}^{d} \left(n_\bullet^{C_1} - n_i^{C_1}\right) n_{0(ii)}^{C_2}$		
			1	$\sum_{i=1}^{d} \left(n_\bullet^{C_1} - n_i^{C_1}\right) n_{0\uparrow ii}^{C_2}$	3:45	$\sigma\Big[\sum_{i=1}^{d} n_{ii}^{G'} \left(n_{(0\bullet)}^{G''} - n_{(0i)}^{G''}\right)\Big]$
			2	$\sum_{i=1}^{d} n_0^{C_1} \cdot n_{\bullet\uparrow ii}^{C_2}$	34:5,35:4	$\sigma\Big[\sum_{i=1}^{d} n_i^{G'} \cdot n_{i(0\bullet)}^{G''}\Big]$
			3	$\sum_{i=1}^{d} n_0^{C_1} \left(n_\bullet^{C_1} - n_i^{C_1}\right) n_{ii}^{C_2}$		
			4,5	$\sum_{i=1}^{d} n_i^{C_1} \cdot n_{i(0\bullet)}^{C_2}$		
$\gamma\gamma$			1,2	$\sum_{i=1}^{d} n_i^{C_1} \cdot n_{i\uparrow\bullet\uparrow 0}^{C_2}$	5:6	$\sigma\Big[\sum_{i=1}^{d} n_0^{G'} \cdot n_{(\bullet(ii))}^{G''}\Big]$
			3	$\sum_{i=1}^{d} \binom{n_i^{C_1}}{2} \left(n_{\bullet\uparrow 0}^{C_2} - n_{i\uparrow 0}^{C_2}\right)$		
			4	$\sum_{i=1}^{d} \left(n_\bullet^{C_1} - n_i^{C_1}\right) n_{(ii)\uparrow 0}^{C_2}$		
			5	$\sum_{i=1}^{d} n_{[\bullet(ii)]}^{C_1} \cdot n_0^{C_2}$		
			6	$\sum_{i=1}^{d} n_0^{C_1} \cdot n_{(\bullet(ii))}^{C_2}$		
			1,2	$\sum_{i=1}^{d} n_i^{C_1} \cdot n_{i\uparrow 0\uparrow\bullet}^{C_2}$	5:6	$\sigma\Big[\sum_{i=1}^{d} \left(n_\bullet^{G'} - n_i^{G'}\right) n_{(0(ii))}^{G''}\Big]$
			3	$\sum_{i=1}^{d} \binom{n_i^{C_1}}{2} \left(n_{0\uparrow\bullet}^{C_2} - n_{0\uparrow i}^{C_2}\right)$		
			4	$\sum_{i=1}^{d} n_0^{C_1} \cdot n_{(ii)\uparrow\bullet}^{C_2}$		
			5	$\sum_{i=1}^{d} n_{[0(ii)]}^{C_1} \left(n_\bullet^{C_2} - n_i^{C_2}\right)$		
			6	$\sum_{i=1}^{d} \left(n_\bullet^{C_1} - n_i^{C_1}\right) n_{(0(ii))}^{C_2}$		
			1	$\sum_{i=1}^{d} n_0^{C_1} \cdot n_{\bullet\uparrow i\uparrow i}^{C_2}$	5:6	$\sigma\Big[\sum_{i=1}^{d} n_i^{G'} \cdot n_{(i(0\bullet))}^{G''}\Big]$
			2	$\sum_{i=1}^{d} \left(n_\bullet^{C_1} - n_i^{C_1}\right) n_{0\uparrow i\uparrow i}^{C_2}$		
			3	$\sum_{i=1}^{d} n_0^{C_1} \left(n_\bullet^{C_1} - n_i^{C_1}\right) n_{i\uparrow i}^{C_2}$		
			4	$\sum_{i=1}^{d} n_i^{C_1} \cdot n_{(0\bullet)\uparrow i}^{C_2}$		
			5	$\sum_{i=1}^{d} n_{[i(0\bullet)]}^{C_1} \cdot n_i^{C_2}$		
			6	$\sum_{i=1}^{d} n_i^{C_1} \cdot n_{(i(0\bullet))}^{C_2}$		

Figure A.4: Sums used for finding A for the quartet distance calculation.

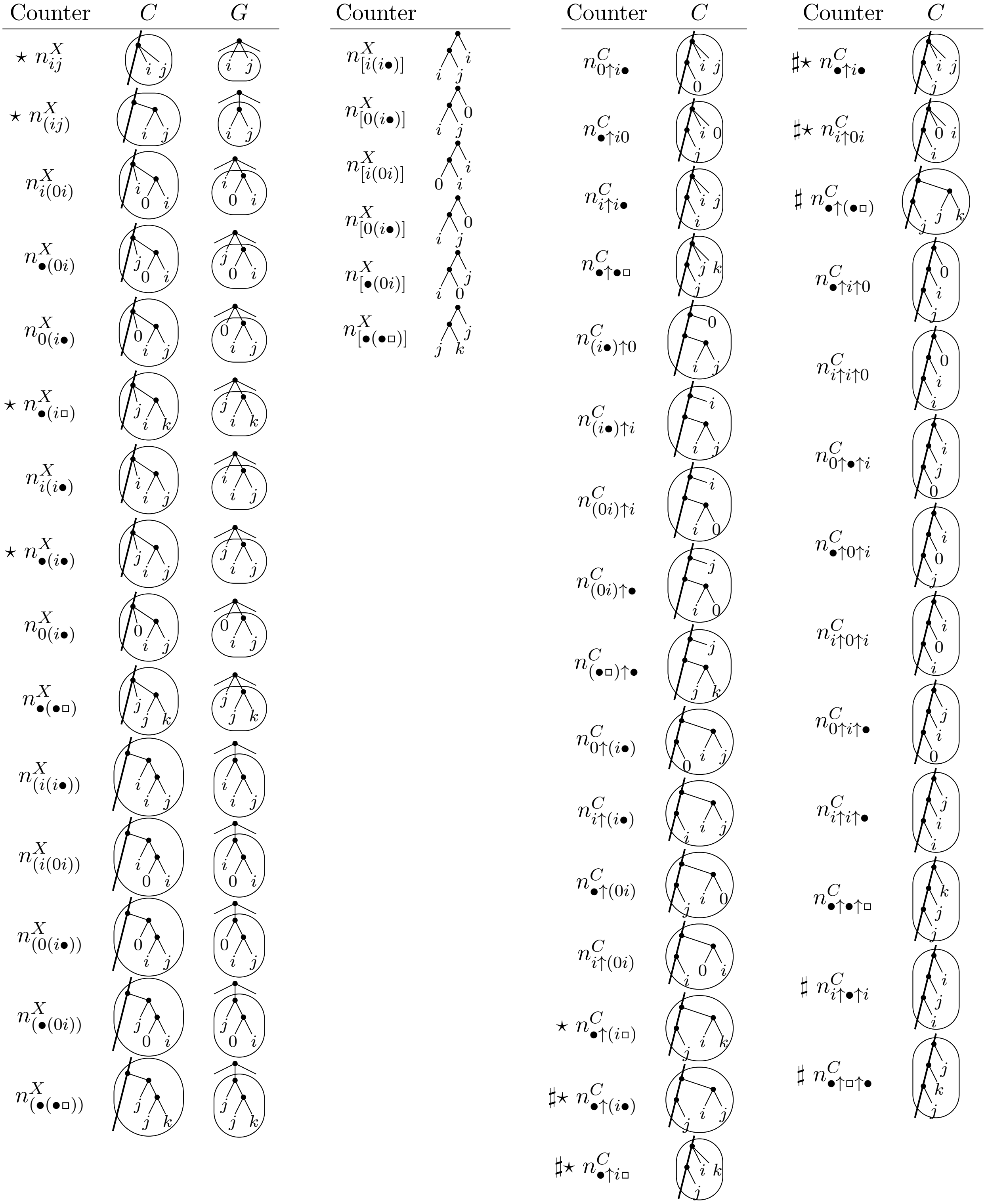
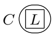
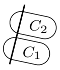
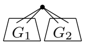
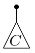
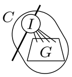

Counter
C
G
$\star\ n^X_{ij}$
$\star\ n^X_{(ij)}$
$n^X_{i(0i)}$
$n^X_{\bullet(0i)}$
$n^X_{0(i\bullet)}$
$\star\ n^X_{\bullet(i\square)}$
$n^X_{i(i\bullet)}$
$\star\ n^X_{\bullet(i\bullet)}$
$n^X_{0(i\bullet)}$
$n^X_{\bullet(\bullet\square)}$
$n^X_{(i(i\bullet))}$
$n^X_{(i(0i))}$
$n^X_{(0(i\bullet))}$
$n^X_{(\bullet(0i))}$
$n^X_{(\bullet(\bullet\square))}$
Counter
$n^X_{[i(i\bullet)]}$
$n^X_{[0(i\bullet)]}$
$n^X_{[i(0i)]}$
$n^X_{[0(i\bullet)]}$
$n^X_{[\bullet(0i)]}$
$n^X_{[\bullet(\bullet\square)]}$
Counter
C
$n^C_{0\uparrow i\bullet}$
$n^C_{\bullet\uparrow i0}$
$n^C_{i\uparrow i\bullet}$
$n^C_{\bullet\uparrow\bullet\square}$
$n^C_{(i\bullet)\uparrow 0}$
$n^C_{(i\bullet)\uparrow i}$
$n^C_{(0i)\uparrow i}$
$n^C_{(0i)\uparrow\bullet}$
$n^C_{(\bullet\square)\uparrow\bullet}$
$n^C_{0\uparrow(i\bullet)}$
$n^C_{i\uparrow(i\bullet)}$
$n^C_{\bullet\uparrow(0i)}$
$n^C_{i\uparrow(0i)}$
$\star\ n^C_{\bullet\uparrow(i\square)}$
$\sharp\star\ n^C_{\bullet\uparrow(i\bullet)}$
$\sharp\star\ n^C_{\bullet\uparrow i\square}$
Counter
C
$\sharp\star\ n^C_{\bullet\uparrow i\bullet}$
$\sharp\star\ n^C_{i\uparrow 0i}$
$\sharp\ n^C_{\bullet\uparrow(\bullet\square)}$
$n^C_{\bullet\uparrow i\uparrow 0}$
$n^C_{i\uparrow i\uparrow 0}$
$n^C_{0\uparrow\bullet\uparrow i}$
$n^C_{\bullet\uparrow 0\uparrow i}$
$n^C_{i\uparrow 0\uparrow i}$
$n^C_{0\uparrow i\uparrow\bullet}$
$n^C_{i\uparrow i\uparrow\bullet}$
$n^C_{\bullet\uparrow\bullet\uparrow\square}$
$\sharp\ n^C_{i\uparrow\bullet\uparrow i}$
$\sharp\ n^C_{\bullet\uparrow\square\uparrow\bullet}$

Figure A.5: Illustration of the different additional counters used to find B for the quartet distance calculation.

