Comparison and Construction of Phylogenetic Trees and Networks

Konstantinos Mampentzidis

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Comparison and Construction of Phylogenetic Trees and Networks

A Dissertation
Presented to the Faculty of Science and Technology
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Konstantinos Mampentzidis
November 7, 2019

Abstract

In this thesis we consider three algorithmic problems that appear in computational biology and are mainly about comparing and constructing rooted phylogenetic trees and networks. A rooted phylogenetic tree is a rooted unordered tree that is used to represent evolutionary relationships among species. A rooted phylogenetic network is a rooted directed acyclic graph, and is able to represent more complex evolutionary relationships that arise because of reticulation events such as horizontal gene transfer and hybridization.

The first problem is about computing the rooted triplet distance between two rooted phylogenetic trees that are built on the same leaf label set. We provide two new algorithms that are cache oblivious, one optimized for binary trees and another that works for trees of arbitrary degree. Our algorithms achieve the best time and space bounds in the RAM model, and at the same time are the first to scale to external memory. We implement our algorithms and compare them experimentally against previous state-of-the-art algorithms, and show that our algorithms achieve the best performance in practice.

The second problem is about computing the rooted triplet distance between two rooted phylogenetic networks that are built on the same leaf label set. Previous algorithms were either based on a trivial approach or only worked under certain restrictions on the degrees of the nodes of the two input networks. We provide algorithms that have no such restrictions, as well as implementations and extensive experiments on both simulated and real datasets illustrating their practical performance.

Finally, the third problem is about constructing rooted phylogenetic trees, where the input is a set of binary trees on three leaves over a leaf label set, and the objective is to construct a rooted phylogenetic tree that has as embedded subtrees the largest number of input trees. This turns out to be an NP-Hard problem. Previous approximation algorithms produced trees with an arbitrary number of internal nodes, some even always produced binary trees. We study the computational complexity of the version of this problem, where there is a constraint on the number of internal nodes that the output tree is allowed to have. We provide several inapproximability results, approximation algorithms, as well as implementations and extensive experiments on both simulated and real datasets.

Resumé

I denne afhandling studeres tre algoritmiske problemstillinger fra Computational Biology, hvor man ønsker at sammenligne og konstruere rodede fylogenetiske træer og netværk. Et rodfæstet fylogenetisk træ er et rodfæstet ikke-ordnet træ som anvendes til at repræsentere evolutionære relationer mellem arter. Et rodfæstet fylogenetisk netværk er en rodfæstet acyklisk graf, som kan repræsentere mere komplekse evolutionære relationer som opstår på grund af reticulation hændelser, som f.eks. horisontal genoverførsel og hybridisering.

Det første problem er at beregne tripletafstanden mellem to rodede fylogenetiske træer, der er bygget over den samme mængde af bladmærker. I afhandlingen præsenteres to nye algoritmer som er "cache oblivious", hvor den ene algoritme er optimeret til binære træer og til træer af vilkårlig grad. Algoritmerne opnår de bedst kendte udførselstider og pladsforbrug i RAM modellen, og er de første algoritmer der skalerer til ekstern hukommelse. Algoritmerne er blevet implementeret og sammenlignet eksperimentelt med de hidtil bedste algoritmeimplementeringer. De præsenterede algoritmer opnår i praksis den bedste performance.

Det andet problem er at beregne tripletafstanden mellem to rodede fylogenetiske netværk, der er bygget over den samme mængde af bladmærker. De hidtidige algoritmer var enten baseret på en triviel tilgang eller kunne kun håndtere netværk med begrænsninger på graderne af knuderne i de to netværk. I afhandlingen præsenteres algoritmer der undgår sådanne input-begrænsninger, og algoritmerne er blevet implementeret og omfangsrigt evalueret på både simulerede og virkelige datasæt for at illustrere deres praktiske performance.