Counter	 C $\boxed{L}$ $\mathbf{L \rightarrow C}$ n_-^C	 $\mathbf{CC \rightarrow C}$ n_-^C	 $\mathbf{GG \rightarrow G}$ n_-^G	 $\mathbf{C \rightarrow G}$ n_-^G	 $\mathbf{IG \rightarrow C}$ n_-^C
Common	0	$n_-^{C_1} + n_-^{C_2}$	$n_-^{G_1} + n_-^{G_2} + g_-(G_1, G_2) + g_-(G_2, G_1)$	0	n_-^G
$\star\ n_{ij}^X$			$n_i^{G'} n_j^{G''}$		
$n_{i(0i)}^X$			$n_i^{G'} n_{(0i)}^{G''}$		
$n_{\bullet(0i)}^X$			$(n_\bullet^{G'} - n_i^{G'}) n_{(0i)}^{G''}$		
$n_{0(i\bullet)}^X$			$g_-(G', G'') = \Bigg\{$ $n_0^{G'} n_{(i\bullet)}^{G''}$		
$\star\ n_{\bullet(i\square)}^X$			$\sum_{j=1, j\neq i}^d (n_\bullet^{G'} - n_i^{G'} - n_j^{G'}) n_{(ij)}^{G''}$		
$n_{i(i\bullet)}^X$			$n_i^{G'} n_{(i\bullet)}^{G''}$		
$\sharp\star\ n_{\bullet(i\bullet)}^X$			$\sum_{j=1, j\neq i}^d n_j^{G'} \cdot n_{(ij)}^{G''}$		
Common	0	$n_-^{C_1} + n_-^{C_2}$	$n_-^{G_1} + n_-^{G_2}$		n_-^G
$\sharp\star\ n_{(ij)}^X$				$n_i^C \cdot n_j^C$	
$n_{(i(0i))}^X$				$n_{[i(0i)]}^C$	
$n_{(0(i\bullet))}^X$				$n_{[0(i\bullet)]}^C$	
$n_{(\bullet(0i))}^X$				$n_{[\bullet(0i)]}^C$	
$n_{(\bullet(\bullet\square))}^X$				$n_{[\bullet(\bullet\square)]}^C$	
Common	0	$n_-^{C_1} + n_-^{C_2} + f_-(C_1, C_2)$	$n_-^{G_1} + n_-^{G_2} + g_-(G_1, G_2) + g_-(G_2, G_1)$	n_-^C	n_-^G
$n_{[i(i\bullet)]}^X$		$n_i^{C_1}(n_\bullet^{C_1} - n_i^{C_1}) n_i^{C_2} + n_i^{C_1} n_{\bullet\uparrow i}^{C_2} + (n_\bullet^{C_1} - n_i^{C_1}) n_{i\uparrow i}^{C_2} + n_i^{C_1} n_{(i\bullet)}^{C_2}$	$n_{(i\bullet)}^{G'} n_i^{G''}$		
$\sharp\ n_{[0(i\bullet)]}^X$		$n_i^{C_1}(n_\bullet^{C_1} - n_i^{C_1}) n_0^{C_2} + (n_\bullet^{C_1} - n_i^{C_1}) n_{i\uparrow 0}^{C_2} + n_0^{C_1} n_{(i\bullet)}^{C_2} + n_i^{C_1}(n_{\bullet\uparrow 0}^{C_2} - n_{i\uparrow 0}^{C_2})$	$n_0^{G'} n_{(i\bullet)}^{G''}$	$\Bigg\} = f_-(C_1, C_2)$	
$\sharp\ n_{[i(0i)]}^X$		$n_i^{C_1} n_{(0i)}^{C_2} + n_i^{C_1} n_{0\uparrow i}^{C_2} + (n_0^{C_1} n_i^{C_1}) n_i^{C_2} + n_0^{C_1} \cdot n_{i\uparrow i}^{C_2}$	$n_i^{G'} n_{(0i)}^{G''}$	$g_-(G', G'')$	
$\sharp\ n_{[\bullet(0i)]}^X$		$(n_i^{C_1} n_0^{C_1})(n_\bullet^{C_2} - n_i^{C_2}) + (n_\bullet^{C_1} - n_i^{C_1}) n_{(0i)}^{C_2} + n_0^{C_1} \cdot n_{i\uparrow\bullet}^{C_2} + n_i^{C_1} \cdot (n_{0\uparrow\bullet}^{C_2} - n_{0\uparrow i}^{C_2})$	$n_{(0i)}^{G'}(n_\bullet^{G''} - n_i^{G''})$		
Common	0	$n_-^{C_1} + n_-^{C_2} + f_-(C_1, C_2)$	not defined	not defined	0
$n_{0\uparrow i\bullet}^C$		$n_0^{C_1} n_{i\bullet}^{C_2}$			
$n_{\bullet\uparrow 0i}^C$		$(n_\bullet^{C_1} - n_i^{C_1}) n_{0i}^{C_2}$			
$n_{i\uparrow i\bullet}^C$		$n_i^{C_1} n_{i\bullet}^{C_2}$			
$n_{(i\bullet)\uparrow 0}^C$		$n_{(i\bullet)}^{C_1} n_0^{C_2}$			
$n_{(i\bullet)\uparrow i}^C$		$n_{(i\bullet)}^{C_1} n_i^{C_2}$			
$n_{(0i)\uparrow i}^C$		$n_{(0i)}^{C_1} n_i^{C_2}$			
$n_{(0i)\uparrow\bullet}^C$		$n_{(0i)}^{C_1}(n_\bullet^{C_2} - n_i^{C_2})$			
$n_{0\uparrow(i\bullet)}^C$		$n_0^{C_1} n_{(i\bullet)}^{C_2}$			
$n_{i\uparrow(i\bullet)}^C$		$n_i^{C_1} n_{(i\bullet)}^{C_2}$			
$n_{\bullet\uparrow(0i)}^C$		$(n_\bullet^{C_1} - n_i^{C_1}) n_{(0i)}^{C_2}$			
$n_{i\uparrow(0i)}^C$		$n_i^{C_1} n_{(0i)}^{C_2}$			
$\star\ n_{\bullet\uparrow(i\square)}^C$	$f_-(C_1, C_2) = \Bigg\{$	$\sum_{k=1, k\neq i}^d (n_\bullet^{C_1} - n_k^{C_1} - n_i^{C_1}) n_{(ik)}^{C_2}$			
$n_{\bullet\uparrow i\uparrow 0}^C$		$n_{\bullet\uparrow i}^{C_1} n_0^{C_2} + (n_\bullet^{C_1} - n_i^{C_1}) n_{i\uparrow 0}^{C_2}$			
$n_{i\uparrow i\uparrow 0}^C$		$n_{i\uparrow i}^{C_1} n_0^{C_2} + n_i^{C_1} n_{i\uparrow 0}^{C_2}$			
$n_{0\uparrow\bullet\uparrow i}^C$		$(n_{0\uparrow\bullet}^{C_1} - n_{0\uparrow i}^{C_1}) n_i^{C_2} + n_0^{C_1} n_{\bullet\uparrow i}^{C_2}$			
$n_{\bullet\uparrow 0\uparrow i}^C$		$(n_{\bullet\uparrow 0}^{C_1} - n_{i\uparrow 0}^{C_1}) n_i^{C_2} + (n_\bullet^{C_1} - n_i^{C_1}) n_{0\uparrow i}^{C_2}$			
$n_{i\uparrow 0\uparrow i}^C$		$n_{i\uparrow 0}^{C_1} n_i^{C_2} + n_i^{C_1} n_{0\uparrow i}^{C_2}$			
$n_{0\uparrow i\uparrow\bullet}^C$		$n_0^{C_1} n_{i\uparrow\bullet}^{C_2} + n_{0\uparrow i}^{C_1}(n_\bullet^{C_2} - n_i^{C_2})$			
$n_{i\uparrow i\uparrow\bullet}^C$		$n_{i\uparrow i}^{C_1}(n_\bullet^{C_2} - n_i^{C_2}) + n_i^{C_1} n_{i\uparrow\bullet}^{C_2}$			
$\sharp\ n_{i\uparrow\bullet\uparrow i}^C$		$n_i^{C_1} n_{\bullet\uparrow i}^{C_2} + n_{i\uparrow\bullet}^{C_1} n_i^{C_2}$			
$\sharp\star\ n_{\bullet\uparrow(i\bullet)}^C$		$\sum_{k=1, k\neq i}^d n_k^{C_1} n_{(ik)}^{C_2}$			
$\sharp\star\ n_{\bullet\uparrow i\square}^C$		$\sum_{k=1, k\neq i}^d (n_\bullet^{C_1} - n_k^{C_1} - n_i^{C_1}) n_{ik}^{C_2}$			
$\sharp\star\ n_{\bullet\uparrow i\bullet}^C$		$\sum_{k=1, k\neq i}^d n_k^{C_1} n_{ik}^{C_2}$			
$\sharp\ n_{i\uparrow 0i}^C$		$n_i^{C_1} n_{0i}^{C_2}$			

$$
\boxed{
\begin{aligned}
n_{[\bullet(\bullet\square)]}^X &= \textstyle\sum_{i=1}^d n_{[i(i\bullet)]}^X \\
n_{\bullet(\bullet\square)}^X &= \textstyle\sum_{i=1}^d n_{i(i\bullet)}^X \\
n_{\bullet\uparrow\bullet\square}^C &= \textstyle\sum_{i=1}^d n_{i\uparrow i\bullet}^C \\
n_{(\bullet\square)\uparrow\bullet}^C &= \textstyle\sum_{i=1}^d n_{(i\bullet)\uparrow i}^C \\
\sharp\ n_{\bullet\uparrow\bullet\uparrow\square}^C &= \textstyle\sum_{i=1}^d n_{i\uparrow i\uparrow\bullet}^C \\
\sharp\ n_{\bullet\uparrow(\bullet\square)}^C &= \textstyle\sum_{i=1}^d n_{i\uparrow(i\bullet)}^C \\
\sharp\ n_{\bullet\uparrow\square\uparrow\bullet}^C &= \textstyle\sum_{i=1}^d n_{i\uparrow\bullet\uparrow i}^C
\end{aligned}
}
$$

Figure A.6: Additional counters used for the B calculation of the quartet distance calculation.

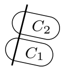
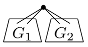
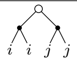
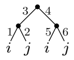
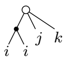
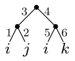
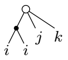
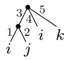
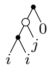
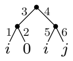
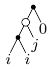
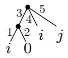
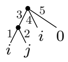
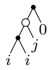
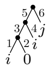
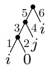
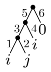
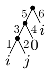
Case	T_1	T_2		$\mathbf{CC}\to\mathbf{C}$ 		$\mathbf{GG}\to\mathbf{G}$ 
$\alpha\alpha$			1,2,5,6	$\sum_{i=1}^{d} n_i^{C_1} \cdot n_{\bullet\uparrow(i\bullet)}^{C_2}$	3:4	$\star$ $\sum_{i=1}^{d-1}\sum_{j=i+1}^{d} n_{(ij)}^{G_1}\cdot n_{(ij)}^{G_2}$
			3,4	$\star$ $\sum_{i=1}^{d-1}\sum_{j=i+1}^{d} n_i^{C_1}\cdot n_j^{C_1}\cdot n_{(ij)}^{C_2}$		
$\beta\alpha$			1,5	$\sum_{i=1}^{d} n_i^{C_1}\cdot n_{\bullet\uparrow(i\square)}^{C_2}$	3:4	$\star$ $\sum_{i=1}^{d}\sum_{j=1,j\neq i}^{d} n_{(ij)}^{G_1}\left(n_{(i\bullet)}^{G_2}-n_{(ij)}^{G_2}\right)$
			2,6	$\sum_{j=1}^{d} n_j^{C_1}\left(n^{C_2}_{\bullet\uparrow(\bullet\square)} - n^{C_2}_{j\uparrow(j\bullet)} - n^{C_2}_{\bullet\uparrow(j\bullet)}\right)$		
			3,4	$\star$ $\sum_{i=1}^{d}\sum_{j=1,j\neq i}^{d} n_i^{C_1}\cdot n_j^{C_1}\left(n^{C_2}_{(i\bullet)} - n^{C_2}_{(ij)}\right)$		
$\beta\beta$			1	$\sum_{i=1}^{d} n_i^{C_1}\cdot n^{C_2}_{\bullet\uparrow i\square}$	3:45	$\star$ $\sigma\left[\sum_{i=1}^{d}\sum_{j=1,j\neq i}^{d} n^{G'}_{(ij)}\left(n^{G''}_{i\bullet} - n^{G''}_{ij}\right)\right]$
			2	$\sum_{j=1}^{d} n_j^{C_1}\left(n^{C_2}_{\bullet\uparrow\bullet\square} - n^{C_2}_{\bullet\uparrow j\bullet} - n^{C_2}_{j\uparrow j\bullet}\right)$	35:4	$\sigma\left[\sum_{i=1}^{d} n^{G'}_{\bullet(i\square)}\cdot n^{G''}_i\right]$
			3	$\star$ $\sum_{i=1}^{d}\sum_{j=1,j\neq i}^{d} n_i^{C_1}\cdot n_j^{C_1}\left(n^{C_2}_{i\bullet} - n^{C_2}_{ij}\right)$	34:5	$\sigma\left[\sum_{k=1}^{d} n_k^{G'}\left(n^{G''}_{\bullet(\bullet\square)} - n^{G''}_{k(k\bullet)} - n^{G''}_{\bullet(k\bullet)}\right)\right]$
			4	$\sum_{i=1}^{d} n_i^{C_1}\cdot n^{C_2}_{\bullet(i\square)}$		
			5	$\sum_{k=1}^{d} n_k^{C_1}\left(n^{C_2}_{\bullet(\bullet\square)} - n^{C_2}_{\bullet(k\bullet)} - n^{C_2}_{k(k\bullet)}\right)$		
$\gamma\alpha$			1	$\sum_{i=1}^{d} n_i^{C_1}\cdot n^{C_2}_{0\uparrow(i\bullet)}$	3:4	$\sigma\left[\sum_{i=1}^{d} n^{G'}_{(0i)} n^{G''}_{(i\bullet)}\right]$
			2	$n_0^{C_1}\cdot\sum_{i=1}^{d} n^{C_2}_{i\uparrow(i\bullet)}$		
			3	$n_0^{C_1}\cdot\sum_{i=1}^{d} n_i^{C_1}\cdot n^{C_2}_{(i\bullet)}$		
			4	$\sum_{i=1}^{d} n_i^{C_1}\left(n_\bullet^{C_1}-n_i^{C_1}\right) n^{C_2}_{(0i)}$		
			5	$\sum_{i=1}^{d} n_i^{C_1}\cdot n^{C_2}_{\bullet\uparrow(0i)}$		
			6	$\sum_{i=1}^{d} \left(n_\bullet^{C_1}-n_i^{C_1}\right)\cdot n^{C_2}_{i\uparrow(0i)}$		
$\gamma\beta$			1	$\sum_{i=1}^{d} n_i^{C_1}\cdot n^{C_2}_{0\uparrow i\bullet}$	3:45	$\sigma\left[\sum_{i=1}^{d} n^{G'}_{(0i)}\cdot n^{G''}_{i\bullet}\right]$
			2	$n_0^{C_1}\cdot n^{C_2}_{\bullet\uparrow\bullet\square}$	34:5	$\sigma\left[\sum_{i=1}^{d} n^{G'}_{i(0i)}\left(n^{G''}_\bullet - n^{G''}_i\right)\right]$
			3	$n_0^{C_1}\cdot\sum_{i=1}^{d} n_i^{C_1}\cdot n^{C_2}_{i\bullet}$	35:4	$\sigma\left[\sum_{i=1}^{d} n^{G'}_{\bullet(0i)}\cdot n^{G''}_i\right]$
			4	$\sum_{i=1}^{d} n_i^{C_1}\cdot n^{C_2}_{\bullet(0i)}$		
			5	$\sum_{i=1}^{d} \left(n_\bullet^{C_1}-n_i^{C_1}\right)\cdot n^{C_2}_{i(0i)}$		
			1	$\sum_{i=1}^{d} n_i^{C_1}\cdot n^{C_2}_{\bullet\uparrow 0i}$	3:45	$\sigma\left[\sum_{i=1}^{d} n^{G'}_{(i\bullet)}\cdot n^{G''}_{0i}\right]$
			2	$\sum_{i=1}^{d} \left(n_\bullet^{C_1}-n_i^{C_1}\right)\cdot n^{C_2}_{i\uparrow 0i}$	34:5	$\sigma\left[n^{G'}_{\bullet(\bullet\square)}\cdot n^{G''}_0\right]$
			3	$\sum_{i=1}^{d} n_i^{C_1}\left(n_\bullet^{C_1}-n_i^{C_1}\right) n^{C_2}_{0i}$	35:4	$\sigma\left[\sum_{i=1}^{d} n^{G'}_{0(i\bullet)}\cdot n^{G''}_i\right]$
			4	$\sum_{i=1}^{d} n_i^{C_1}\cdot n^{C_2}_{0(i\bullet)}$		
			5	$\sharp$ $n_0^{C_1}\cdot n^{C_2}_{\bullet(\bullet\square)}$		
$\gamma\gamma$			1	$\sum_{i=1}^{d} n_i^{C_1}\cdot n^{C_2}_{0\uparrow i\uparrow\bullet}$	5:6	$\sigma\left[\sum_{i=1}^{d} \left(n^{G'}_\bullet - n^{G'}_i\right) n^{G''}_{(i(0i))}\right]$
			2	$n_0^{C_1}\cdot n^{C_2}_{\bullet\uparrow\bullet\uparrow\square}$		
			3	$n_0^{C_1}\cdot\sum_{i=1}^{d} n_i^{C_1}\cdot n^{C_2}_{i\uparrow\bullet}$		
			4	$\sum_{i=1}^{d} n_i^{C_1}\cdot n^{C_2}_{(0i)\uparrow\bullet}$		
			5	$\sum_{i=1}^{d} n^{C_1}_{[i(0i)]}\left(n^{C_2}_\bullet - n^{C_2}_i\right)$		
			6	$\sum_{i=1}^{d} \left(n_\bullet^{C_1}-n_i^{C_1}\right)\cdot n^{C_2}_{(i(0i))}$		
			1	$\sum_{i=1}^{d} n_i^{C_1}\cdot n^{C_2}_{0\uparrow\bullet\uparrow i}$	5:6	$\sigma\left[\sum_{i=1}^{d} n^{G'}_i\cdot n^{G''}_{(\bullet(0i))}\right]$
			2	$n_0^{C_1}\cdot n^{C_2}_{\bullet\uparrow\square\uparrow\bullet}$		
			3	$n_0^{C_1}\cdot\sum_{i=1}^{d} n_i^{C_1}\cdot n^{C_2}_{\bullet\uparrow i}$		
			4	$\sum_{i=1}^{d} \left(n_\bullet^{C_1}-n_i^{C_1}\right)\cdot n^{C_2}_{(0i)\uparrow i}$		
			5	$\sharp$ $\sum_{i=1}^{d} n^{C_1}_{[\bullet(0i)]}\cdot n^{C_2}_i$		
			6	$\sum_{i=1}^{d} n_i^{C_1}\cdot n^{C_2}_{(\bullet(0i))}$		
			1	$\sum_{i=1}^{d} n_i^{C_1}\cdot n^{C_2}_{\bullet\uparrow i\uparrow 0}$	5:6	$\sharp$ $\sigma\left[n_0^{G'}\cdot n^{G''}_{(\bullet(\bullet\square))}\right]$
			2	$\sum_{i=1}^{d} \left(n_\bullet^{C_1}-n_i^{C_1}\right) n^{C_2}_{i\uparrow i\uparrow 0}$		
			3	$\sum_{i=1}^{d} n_i^{C_1}\left(n_\bullet^{C_1}-n_i^{C_1}\right) n^{C_2}_{i\uparrow 0}$		
			4	$\sum_{i=1}^{d} n_i^{C_1}\cdot n^{C_2}_{(i\bullet)\uparrow 0}$		
			5	$\sharp$ $n_0^{C_2}\cdot\sum_{i=1}^{d} n^{C_1}_{[i(i\bullet)]}$		
			6	$n_0^{C_1}\cdot n^{C_2}_{(\bullet(\bullet\square))}$		
			1	$\sum_{i=1}^{d} n_i^{C_1}\cdot n^{C_2}_{\bullet\uparrow 0\uparrow i}$	5:6	$\sigma\left[\sum_{i=1}^{d} n^{G'}_i\cdot n^{G''}_{(0(i\bullet))}\right]$
			2	$\sum_{i=1}^{d} \left(n_\bullet^{C_1}-n_i^{C_1}\right) n^{C_2}_{i\uparrow 0\uparrow i}$		
			3	$\sum_{i=1}^{d} n_i^{C_1}\left(n_\bullet^{C_1}-n_i^{C_1}\right)\cdot n^{C_2}_{0\uparrow i}$		
			4	$n_0^{C_1}\cdot n^{C_2}_{(\bullet\square)\uparrow\bullet}$		
			5	$\sum_{i=1}^{d} n^{C_1}_{[0(i\bullet)]}\cdot n^{C_2}_i$		
			6	$\sum_{i=1}^{d} n_i^{C_1}\cdot n^{C_2}_{(0(i\bullet))}$		

Figure A.7: Sums used for finding B for the quartet distance calculation.