Endelig, så er det tredje problem at konstruere rodede fylogenetiske træer, hvor inputtet er en mængde af binære træer med tre blade, hvor bladene er markeret med mærker fra en mængde af bladmærker, og hvor målet er at konstruere et fylogenetisk træ, som har det største antal inputtæer som indlejrede deltræer. Dette problem er NP-hårdt. Hidtidige approksimationsalgoritmer producerede træer med et vilkårligt antal indre knuder, nogle algoritmer sågar altid binære træer. I afhandlingen studeres beregningskompleksiteten for varianten af problemet, hvor der er en begrænsning på hvor mange interne knuder outputtræet må have. Der gives en række ikke-approksimere resultater, approksimationsalgoritmer præsenteres, og algoritmerne er blevet implementeret og omfangsrigt evalueret på både simulerede og virkelige datasæt.

Preface

This thesis is based on the following three papers, one for each problem stated in the abstract:

[11] Gerth Stølting Brodal and Konstantinos Mampentzidis. Cache Oblivious Algorithms for Computing the Triplet Distance Between Trees. In *25th Annual European Symposium on Algorithms (ESA 2017)*, volume 87 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. 2017.

[52] Jesper Jansson, Konstantinos Mampentzidis, Ramesh Rajaby, and Wing-Kin Sung. Computing the Rooted Triplet Distance Between Phylogenetic Networks. In *Combinatorial Algorithms*, pages 290–303. Springer International Publishing, 2019.

[51] Jesper Jansson, Konstantinos Mampentzidis, and Sandhya Thekkumpadan Puthiyaveedu. Building a Small and Informative Phylogenetic Supertree. In *19th International Workshop on Algorithms in Bioinformatics (WABI 2019)*, volume 143 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. 2019.

In Chapter 1 we start by giving a concise statement about the algorithmic problems considered in this thesis. Then, we provide an introduction to algorithms, models of computation, and algorithm engineering. Finally, for every paper listed above, we provide a survey of previous results as well as an overview of our contributions. Chapters 2, 3, and 4 contain the full versions of our papers and are currently submitted to journals for review. Finally, in the appendix we include additional algorithms and experiments related to Chapters 2, 3, and 4.

Acknowledgments

There are a lot of people who supported me during my PhD, and for that I am very thankful. It would probably take several pages to list all of them, and tomorrow, when the submission deadline has passed, I will most likely realize that I forgot someone and then feel bad about it. This is why I would like to mention the ones who I feel have had a significant impact.

I would like to thank Gerth Stølting Brodal for being the best advisor anyone could ever hope for. Without Gerth, none of this would be possible. Moreover, I would like to thank Lars Arge for creating MADALGO and giving me the opportunity to join the group as a PhD student. Such a great atmosphere in the group, so many activities, talks, and seminars to get inspired from. I feel very lucky to have been a member of such an amazing group. I would also like to thank Trine Ji Holmgaard Jensen for the many years of support she has provided to me as the center manager of MADALGO.

I would like to thank my professors from Greece Anastasios Gounaris, Kostas Tsichlas, and Yannis Manolopoulos for their support when applying for my PhD program. I am especially very thankful to Kostas Tsichlas for forwarding me an email he received from Gerth when Gerth was looking for new PhD students, which would then lead to me applying.

I would like to thank my co-authors Gerth Stølting Brodal, Jesper Jansson, Wing-Kin Sung, Sandhya Thekkumpadan Puthiyaveedu, and Ramesh Rajaby for all the fun research we have conducted together. In particular, I would like to thank Jesper Jansson for accepting me to come visit him in Hong Kong. My 5 month stay in Hong Kong has been the best experience of my life. Our meetings and discussions in Hong Kong with Jesper and our collaboration in the remaining year of my PhD education have been invaluable.

I would like to thank Martin Storgaard Dieu for the several course related projects we worked on together during my first two years as a PhD student. Our collaboration was very fruitful and most importantly fun.

I would like to thank Aarhus University for providing such a great learning environment at the Computer Science Department. I am especially thankful for giving me the opportunities to be a teaching assistant in the undergraduate course on algorithms and data structures.

I would like to thank Christian Nørgaard Storm Pedersen and Aslan Askarov for being in my PhD support group and for all the advice they have

given to me since the beginning of my PhD education.