Appendix B

Additional Counters and Sums for the A and B Calculations

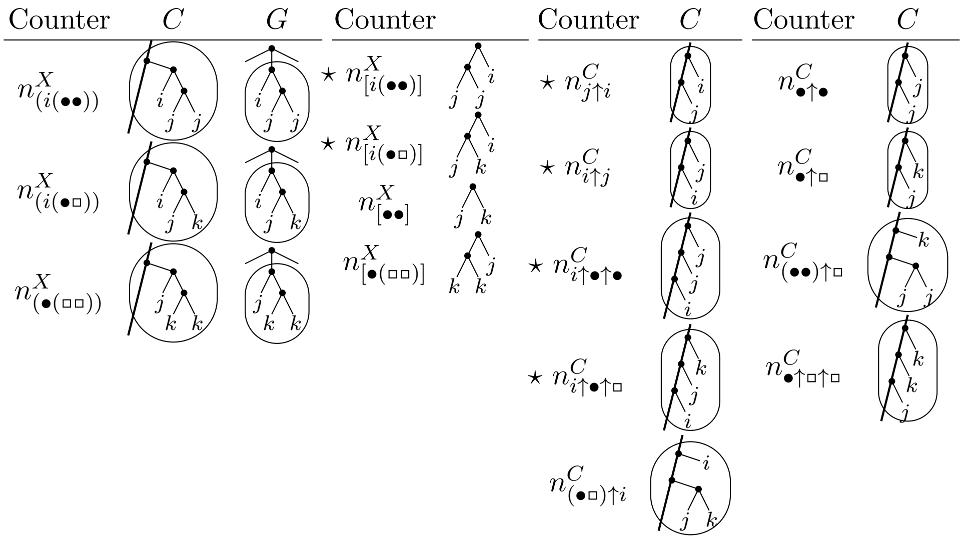

Counter
C
G
Counter
Counter
C
Counter
C
$n^X_{(i(\bullet\bullet))}$
$n^X_{(i(\bullet\square))}$
$n^X_{(\bullet(\square\square))}$
⋆ $n^X_{[i(\bullet\bullet)]}$
⋆ $n^X_{[i(\bullet\square)]}$
$n^X_{[\bullet\bullet]}$
$n^X_{[\bullet(\square\square)]}$
⋆ $n^C_{j\uparrow i}$
⋆ $n^C_{i\uparrow j}$
⋆ $n^C_{i\uparrow\bullet\uparrow\bullet}$
⋆ $n^C_{i\uparrow\bullet\uparrow\square}$
$n^C_{(\bullet\square)\uparrow i}$
$n^C_{\bullet\uparrow\bullet}$
$n^C_{\bullet\uparrow\square}$
$n^C_{(\bullet\bullet)\uparrow\square}$
$n^C_{\bullet\uparrow\square\uparrow\square}$

Figure B.1: Illustration of the additional counters used to find A for our variation of the quartet distance calculation.

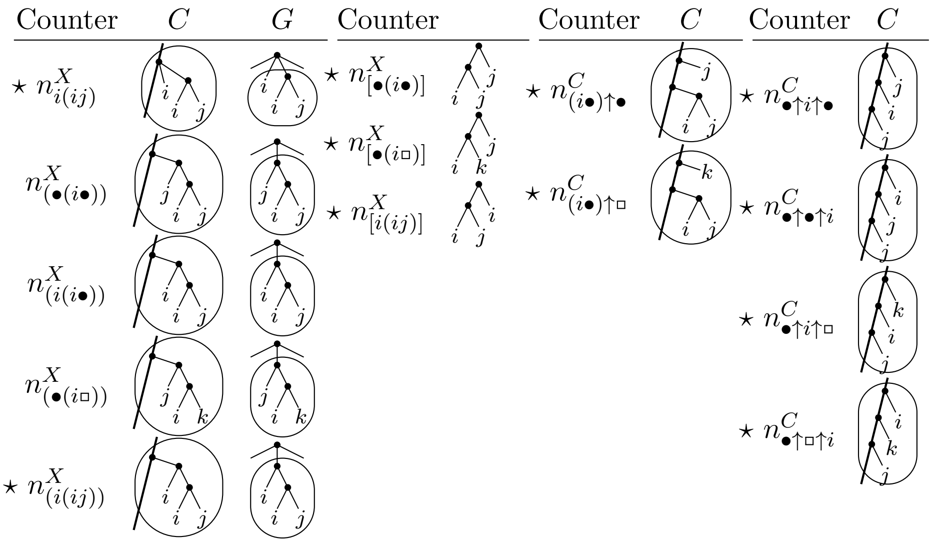
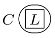
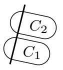
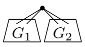
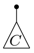
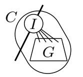

Counter
C
G
Counter
Counter
C
Counter
C
⋆ $n^X_{i(ij)}$
$n^X_{(\bullet(i\bullet))}$
$n^X_{(i(i\bullet))}$
$n^X_{(\bullet(i\square))}$
⋆ $n^X_{(i(ij))}$
⋆ $n^X_{[\bullet(i\bullet)]}$
⋆ $n^X_{[\bullet(i\square)]}$
⋆ $n^X_{[i(ij)]}$
⋆ $n^C_{(i\bullet)\uparrow\bullet}$
⋆ $n^C_{(i\bullet)\uparrow\square}$
⋆ $n^C_{\bullet\uparrow i\uparrow\bullet}$
⋆ $n^C_{\bullet\uparrow\bullet\uparrow i}$
⋆ $n^C_{\bullet\uparrow i\uparrow\square}$
⋆ $n^C_{\bullet\uparrow\square\uparrow i}$

Figure B.2: Illustration of the additional counters used to find B for our variation of the quartet distance calculation.

	 **L → C**	 **CC → C**	 **GG → G**	 **C → G**	 **IG → C**
Counter	$n^{C}_{\_}$	$n^{C}_{\_}$	$n^{G}_{\_}$	$n^{G}_{\_}$	$n^{C}_{\_}$
Common	0	$n^{C_1}_{\_} + n^{C_2}_{\_}$	$n^{G_1}_{\_} + n^{G_2}_{\_}$		$n^{G}_{\_}$
$n^{X}_{(i(\bullet\bullet))}$				$n^{C}_{[i(\bullet\bullet)]}$	
$n^{X}_{(i(\bullet\square))}$				$n^{C}_{[i(\bullet\square)]}$	
Common	0	$n^{C_1}_{\_} + n^{C_2}_{\_} + f\_(C_1, C_2)$	$n^{G_1}_{\_} + n^{G_2}_{\_} + g\_(G_1, G_2) + g\_(G_2, G_1)$	$n^{C}_{\_}$	$n^{G}_{\_}$
⋆ $n^{X}_{[i(\bullet\bullet)]}$		$\left(n^{C_1}_{[\bullet\bullet]} - \binom{n^{C_1}_i}{2}\right) n^{C_2}_i + \left(\sum_{j=1,j\neq i}^{d} n^{C_1}_j \cdot n^{C_2}_{j\uparrow i}\right) + n^{C_1}_i \left(n^{C_2}_{(\bullet\bullet)} - n^{C_2}_{(ii)}\right)$	$n^{G'}_i \left(n^{G''}_{(\bullet\bullet)} - n^{G''}_{(ii)}\right)$	$\rbrace = f\_(C_1, C_2)$	
⋆ $n^{X}_{[i(\bullet\square)]}$		$\left(n^{C_1}_{[\bullet\square]} - n^{C_1}_{[i\bullet]}\right) n^{C_2}_i + \left(\sum_{k=1,k\neq i}^{d} n^{C_1}_k (n^{C_2}_{\bullet\uparrow i} - n^{C_2}_{k\uparrow i})\right) + n^{C_1}_i \left(n^{C_2}_{(\bullet\square)} - n^{C_2}_{(i\bullet)}\right)$	$n^{G'}_i \left(n^{G''}_{(\bullet\square)} - n^{G''}_{(i\bullet)}\right)$	$g\_(G', G'')$	
Common	0	$n^{C_1}_{\_} + n^{C_2}_{\_} + f\_(C_1, C_2)$	not defined	not defined	0
⋆ $n^{C}_{j\uparrow i}$		$n^{C_1}_j \cdot n^{C_2}_i$			
⋆ $n^{C}_{i\uparrow j}$		$n^{C_1}_i \cdot n^{C_2}_j$			
⋆ $n^{C}_{i\uparrow\bullet\uparrow\bullet}$		$n^{C_1}_i \left(n^{C_2}_{\bullet\uparrow\bullet} - n^{C_2}_{i\uparrow i}\right) + \left(\sum_{j=1,j\neq i}^{d} n^{C_1}_{i\uparrow j} \cdot n^{C_2}_j\right)$			
⋆ $n^{C}_{i\uparrow\bullet\uparrow\square}$		$n^{C_1}_i \left(n^{C_2}_{\bullet\uparrow\square} - n^{C_2}_{i\uparrow\bullet} - n^{C_2}_{\bullet\uparrow i}\right) + \left(\sum_{j=1,j\neq i}^{d} n^{C_1}_{i\uparrow j} \left(n^{C_2}_{\bullet} - n^{C_2}_i - n^{C_2}_j\right)\right)$			
$n^{C}_{(\bullet\square)\uparrow i}$		$\left(n^{C_1}_{(\bullet\square)} - n^{C_1}_{(i\bullet)}\right) \cdot n^{C_2}_i$			

$$\begin{aligned} n^{X}_{(\bullet(\square\square))} &= \sum_{i=1}^{d} n^{X}_{(i(\bullet\bullet))} & n^{C}_{\bullet\uparrow\bullet} &= \sum_{i=1}^{d} n^{C}_{i\uparrow i} \\ n^{X}_{[\bullet\bullet]} &= \sum_{i=1}^{d} \binom{n^{X}_i}{2} & n^{C}_{\bullet\uparrow\square} &= \sum_{i=1}^{d} n^{C}_{i\uparrow\bullet} \\ n^{X}_{[\bullet(\square\square)]} &= \sum_{i=1}^{d} n^{X}_{[i(\bullet\bullet)]} & n^{C}_{(\bullet\bullet)\uparrow\square} &= \sum_{i=1}^{d} n^{C}_{(ii)\uparrow\bullet} \\ & & n^{C}_{\bullet\uparrow\square\uparrow\square} &= \sum_{i=1}^{d} n^{C}_{\bullet\uparrow i\uparrow i} \end{aligned}$$

Figure B.3: Additional counters used for the A calculation of our variation of the quartet distance calculation.

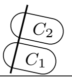
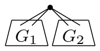
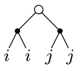
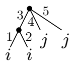
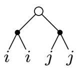
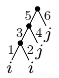
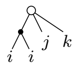
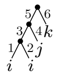
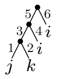
Case	T_1	T_2		**CC → C** 		**GG → G** 
$\alpha\beta$			1,2	$\sum_{i=1}^{d} n^{C_1}_i \cdot n^{C_2}_{i\uparrow\bullet\bullet}$	3:45	$\sigma\left[\sum_{i=1}^{d} n^{G'}_{(ii)} \left(n^{G''}_{\bullet\bullet} - n^{G''}_{ii}\right)\right]$
			3	$\sum_{i=1}^{d} \binom{n^{C_1}_i}{2} \left(n^{C_2}_{\bullet\bullet} - n^{C_2}_{ii}\right)$	34:5,35:4	$\sigma\left[\sum_{i=1}^{d} n^{G'}_{i(\bullet\bullet)} \cdot n^{G''}_i\right]$
			4,5	$\sum_{i=1}^{d} n^{C_1}_i \cdot n^{C_2}_{i(\bullet\bullet)}$		
$\alpha\gamma$			1,2	$\sum_{i=1}^{d} n^{C_1}_i \cdot n^{C_2}_{i\uparrow\bullet\uparrow\bullet}$	5:6	$\sigma\left[\sum_{i=1}^{d} n^{G'}_i \cdot n^{G''}_{(i(\bullet\bullet))}\right]$
			3	$\sum_{i=1}^{d} \binom{n^{C_1}_i}{2} \left(n^{C_2}_{\bullet\uparrow\bullet} - n^{C_2}_{i\uparrow i}\right)$		
			4	$\sum_{i=1}^{d} n^{C_1}_i \cdot n^{C_2}_{(\bullet\bullet)\uparrow i}$		
			5	$\sum_{i=1}^{d} n^{C_1}_{[i(\bullet\bullet)]} \cdot n^{C_2}_i$		
			6	$\sum_{i=1}^{d} n^{C_1}_i \cdot n^{C_2}_{(i(\bullet\bullet))}$		
$\beta\gamma$			1,2	$\sum_{i=1}^{d} n^{C_1}_i \cdot n^{C_2}_{i\uparrow\bullet\uparrow\square}$	5:6	$\sigma\left[\sum_{i=1}^{d} n^{G'}_i \left(n^{G''}_{(\bullet(\square\square))} - n^{G''}_{(i(\bullet\bullet))} - n^{G''}_{(\bullet(ii))}\right)\right]$
			3	$\sum_{i=1}^{d} \binom{n^{C_1}_i}{2} \left(n^{C_2}_{\bullet\uparrow\square} - n^{C_2}_{i\uparrow\bullet} - n^{C_2}_{\bullet\uparrow i}\right)$		
			4	$\sum_{i=1}^{d} n^{C_1}_i \left(n^{C_2}_{(\bullet\bullet)\uparrow\square} - n^{C_2}_{(ii)\uparrow\bullet} - n^{C_2}_{(\bullet\bullet)\uparrow i}\right)$		
			5	$\sum_{i=1}^{d} \left(n^{C_1}_{[\bullet(\square\square)]} - n^{C_1}_{[i(\bullet\bullet)]} - n^{C_1}_{[\bullet(ii)]}\right) n^{C_2}_i$		
			6	$\sum_{i=1}^{d} n^{C_1}_i \left(n^{C_2}_{(\bullet(\square\square))} - n^{C_2}_{(i(\bullet\bullet))} - n^{C_2}_{(\bullet(ii))}\right)$		
			1,2	$\sum_{i=1}^{d} n^{C_1}_i \left(n^{C_2}_{\bullet\uparrow\square\uparrow\square} - n^{C_2}_{i\uparrow\bullet\uparrow\bullet} - n^{C_2}_{\bullet\uparrow i\uparrow i}\right)$	5:6	$\sigma\left[\sum_{i=1}^{d} n^{G'}_i \cdot n^{G''}_{(i(\bullet\square))}\right]$
			3	$\sum_{i=1}^{d} \left(n^{C_1}_{[\bullet\square]} - n^{C_1}_{[i\bullet]}\right) \cdot n^{C_2}_{i\uparrow i}$		
			4	$\sum_{i=1}^{d} n^{C_1}_i \cdot n^{C_2}_{(\bullet\square)\uparrow i}$		
			5	$\sum_{i=1}^{d} n^{C_1}_{[i(\bullet\square)]} \cdot n^{C_2}_i$		
			6	$\sum_{i=1}^{d} n^{C_1}_i \cdot n^{C_2}_{(i(\bullet\square))}$		

Figure B.4: Additional sums for finding A for our variation of the quartet distance calculation.

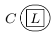
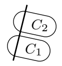
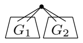
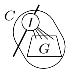
Counter	 **L → C** $n_{\_}^{C}$	 **CC → C** $n_{\_}^{C}$	 **GG → G** $n_{\_}^{G}$		**C → G** $n_{\_}^{G}$	 **IG → C** $n_{\_}^{C}$
Common	0	$n_{\_}^{C_1} + n_{\_}^{C_2}$	$n_{\_}^{G_1} + n_{\_}^{G_2} + g_{\_}(G_1, G_2) + g_{\_}(G_2, G_1)$		0	$n_{\_}^{G}$
$\star\ n_{i(ij)}^{X}$			$g_{-}(G', G'') = \{ \quad n_i^{G'} \cdot n_{(ij)}^{G''}$			
Common	0	$n_{\_}^{C_1} + n_{\_}^{C_2}$	$n_{\_}^{G_1} + n_{\_}^{G_2}$			$n_{\_}^{G}$
$n_{(\bullet(i\bullet))}^{X}$					$n_{[\bullet(i\bullet)]}^{X}$	
$n_{(i(i\bullet))}^{X}$					$n_{[i(i\bullet)]}^{X}$	
$n_{(\bullet(i\square))}^{X}$					$n_{[\bullet(i\square)]}^{X}$	
$\star\ n_{(i(ij))}^{X}$					$n_{[i(ij)]}^{X}$	
Common	0	$n_{\_}^{C_1} + n_{\_}^{C_2} + f_{\_}(C_1, C_2)$	$n_{\_}^{G_1} + n_{\_}^{G_2} + g_{\_}(G_1, G_2) + g_{\_}(G_2, G_1)$		$n_{\_}^{C}$	$n_{\_}^{G}$
$\star\ n_{[\bullet(i\bullet)]}^{X}$		$n_i^{C_1}\left(n_{\bullet\uparrow\bullet}^{C_2} - n_{i\uparrow i}^{C_2}\right) + \left(\sum_{j=1,j\neq i}^{d} n_i^{C_1} \cdot n_j^{C_1} \cdot n_j^{C_2} + n_j^{C_1} \cdot n_{(ij)}^{C_2} + n_j^{C_1} \cdot n_{i\uparrow j}^{C_2}\right)$	$\sum_{j=1,j\neq i}^{d} n_j^{G'} \cdot n_{(ij)}^{G''}$	$\left.\right\}$		
$\star\ n_{[\bullet(i\square)]}^{X}$		$n_i^{C_1}\left(n_{\bullet\uparrow\square}^{C_2} - n_{i\uparrow\bullet}^{C_2} - n_{\bullet\uparrow i}^{C_2}\right) + \left(\sum_{j=1,j\neq i}^{d} n_i^{C_1} \cdot n_j^{C_1} \cdot \left(n_{\bullet-i-j}^{C_2}\right) + \left(n_{\bullet-i-j}^{C_1}\right)\left(n_{(ij)}^{C_2} + n_{i\uparrow j}^{C_2}\right)\right)$	$\sum_{j=1,j\neq i}^{d}\left(n_{\bullet-i-j}^{G'}\right) n_{(ij)}^{G''}$	$= \quad f_{\_}(C_1, C_2)$ $g_{\_}(G', G'')$		
$\star\ n_{[i(ij)]}^{X}$		$n_i^{C_1} \cdot n_j^{C_1} \cdot n_i^{C_2} + n_i^{C_1} \cdot n_{(ij)}^{C_2} + n_i^{C_1} \cdot n_{j\uparrow i}^{C_2} + n_j^{C_1} \cdot n_{i\uparrow i}^{C_2}$	$n_i^{G'} \cdot n_{(ij)}^{G''}$			
Common	0	$n_{\_}^{C_1} + n_{\_}^{C_2} + f_{\_}(C_1, C_2)$	not defined		not defined	0
$\star\ n_{\bullet\uparrow i\uparrow\bullet}^{C}$		$\sum_{j=1,j\neq i}^{d} n_j^{C_1} \cdot n_{i\uparrow j}^{C_2} + n_{j\uparrow i}^{C_1} \cdot n_j^{C_2}$				
$\star\ n_{\bullet\uparrow\bullet\uparrow i}^{C}$		$\left(\sum_{j=1,j\neq i}^{d} n_j^{C_1} \cdot n_{j\uparrow i}^{C_2}\right) + \left(n_{\bullet\uparrow\bullet}^{C_1} - n_{i\uparrow i}^{C_1}\right) n_i^{C_2}$				
$\star\ n_{(i\bullet)\uparrow\bullet}^{C}$		$\sum_{j=1,j\neq i}^{d} n_{(ij)}^{C_1} \cdot n_j^{C_2}$				
$\star\ n_{\bullet\uparrow i\uparrow\square}^{C}$		$\sum_{j=1,j\neq i}^{d} n_j^{C_1}\left(n_{i\uparrow\bullet}^{C_2} - n_{i\uparrow j}^{C_2}\right) + n_{j\uparrow i}^{C_1}\left(n_{\bullet-i-j}^{C_2}\right)$				
$\star\ n_{(i\bullet)\uparrow\square}^{C}$		$\sum_{j=1,j\neq i}^{d} n_{(ij)}^{C_1}\left(n_{\bullet-i-j}^{C_2}\right)$				
$\star\ n_{\bullet\uparrow\square\uparrow i}^{C}$		$\left(\sum_{j=1,j\neq i}^{d} n_j^{C_1}\left(n_{\bullet\uparrow i}^{C_2} - n_{j\uparrow i}^{C_2}\right)\right) + \left(n_{\bullet\uparrow\square}^{C_1} - n_{\bullet\uparrow i}^{C_1} - n_{i\uparrow\bullet}^{C_1}\right) n_i^{C_2}$				

Figure B.5: Additional counters used for the B calculation of our variation of the quartet distance calculation.