I would like to thank all my friends for their support, especially Kallidoros Chatzopoulos and Giorgis Nomikos for all the fun times together and for regularly reminding me that life is not only about work.

I would like to thank my dear grandmother Nina Sachno and aunt Louiza Mampentzidou for being very close to my family and their moral support all these years.

Finally and most importantly, I would like to thank my parents Liountmila Mampentzidou and Ioannis Mampentzidis, as well as my brothers Dimitrios Mampentzidis and Alexandros Mampentzidis, for being such a great family, for always supporting me, and always believing in me.

Konstantinos Mampentzidis,
Aarhus, July 31, 2019.

Post-submission updates – November 7, 2019

1. Updated reference [51] after the proceedings were published.
2. Clarified the definition of the consistency of a triplet with a network or a block to cover a special case. Places affected: second paragraph of pages 15 and 63, first paragraph of page 79, and the items A and B in the concluding remarks of page 88.
3. Fixed a reference in the second paragraph of page 17.
4. In the statement of Lemma 2, it should be $|T|$ instead of n.
5. In the proof of Lemma 2: removed the unnecessary dependency that existed before for T to be left-heavy.
6. In page 37 and the sentence before Theorem 1: emphasized the connection between $|T|$ and n.
7. In the list of page 54: changed the steps to emphasize that in our implementation we first create the components $\texttt{comp}(T_1(u_l))$ and $\texttt{comp}(T_2(u_l))$ and recurse on them, then after we return from the recursive call we create the components $\texttt{comp}(T_1(u_r))$ and $\texttt{comp}(T_2(u_r))$ and recurse on them, and finally, after we return from the recursive call we create the components $\texttt{comp}(T_1(u_p))$ and $\texttt{comp}(T_2(u_p))$ and recurse on them.
8. In the second list of page 55: clarified the choice for $|\Lambda_l|$.
9. Corrected the statements of Lemmas 14 and 15, to require the lowest common ancestor of every pair of leaves (x, y), (x, z), and (y, z) to be w. Before we had that the lowest common ancestor of x, y, and z is w. Similarly, the captions of Algorithms 7 and 8 were updated.
10. The rest of the updates are typos and language corrections.

Contents

Abstract **i**

Resumé **iii**

Preface **v**

Acknowledgments **vii**

Contents **ix**

1 Introduction **1**
1.1 Algorithms and Models of Computation 2
1.2 Algorithm Engineering 4
1.3 Comparing Phylogenetic Trees 5
1.4 Comparing Phylogenetic Networks 13
1.5 Building Small and Informative Supertrees 22

2 Cache Oblivious Algorithms for Computing the Triplet Distance Between Trees **29**
2.1 Introduction 30
2.2 Previous Approaches 33
2.3 The New Algorithm for Binary Trees 34
2.4 The New Algorithm for General Trees 43
2.5 Implementation 53
2.6 Experiments 55
2.7 Conclusion 59

3 Computing the Rooted Triplet Distance between Phylogenetic Networks **61**
3.1 Introduction 62
3.2 A First Approach 66
3.3 A Second Approach 71
3.4 Implementation and Experiments 85
3.5 Concluding Remarks 88

4 Building a Small and Informative Phylogenetic Supertree **91**
4.1 Introduction 92
4.2 Computational Complexity of q-MAXRTC 95
4.3 Approximability of q-MAXRTC 96
4.4 Implementation and Experiments 103
4.5 Motivation for q-MAXRTC: Faster Rooted Triplet Distance 107
4.6 Concluding Remarks 113

A Additional Experiments for Chapter 2 **117**

B Additional Algorithms and Experiments for Chapter 3 **135**

C Additional Algorithms and Experiments for Chapter 4 **143**

Bibliography **149**

Chapter 1

Introduction

In this thesis we consider three algorithmic problems that appear in computational biology and are mainly about comparing and constructing rooted phylogenetic trees and networks.