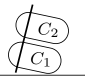
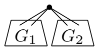
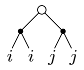
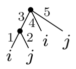
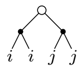
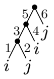
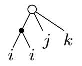
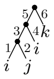
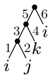
Case	T_1	T_2			**CC → C** 			**GG → G** 
$\alpha\beta$			1,2		$\sum_{i=1}^{d} n_i^{C_1} \cdot n_{\bullet\uparrow i\bullet}^{C_2}$	3:45	$\star$	$\sigma\left[\sum_{i=1}^{d-1}\sum_{j=i+1}^{d} n_{(ij)}^{G'} n_{ij}^{G''}\right]$
			3	$\star$	$\sum_{i=1}^{d-1}\sum_{j=i+1}^{d} n_i^{C_1} \cdot n_j^{C_1} \cdot n_{ij}^{C_2}$	34:5,35:4		$\sigma\left[\sum_{i=1}^{d} n_{\bullet(i\bullet)}^{G'} n_i^{G''}\right]$
			4,5		$\sum_{i=1}^{d} n_i^{C_1} \cdot n_{\bullet(i\bullet)}^{C_2}$			
$\alpha\gamma$			1		$\sum_{i=1}^{d} n_i^{C_1} \cdot n_{\bullet\uparrow i\uparrow\bullet}^{C_2}$	5:6		$\sigma\left[\sum_{j=1}^{d} n_j^{G'} \cdot n_{(\bullet(j\bullet))}^{G''}\right]$
			2		$\sum_{j=1}^{d} n_j^{C_1} \cdot n_{\bullet\uparrow\bullet\uparrow j}^{C_2}$			
			3	$\star$	$\sum_{i=1}^{d}\sum_{j=1,j\neq i}^{d} n_i^{C_1} \cdot n_j^{C_1} \cdot n_{i\uparrow j}^{C_2}$			
			4		$\sum_{i=1}^{d} n_i^{C_1} \cdot n_{(i\bullet)\uparrow\bullet}^{C_2}$			
			5		$\sum_{j=1}^{d} n_{[\bullet(j\bullet)]}^{C_1} \cdot n_j^{C_2}$			
			6		$\sum_{j=1}^{d} n_j^{C_1} \cdot n_{(\bullet(j\bullet))}^{C_2}$			
$\beta\gamma$			1		$\sum_{i=1}^{d} n_i^{C_1} \cdot n_{\bullet\uparrow i\uparrow\square}^{C_2}$	5:6		$\sigma\left[\sum_{k=1}^{d} n_k^{G'}\left(n_{(\bullet(\bullet\square))}^{G''} - n_{(k(k\bullet))}^{G''} - n_{(\bullet(k\bullet))}^{G''}\right)\right]$
			2		$\sum_{j=1}^{d} n_j^{C_1}\left(n_{\bullet\uparrow\bullet\uparrow\square}^{C_2} - n_{j\uparrow j\uparrow\bullet}^{C_2} - n_{\bullet\uparrow\bullet\uparrow j}^{C_2}\right)$			
			3	$\star$	$\sum_{i=1}^{d}\sum_{j=1,j\neq i}^{d} n_i^{C_1} \cdot n_j^{C_1}\left(n_{i\uparrow\bullet}^{C_2} - n_{i\uparrow j}^{C_2}\right)$			
			4		$\sum_{i=1}^{d} n_i^{C_1} \cdot n_{(i\bullet)\uparrow\square}^{C_2}$			
			5	$\star$	$\sum_{i=1}^{d}\sum_{j=1,j\neq i}^{d} n_{[i(ij)]}^{C_1}\left(n_{\bullet}^{C_2} - n_i^{C_2} - n_j^{C_2}\right)$			
			6	$\star$	$\sum_{i=1}^{d}\sum_{k=1,k\neq i}^{d} n_k^{C_1}\left(n_{(i(i\bullet))}^{C_2} - n_{(i(ik))}^{C_2}\right)$			
			1		$\sum_{i=1}^{d} n_i^{C_1} \cdot n_{\bullet\uparrow\square\uparrow i}^{C_2}$	5:6		$\sigma\left[\sum_{i=1}^{d} n_i^{G'} \cdot n_{(\bullet(i\square))}^{G''}\right]$
			2		$\sum_{j=1}^{d} n_j^{C_1}\left(n_{\bullet\uparrow\square\uparrow\bullet}^{C_2} - n_{j\uparrow\bullet\uparrow j}^{C_2} - n_{\bullet\uparrow j\uparrow\bullet}^{C_2}\right)$			
			3	$\star$	$\sum_{i=1}^{d}\sum_{j=1,j\neq i}^{d} n_i^{C_1} \cdot n_j^{C_1}\left(n_{\bullet\uparrow i}^{C_2} - n_{j\uparrow i}^{C_2}\right)$			
			4		$\sum_{k=1}^{d} n_k^{C_1}\left(n_{(\bullet\square)\uparrow\bullet}^{C_2} - n_{(k\bullet)\uparrow k}^{C_2} - n_{(k\bullet)\uparrow\bullet}^{C_2}\right)$			
			5		$\sum_{i=1}^{d} n_{[\bullet(i\square)]}^{C_1} \cdot n_i^{C_2}$			
			6		$\sum_{i=1}^{d} n_i^{C_1} \cdot n_{(\bullet(i\square))}^{C_2}$			

Figure B.6: Additional sums for finding B for our variation of the quartet distance calculation.

Appendix C

Counters and Sums for Calculating E

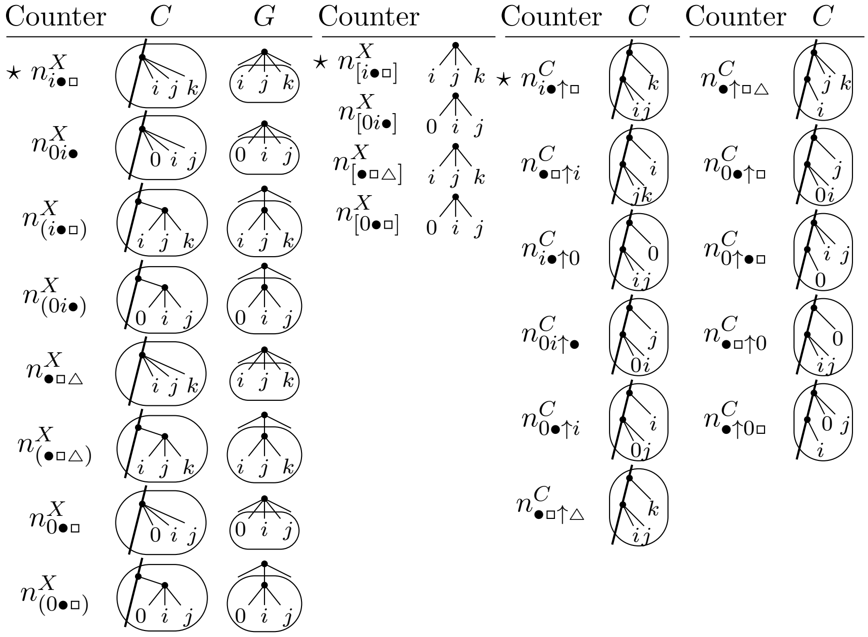
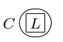
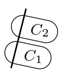
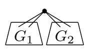
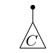
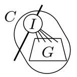

Counter
C
G
$\star\ n^X_{i\bullet\square}$
$n^X_{0i\bullet}$
$n^X_{(i\bullet\square)}$
$n^X_{(0i\bullet)}$
$n^X_{\bullet\square\triangle}$
$n^X_{(\bullet\square\triangle)}$
$n^X_{0\bullet\square}$
$n^X_{(0\bullet\square)}$
$\star\ n^X_{[i\bullet\square]}$
$n^X_{[0i\bullet]}$
$n^X_{[\bullet\square\triangle]}$
$n^X_{[0\bullet\square]}$
$\star\ n^C_{i\bullet\uparrow\square}$
$n^C_{\bullet\square\uparrow i}$
$n^C_{i\bullet\uparrow 0}$
$n^C_{0i\uparrow\bullet}$
$n^C_{0\bullet\uparrow i}$
$n^C_{\bullet\square\uparrow\triangle}$
$n^C_{\bullet\uparrow\square\triangle}$
$n^C_{0\bullet\uparrow\square}$
$n^C_{0\uparrow\bullet\square}$
$n^C_{\bullet\square\uparrow 0}$
$n^C_{\bullet\uparrow 0\square}$
i j k
0 i j
ij
jk
0i
0j
0

Figure C.1: Illustration of the additional counters used to find E for the quartet distance calculation.

	 **L → C**	 **CC → C**	 **GG → G**	 **C → G**	 **IG → C**
Counter	n^C_{-}	n^C_{-}	n^G_{-}	n^G_{-}	n^C_{-}
Common	0	$n^{C_1}_{-}+n^{C_2}_{-}$	$n^{G_1}_{-}+n^{G_2}_{-}+g\_(G_1,G_2)+g\_(G_2,G_1)$	0	n^G_{-}
$\star\ n^X_{i\bullet\square}$			$g_{-}(G',G'')=\Big\{\ \left(\sum_{j=1,j\neq i}^d n^{G'}_{ij}\cdot n^{G''}_{\bullet-i-j}\right)+n^{G'}_i\left(n^{G''}_{\bullet\square}-n^{G''}_{i\bullet}\right)$		
$n^X_{0i\bullet}$			$n^{G'}_{0i}\left(n^{G''}_{\bullet}-n^{G''}_{i}\right)+n^{G'}_i\left(n^{G''}_{0\bullet}-n^{G''}_{0i}\right)+n^{G'}_{i\bullet}\cdot n^{G''}_0$		
Common	0	$n^{C_1}_{-}+n^{C_2}_{-}$	$n^{G_1}_{-}+n^{G_2}_{-}$		n^G_{-}
$n^X_{(i\bullet\square)}$				$n^C_{[i\bullet\square]}$	
$n^X_{(0i\bullet)}$				$n^C_{[0i\bullet]}$	
Common	0	$n^{C_1}_{-}+n^{C_2}_{-}+f\_(C_1,C_2)$	$n^{G_1}_{-}+n^{G_2}_{-}+g\_(G_1,G_2)+g\_(G_2,G_1)$	n^C_{-}	n^G_{-}
$\star\ n^X_{[i\bullet\square]}$		$n^{C_1}_i\left(n^{C_2}_{\bullet\square}-n^{C_2}_{i\bullet}\right)+\sum_{j=1,j\neq i}^d n^{C_1}_{\bullet-i-j}\cdot n^{C_2}_{ij}$	$\left(\sum_{j=1,j\neq i}^d n^{G'}_{ij}\cdot n^{G''}_{\bullet-i-j}\right)+n^{G'}_i\left(n^{G''}_{\bullet\square}-n^{G''}_{i\bullet}\right)$	$\Big\}=$	$f\_(C_1,C_2)$
$n^X_{[0i\bullet]}$		$n^{C_1}_0\cdot n^{C_2}_{i\bullet}+n^{C_1}_i\left(n^{C_2}_{0\bullet}-n^{C_2}_{0i}\right)+\left(n^{C_1}_{\bullet}-n^{C_1}_i\right)n^{C_2}_{0i}$	$n^{G'}_{0i}\left(n^{G''}_{\bullet}-n^{G''}_{i}\right)+n^{G'}_i\left(n^{G''}_{0\bullet}-n^{G''}_{0i}\right)+n^{G'}_{i\bullet}\cdot n^{G''}_0$		$g\_(G',G'')$
Common	0	$n^{C_1}_{-}+n^{C_2}_{-}+f\_(C_1,C_2)$	not defined	not defined	0
$\star\ n^C_{i\bullet\uparrow\square}$		$\sum_{j=1,j\neq i}^d n^{C_1}_{ij}\cdot n^{C_2}_{\bullet-i-j}$			
$n^C_{\bullet\square\uparrow i}$		$\left(n^{C_1}_{\bullet\square}-n^{C_1}_{i\bullet}\right)n^{C_2}_i$			
$n^C_{i\bullet\uparrow 0}$		$n^{C_1}_{i\bullet}\cdot n^{C_2}_0$			
$n^C_{0i\uparrow\bullet}$		$n^{C_1}_{0i}\left(n^{C_2}_{\bullet}-n^{C_2}_i\right)$			
$n^C_{0\bullet\uparrow i}$		$\left(n^{C_1}_{0\bullet}-n^{C_1}_{0i}\right)n^{C_2}_i$			

$$n^X_{\bullet\square\triangle}=\tfrac{1}{3}\sum_{i=1}^d n^X_{i\bullet\square}\qquad n^X_{(0\bullet\square)}=\tfrac{1}{2}\sum_{i=1}^d n^X_{(0i\bullet)}\qquad n^C_{\bullet\square\uparrow\triangle}=\sum_{i=1}^d n^C_{\bullet\square\uparrow i}\qquad n^C_{0\uparrow\bullet\square}=\tfrac{1}{2}\sum_{i=1}^d n^C_{0\uparrow i\bullet}$$

$$n^X_{(\bullet\square\triangle)}=\tfrac{1}{3}\sum_{i=1}^d n^X_{(i\bullet\square)}\qquad n^X_{[\bullet\square\triangle]}=\tfrac{1}{3}\sum_{i=1}^d n^X_{[i\bullet\square]}\qquad n^C_{\bullet\uparrow\square\triangle}=\sum_{i=1}^d n^C_{i\uparrow\bullet\square}\qquad n^C_{\bullet\square\uparrow 0}=\tfrac{1}{2}\sum_{i=1}^d n^C_{i\bullet\uparrow 0}$$

$$n^X_{0\bullet\square}=\tfrac{1}{2}\sum_{i=1}^d n^X_{0i\bullet}\qquad n^X_{[0\bullet\square]}=\tfrac{1}{2}\sum_{i=1}^d n^X_{[0i\bullet]}\qquad n^C_{0\bullet\uparrow\square}=\sum_{i=1}^d n^C_{0i\uparrow\bullet}\qquad n^C_{\bullet\uparrow 0\square}=\sum_{i=1}^d n^C_{i\uparrow 0\bullet}$$

Figure C.2: Additional counters used for the E calculation of the quartet distance calculation. Additionally the counters n^X_{ij} and $n^C_{0\uparrow i\bullet}$ from the B calculation are used.