In biology, the evolutionary relationships that exist between species are commonly represented by a *rooted phylogenetic tree*. The leaves in such a tree can correspond to species that exist today and internal nodes to ancestor species that used to exist in the past. When studying the evolution of a fixed set of species, because of the different available data and algorithms that are used to construct the phylogenetic trees, it is possible to obtain two trees with the same leaf label set that look structurally different. Quantifying this structural difference is essential to improve the quality of evolutionary inferences. Hence, it becomes natural to ask the following: Given two rooted phylogenetic trees that are built on the same leaf label set, how different are they? A popular distance measure that is used in the literature, and we also consider in this thesis, is the *rooted triplet distance* [25].

Trees are simple data structures that only model the transfer of genes from ancestors to their direct descendants, i.e., from parents to their children. However, evolution consists of more complicated events, called reticulation events [43], that can make it possible for genes to be transferred between unrelated species (e.g., hybrid species). A tree is by definition an undirected graph with no cycles, thus it cannot be used to represent reticulation events. Instead, a *rooted phylogenetic network* that has a rooted directed acyclic graph as an underlying structure is used. Such a network can represent both the events that a tree represents, as well as the reticulation events. Note that a tree can be thought of as a network with no reticulation events. For the same reasons as with trees, it becomes useful to define distance measures for the comparison of two rooted phylogenetic networks that are built on the same leaf label set. In this thesis, we consider the extension by Gambette and Huber [33] of the rooted triplet distance measure from the case of rooted phylogenetic trees to the case of rooted phylogenetic networks.

Finally, we consider the problem of constructing rooted phylogenetic trees from a given set of observations. This is commonly referred to in the literature as the *supertree problem* [9]. There are many different variations of this problem. We propose and study the computational complexity of a natural combination of the two problems considered in [48] and [13].

1.1 Algorithms and Models of Computation

Algorithms are recipes for solving all kinds of problems. Different algorithms lead to different solutions, some more preferred over others. For example, let us say that you want to travel from Aarhus to Copenhagen by plane. One possible algorithm is to fly from Aarhus to Athens, and then fly from Athens to Copenhagen. Another algorithm is to fly from Aarhus to Copenhagen directly. Which of the two algorithms is better and why? If time is the main concern and assuming good enough flight conditions, then obviously the latter algorithm is better. If cost is the main concern, then the answer to this question depends on the plane tickets' prices.

In Computer Science and in this thesis, we consider problems that are *computational*, i.e., problems that can be solved by a computer and the algorithms of which would typically require a lot more time to execute by hand instead of by a computer (e.g., sorting 1 million numbers in ascending order). Such a problem consists of a description of the *input* (e.g., n numbers), as well as a description of the *output* (e.g., the n numbers sorted in ascending order). Then, given a valid input, an *algorithm* consists of a sequence of instructions/operations that are intended for the computer to perform in order to produce the desired output.

The process of creating a new algorithm to solve a problem consists of three phases. In the first phase, we *design* the algorithm, where the sequence of operations to be performed by the computer is described. An important aspect of algorithm design is to also decide how the data should be organized in memory, i.e., which *data structure* to use, in order to be able to support the operations as efficiently as possible. In the second phase, we prove the *correctness* of the algorithm, where we show that for any valid input, the algorithm will produce an output, and that the output produced will always be the desired output. In the third and final phase, we *analyze* the algorithm where we prove some guarantees about the algorithms' resource requirements, for example time (how long will we have to wait for the output?) and space (how much memory will the algorithm need to use?) requirements.

Algorithms are designed and analyzed following a model of computation, where the information about the operations that an algorithm is allowed to perform, and the amount of memory that an algorithm is allowed to use is precisely defined. In this thesis, we mainly consider the word RAM model and the cache oblivious model.

1.1.1 Word RAM Model

The word RAM (*word random access machine*) model [31] is an extension of the RAM (*random access machine*) model [75], and resembles closely how modern computers work. The RAM model dates back to the work of Von Neumann in the 1940s. In this model, the computer has access to an unlimited amount of *main memory*, and every typical operation that can be performed by a processor on a register (e.g., $+$, $*$, $-$, $=$, $/$, load, store) is said to require 1 unit of time. One drawback of this model is the assumption that the size of a register is assumed to be unlimited, meaning that for e.g., adding two numbers of arbitrary size would only require 1 unit of time. However, in modern computers a typical register can only hold 32 or 64 bits. To capture this, in the word RAM model the main memory is divided into cells, called *words*. Each word can hold w bits, and every typical operation, including bit-shift operations, is applied on words and assumed to require 1 unit of time. What makes this model more realistic is the fact that the size of a word is limited. More precisely, if n is the number of words in a given input, then w is large enough so that we are able to at least index any word in the input, i.e., $w \geq c \log n$, where c is a large enough non-negative constant.