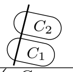
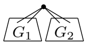
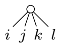
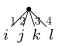
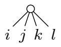
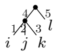
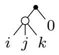
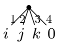
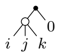
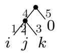
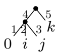
Case	T_1	T_2		$\mathbf{CC} \to \mathbf{C}$ 		$\mathbf{GG} \to \mathbf{G}$ 
$\delta\delta$			1,2,3,4	$\sum_{i=1}^{d} n_i^{C_1} \left(n^{C_2}_{\bullet\square\triangle} - n^{C_2}_{i\bullet\square}\right)$	1:234	$\sigma\left[\sum_{i=1}^{d} n_i^{G'} \left(n^{G''}_{\bullet\square\triangle} - n^{G''}_{i\bullet\square}\right)\right]$
					12:34	$\star \sum_{i=1}^{d-1} \sum_{j=i+1}^{d} n_{ij}^{G_1} \left(n^{G_2}_{\bullet\square} - n^{G_2}_{i\bullet} - n^{G_2}_{j\bullet} + n^{G_2}_{ij}\right)$
$\delta\epsilon$			1,2,3	$\sum_{i=1}^{d} n_i^{C_1} \left(n^{C_2}_{\bullet\square\uparrow\triangle} - n^{C_2}_{i\bullet\uparrow\square} - n^{C_2}_{\bullet\square\uparrow i}\right)$	1234:5	$\sigma\left[\sum_{i=1}^{d} \left(n^{G'}_{(\bullet\square\triangle)} - n^{G'}_{(i\bullet\square)}\right) n_i^{G''}\right]$
			4	$\sum_{i=1}^{d} \left(n^{C_1}_{[\bullet\square\triangle]} - n^{C_1}_{[i\bullet\square]}\right) n_i^{C_2}$		
			5	$\sum_{i=1}^{d} n_i^{C_1} \left(n^{C_2}_{(\bullet\square\triangle)} - n^{C_2}_{(i\bullet\square)}\right)$		
$\epsilon\delta$			1,2,3	$\sum_{i=1}^{d} n_i^{C_1} \left(n^{C_2}_{0\bullet\square} - n^{C_2}_{0i\bullet}\right)$	1:234	$\sigma\left[\sum_{i=1}^{d} n_i^{G'} \left(n^{G''}_{0\bullet\square} - n^{G''}_{0i\bullet}\right)\right]$
			4	$n_0^{C_1} \cdot n^{C_2}_{\bullet\square\triangle}$	4:123	$\sigma\left[n_0^{G'} \cdot n^{G''}_{\bullet\square\triangle}\right]$
					14:23	$\sigma\left[\sum_{i=1}^{d} n_{0i}^{G'} \left(n^{G''}_{\bullet\square} - n^{G''}_{i\bullet}\right)\right]$
$\epsilon\epsilon$			1,2,3	$\sum_{i=1}^{d} n_i^{C_1} \left(n^{C_2}_{\bullet\square\uparrow 0} - n^{C_2}_{i\bullet\uparrow 0}\right)$	1234:5	$\sigma\left[n^{G'}_{(\bullet\square\triangle)} \cdot n_0^{G''}\right]$
			4	$n^{C_1}_{[\bullet\square\triangle]} \cdot n_0^{C_2}$		
			5	$n_0^{C_1} \cdot n^{C_2}_{(\bullet\square\triangle)}$		
			1	$n_0^{C_1} \cdot n^{C_2}_{\bullet\square\uparrow\triangle}$	1234:5	$\sigma\left[\sum_{i=1}^{d} \left(n^{G'}_{(0\bullet\square)} - n^{G'}_{(0i\bullet)}\right) n_i^{G''}\right]$
			2,3	$\sum_{i=1}^{d} n_i^{C_1} \left(n^{C_2}_{0\bullet\uparrow\square} - n^{C_2}_{0i\uparrow\bullet} - n^{C_2}_{0\bullet\uparrow i}\right)$		
			4	$\sum_{i=1}^{d} \left(n^{C_1}_{[0\bullet\square]} - n^{C_1}_{[0i\bullet]}\right) n_i^{C_2}$		
			5	$\sum_{i=1}^{d} n_i^{C_1} \left(n^{C_2}_{(0\bullet\square)} - n^{C_2}_{(0i\bullet)}\right)$		

Figure C.3: New sums for finding E for the quartet distance calculation.

Appendix D

Raw Runtime Data

	100	158	251	398	630	1,000	1,584	2,511	3,981	6,309	10,000	15,848	25,118	39,810	63,095	100,000	158,489	251,188	398,107	630,957	1,000,000
Balanced, random, contract enabled																					
$d_1 = 2$, $d_2 = 2$, B	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.4	0.6	1.1	1.9	3.3	5.7	9.9	16.8	29.0	49.2	82.7	140.2
$d_1 = 2$, $d_2 = 2$, E	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.4	0.8	1.3	2.3	4.1	7.1	12.3	21.5	37.1	62.3	105.5
$d_1 = 2$, $d_2 = 1024$, B	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.5	0.9	1.5	2.4	4.1	6.9	11.4	19.3	31.7	51.9
$d_1 = 2$, $d_2 = 1024$, E	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.2	0.4	0.6	1.1	1.8	3.1	5.1	8.5	14.6	24.1	39.9
$d_1 = 16$, $d_2 = 16$, B	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.4	0.6	1.0	1.7	2.8	5.1	8.8	14.7	24.0	39.1	63.9
$d_1 = 16$, $d_2 = 16$, E	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.6	0.9	1.5	2.9	5.2	8.7	14.2	23.4	38.1
$d_1 = 256$, $d_2 = 256$, B	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.2	0.4	0.6	1.1	1.8	3.2	6.2	10.6	17.5	27.3	43.7	68.7	112.0	
$d_1 = 256$, $d_2 = 256$, E	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.4	0.7	1.2	2.2	3.9	6.6	10.8	17.8	28.0	45.3	72.7

Table D.1: B vs E for the quartet distance calculation. Columns are the number of leaves and all runtimes are in seconds.

	100	158	251	398	630	1,000	1,584	2,511	3,981	6,309	10,000	15,848	25,118	39,810	63,095	100,000	158,489	251,188	398,107	630,957	1,000,000
Balanced, random																					
Balanced, random, new runs, $d = 2$																					
[1], contract enabled	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.3	0.5	0.8	1.5	2.7	4.6	8.1	14.2	24.9	43.3	73.9	125.1	213.5
[1], contract disabled	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.2	0.3	0.6	1.0	1.9	3.5	6.2	11.6	20.8	37.0	66.6	119.0	207.6	367.9
Contract enabled, B	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.2	0.4	0.7	1.2	2.0	3.4	5.8	9.9	16.8	29.1	49.2	83.2	139.7
Contract enabled, E	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.2	0.3	0.4	0.8	1.4	2.4	4.2	7.1	12.2	21.7	36.9	62.3	104.6
Contract disabled, B	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.3	0.5	0.8	1.5	2.7	4.6	8.4	15.1	25.8	46.2	81.7	138.8	245.4
Contract disabled, E	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.5	1.0	1.9	3.1	5.7	10.3	18.0	32.5	58.8	100.8	178.2
Balanced, random, new runs, $d = 16$																					
[1], contract enabled	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.3	0.5	0.8	1.3	2.2	3.6	6.8	12.2	20.4	33.4	55.2	89.6
[1], contract disabled	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.5	0.9	1.6	2.7	4.5	8.7	15.9	27.4	45.8	76.5	127.2
Contract enabled, B	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.2	0.4	0.6	1.0	1.7	2.8	5.1	8.8	14.8	24.0	39.1	63.8
Contract enabled, E	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.6	0.9	1.5	2.9	5.2	8.7	14.3	23.4	38.3
Contract disabled, B	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.4	0.8	1.3	2.2	3.5	6.7	11.8	20.0	33.2	56.3	94.3
Contract disabled, E	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.4	0.7	1.2	2.1	3.9	7.4	12.6	21.1	35.9	59.1
Balanced, random, new runs, $d = 256$																					
[1], contract enabled	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.5	0.8	1.3	2.1	3.8	7.2	12.4	20.7	33.0	53.8	85.3	139.4	228.0
[1], contract disabled	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.5	0.8	1.3	2.2	3.9	7.3	14.2	24.0	38.2	62.9	101.7	165.9	266.6
Contract enabled, B	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.2	0.4	0.6	1.1	1.8	3.3	6.2	10.6	17.5	27.3	43.7	68.7	111.9	
Contract enabled, E	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.4	0.7	1.2	2.2	3.9	6.6	10.8	17.8	28.0	45.2	72.6
Contract disabled, B	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.2	0.4	0.6	1.1	1.8	3.3	6.2	11.8	20.2	31.5	50.9	81.1	131.2	
Contract disabled, E	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.2	0.3	0.4	0.7	1.2	2.3	4.3	7.5	13.2	22.5	36.7	58.8	94.0
Balanced, random, new runs, $d_1 = 2$, $d_2 = 1024$																					
[1], contract enabled	0.0	0.0	0.0	0.1	0.1	0.3	0.5	0.9	1.5	2.4	3.9	6.2	10.1	16.3	26.2	42.6					
[1], contract disabled	0.0	0.0	0.0	0.1	0.1	0.3	0.5	0.9	1.5	2.5	4.0	6.5	10.7	17.4	28.1	46.7					
Contract enabled, B	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.5	0.9	1.5	2.4	4.1	6.9	11.4	19.2	31.7	52.0
Contract enabled, E	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.4	0.6	1.0	1.8	3.1	5.1	8.5	14.5	24.0	40.1
Contract disabled, B	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.3	0.5	0.8	1.5	2.7	4.7	8.5	14.8	25.5	46.3	78.6	135.3
Contract disabled, E	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.6	1.1	2.0	3.5	6.4	11.3	19.7	35.8	62.0	106.9

Table D.2: More runtimes for B vs E for the quartet distance calculation. Columns are the number of leaves and all runtimes are in seconds.

	100	158	251	398	630	1,000	1,584	2,511	3,981	6,309	10,000	15,848	25,118	39,810	63,095	100,000	158,489	251,188	398,107	630,957	1,000,000
Balanced, random																					
Contract enabled	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.4	0.6	1.1	2.0	3.4	5.8	10.0	16.8	29.2	49.7	83.2	139.7
Contract disabled	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.2	0.4	0.8	1.5	2.7	4.6	8.4	15.1	25.9	46.5	81.8	139.7	246.3
[1], contract enabled	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.2	0.3	0.5	0.8	1.5	2.7	4.7	8.2	14.4	25.2	43.7	74.7	126.4	215.1
[1], contract disabled	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.6	1.0	1.9	3.5	6.2	11.6	20.8	37.0	67.5	119.1	207.1	366.9
$O(n^{2.688})$ [13]	0.0	0.0	0.0	0.1	0.2	0.6	1.5	3.8	9.6	24.5	62.7										
$O(n \cdot \lg^2 n)$ [16]	0.1	0.2	0.3	0.6	1.0	1.7	3.0	5.4	9.3	17.3	29.4										
Balanced, leaf moved																					
Contract enabled	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.6	1.0	1.6	2.7	4.6	8.0	13.4	22.3	37.0
[1], contract enabled	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.5	0.8	1.4	2.4	4.1	7.1	12.5	20.9	35.5	59.3
$O(n^{2.688})$ [13]	0.0	0.0	0.0	0.1	0.2	0.6	1.5	3.8	9.6	24.6	62.8										
$O(n \cdot \lg^2 n)$ [16]	0.0	0.1	0.1	0.2	0.3	0.4	0.8	1.2	2.1	3.7	5.9										
75% left-biased, random																					
Contract enabled	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.6	1.0	1.7	2.9	5.0	8.6	15.2	26.2	44.2	77.0	129.4
[1], contract enabled	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.4	0.7	1.2	2.2	3.8	6.7	11.5	21.0	36.9	63.0	112.0	193.2
T_1 balanced, T_2 99% leaf-biased																					
Contract enabled	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.6	1.0	1.7	3.1	5.3	9.5	16.4	28.3	48.5	80.9	134.5
99% left-biased, random																					
Contract enabled	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.4	0.7	1.2	2.1	3.6	6.4	28.2	64.3	115.1	201.5	333.1
Contract disabled	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.5	0.8	1.4	2.4	4.3	7.7	13.4	23.0	39.4	66.2	110.1
[1], contract enabled	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.5	0.8	1.4	2.5	4.3	7.7	44.1	105.6	187.6	333.4	551.5
[1], contract disabled	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.5	0.9	1.6	2.8	5.1	9.3	16.7	29.5	51.0	85.9	145.1

Table D.3: Quartet distance calculation for binary trees. Columns are the number of leaves and all runtimes are in seconds.

	100	158	251	398	630	1,000	1,584	2,511	3,981	6,309	10,000	15,848	25,118	39,810	63,095	100,000	158,489	251,188	398,107	630,957	1,000,000
Balanced, random, different values for d, contract enabled																					
$d = 2$	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.4	0.6	1.1	2.0	3.4	5.8	10.0	16.8	29.2	49.7	83.2	139.7
$d = 4$	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.2	0.4	0.7	1.3	2.3	3.7	6.5	11.4	18.5	32.0	55.3	89.5
$d = 8$	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.2	0.4	0.6	1.0	1.7	3.1	5.4	8.7	14.1	25.4	43.9	71.5
$d = 16$	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.2	0.4	0.6	1.0	1.7	2.8	5.0	8.8	14.7	23.9	39.0	63.5
$d = 32$	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.2	0.4	0.6	1.1	2.1	3.4	5.7	9.3	15.1	25.3	43.1	72.5
$d = 64$	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.5	0.9	1.4	2.4	4.1	7.0	12.0	20.3	34.4	56.0	90.1
$d = 128$	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.2	0.4	0.8	1.4	2.3	3.7	6.0	9.6	15.9	26.4	44.4	76.3	130.4
$d = 256$	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.2	0.4	0.6	1.1	1.8	3.2	6.1	10.5	17.2	26.7	42.8	67.1	109.4	
$d = 512$	0.0	0.0	0.0	0.1	0.1	0.2	0.2	0.4	0.6	1.0	1.7	2.7	4.6	8.2	13.7	24.7	47.5	81.8			
$d = 1024$	0.0	0.0	0.0	0.1	0.1	0.3	0.5	0.7	1.2	1.9	3.0	4.8	7.8	12.8	21.2	36.2	63.4				
$d_1 = 2$, $d_2 = 8$	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.3	0.5	0.8	1.4	2.5	4.7	8.0	13.3	21.6	40.8	69.9	113.6
$d_1 = 8$, $d_2 = 2$	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.5	0.9	1.5	2.5	4.2	7.2	12.1	20.5	33.9	57.2	96.9
$d_1 = 2$, $d_2 = 1024$	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.5	0.9	1.5	2.5	4.3	7.2	11.9	20.0	33.0	52.9
Balanced, random, $d_1 = 2$, $d_2 = 1024$																					
Contract Enabled	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.5	0.9	1.5	2.4	4.2					
Contract Disabled	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.5	0.8	1.5	2.7	4.6	8.5					
[1], contract enabled	0.0	0.0	0.0	0.1	0.1	0.3	0.5	0.9	1.5	2.4	3.9	6.3	10.2	16.4	26.3	42.7					
[1], contract disabled	0.0	0.0	0.0	0.1	0.1	0.3	0.5	0.9	1.5	2.5	4.0	6.6	10.8	17.5	28.3	46.7					
Balanced, random, $d = 8$																					
Contract enabled	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.2	0.4										
Contract disabled	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.2	0.4										
$O(n^{2.688})$ [13]	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.8	1.5	9.4	24.7										
Balanced, random, $d = 1024$																					
Contract enabled	0.0	0.0	0.0	0.1	0.1	0.3	0.5	0.7	1.2	1.8	2.9										
Contract disabled	0.0	0.0	0.0	0.1	0.1	0.3	0.5	0.8	1.2	1.9	3.1										
$O(n^{2.688})$ [13]	0.0	0.0	0.0	0.0	0.0	0.1	0.4	1.0	1.6	3.2	7.1										

Table D.4: Quartet distance calculation between arbitrary degree trees. Columns are the number of leaves and all runtimes are in seconds.

	100	158	251	398	630	1,000	1,584	2,511	3,981	6,309	10,000	15,848	25,118	39,810	63,095	100,000	158,489	251,188	398,107	630,957	1,000,000
Balanced, random																					
Contract enabled	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.6	1.1	2.0	3.5	6.2	11.2	19.9	34.0	58.6
Contract disabled	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.3	0.5	0.9	1.8	3.3	6.1	11.5	20.8	36.7	67.8
$O(n \cdot \lg^2 n)$ [15]	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.5	1.0	1.9	3.5	6.4	11.9	21.3	38.3
Balanced, leaf-moved																					
Contract enabled	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.2	0.4	0.7	1.2	2.1	3.8	6.2	10.7	18.3
Contract disabled	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.5	0.9	1.5	2.5	4.2	7.0	12.0
$O(n \cdot \lg^2 n)$ [15]	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.6	1.0	1.7	3.0	5.1	8.8	14.8
99% left-biased, random																					
Contract enabled	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.5	0.8	1.5	12.8	32.3	57.4	102.3	163.9
Contract disabled	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.4	0.7	1.3	2.4	4.4	8.0	14.2	24.8

Table D.5: Triplet distance calculation for binary trees. Columns are the number of leaves and all runtimes are in seconds.

	100	158	251	398	630	1,000	1,584	2,511	3,981	6,309	10,000	15,848	25,118	39,810	63,095	100,000	158,489	251,188	398,107	630,957	1,000,000
Balanced, random, different values for d, contract enabled																					
$d = 2$	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.6	1.1	2.0	3.5	6.2	11.2	19.9	34.0	58.6
$d = 4$	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.4	0.7	1.2	2.2	4.0	6.5	11.9	21.8	34.9
$d = 16$	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	0.6	1.3	2.4	3.9	6.4	10.8	17.9
$d = 128$	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.2	0.3	0.5	0.8	1.4	2.3	4.0	7.0	11.8
$d = 256$	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.2	0.4	0.8	1.5	2.5	4.2	6.9	12.0
$d = 512$	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.2	0.4	0.6	1.1	1.8	3.7	6.9	12.2
$d = 1024$	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.2	0.4	0.6	1.1	1.8	3.1	5.2	8.9
$d_1 = 2$, $d_2 = 1024$	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.3	0.5	0.8	1.4	2.5	4.2	7.3	12.3	21.0
$d_1 = 1024$, $d_2 = 2$	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.2	0.4	0.7	1.2	1.9	3.3	5.5	9.3	15.8

Table D.6: Triplet distance calculation between arbitrary degree trees. Columns are the number of leaves and all runtimes are in seconds.

Appendix E

Guide to the Implementation

To secure availability, the source code has been uploaded to a number of pages on the internet.

The code should be available from one of these locations:

- `http://cs.au.dk/~jensjoha/thesis`
- `http://cs.au.dk/~mholt/thesis`
- `http://www.zsoft.dk/thesis`
- `http://www.t-hawk.com/thesis`

Compilation and Usage

The `Makefile` specifies `-m64`, meaning that all compiles will result in 64-bit programs. This is only useful when your compiler supports it, and you intend to run the program on a 64-bit system.

To instead compile a 32-bit version, manually edit the file `Makefile` to state `-m32` in the two places (`CFLAGS` and `LDFLAGS` at the top) where it currently specifies `-m64`. As noted below, you will further have to specify `NO_N4_128=1` for all compiles.

Compilation

Many different variations are supported. The combination is specified at compile-time. We here list a number of useful commands.

- Triplets:
  `make CONTRACT_NUM=20000`
- Quartets, [1]:
  `make QUARTETS=1 CONTRACT_NUM=20000`
- Quartets, our variation, calculating A and B:
  `make QUARTETS=1 NOSWAP=1 CONTRACT_NUM=20000`
- Quartets, our variation, calculating A and E:
  `make QUARTETS=1 NOSWAP=1 CONTRACT_NUM=20000 CALCE=1`
- Furthermore note that
  - `CONTRACT_NUM=<?>` denotes Q. 20,000 has been specified in the commands above as this was found to perform the best, see Section 3.2.1.
  - Contract can be disabled by specifying `NOEXTRACT=1`.

- 128-bit integers are used by default for n^4 sums but can be disabled with `NO_N4_128=1`. Note that 128-bit integers only works for 64-bit compiles via `gcc` and similar. By disabling 128-bit integers, only 64-bit integers are used, which limits the size of the input, the program can meaningfully process, see Section 3.5.

Running the Program

- All compiles can calculate the triplet distance, but only the quartet-compiles can calculate the quartet distance. Also note that the triplet distance calculation for quartet compiles will be slower, as they will maintain all counters used for the quartet distance calculation.
- Input trees should be given in the Newick format[1] (and the file has to end on the `;` character, i.e. line breaks will not be tolerated).
- Triplets:
  ```
  <program file> fancy calcTripDist <t1> <t2>
  ```
- Quartets:
  ```
  <program file> fancy calcQuartDist <t1> <t2>
  ```

[1] http://en.wikipedia.org/wiki/Newick_format