When analyzing algorithms in this model, we are interested in the *time* and *space* analysis. For the time analysis, one approach is to count exactly how many operations an algorithm performs as a function $t(n)$, where n is the input size. This however would be both tedious and mainly useful for small n, as when n becomes large enough, constants and lower-order terms of $t(n)$ have an insignificant impact to the *growth* of $t(n)$. For this reason, we consider the *asymptotic* behavior [21, Chapter 3] of $t(n)$ as n increases, instead of its *exact* behavior. For the space analysis we follow a similar approach, except now we define a function $s(n)$ that is the maximum number of words allocated at any given point of time during the execution of an algorithm. Note that a trivial relationship between $t(n)$ and $s(n)$ is $t(n) \geq s(n)$.

It is common in the literature with the RAM model to imply the word RAM model. From here on, we also do the same. Moreover, unless otherwise stated, every algorithm is designed and analyzed in this model. Finally, in the analysis we do not explicitly refer to words, and we assume that by default operations are performed on words of size w. For example, if we say that we store an array of n integers in memory, we imply that we have an array of n words containing the integers in memory, i.e., any number used by an algorithm in this thesis can fit in a word.

1.1.2 Cache Oblivious Model

In the RAM model, we assume that there is one type of memory that is infinite, and accessing any element in memory has the same unit cost. However, in a typical modern computer the memory is finite and consists of a multiple

hierarchy of memories (e.g., L1/L2/L3 cache, RAM (*random access memory*), disk), where the cost of accessing one level differs to the cost of accessing another. This difference can be quite big, for example accessing a typical RAM is in the order of nanoseconds, while accessing a hard disk drive is in the order of milliseconds, i.e., 1 million times slower [4]. This means that two algorithms in the RAM model might have the same $O(n)$ time complexity, but in practice their running times might be completely different if one of the algorithms relies heavily on random memory access. This was a main motivating factor behind introducing new models of computation, like the I/O model [1] and the cache oblivious model [32].

The I/O model, introduced by Aggarwal and Vitter [1], is motivated by problems that rely a lot on processing massive amounts of data, i.e., data that cannot fit in RAM. The main bottleneck for these problems lies in the time it takes to transfer data between the disk and RAM. In this model, the memory hierarchy is assumed to consist of a *main memory* (RAM) of fixed size M and an *external memory* (disk) of unlimited size. An algorithm can read/write B consecutive elements from/to the disk, the cost of which corresponds to one I/O operation (or simply I/O). The goal of an algorithm in this model is to minimize the number of I/Os that are incurred during its execution.

The cache oblivious model, introduced by Frigo *et al.* [32], is the same as the I/O model, except the parameters M and B are unknown to the algorithm. An algorithm in this model is described just like it would be described in the RAM model. The analysis however is performed in the I/O model with the assumption that there is an optimal cache replacement strategy for transfering the blocks of size B between the main memory and the external memory. What makes this model exciting is the fact that since the analysis holds for any block and memory size, it also holds for all levels in a multi-level memory hierarchy, thus making algorithms in this model very useful for architectures that have a large memory hierarchy, e.g., modern computers.

1.2 Algorithm Engineering

The I/O and cache oblivious models have been quite successful at addressing central issues, e.g., the cost of accessing different levels of memory being significantly different, that the RAM model could not sufficiently address. However, as computer architecture continues to become more and more complicated and this complexity is not precisely captured by the theoretical models, it can often be difficult for the theoretical algorithmic results to carry over to practice. One seemingly obvious attempt at a solution to this problem could be to create more complex theoretical models that capture this increase in architecture complexity. However, this would result in the design of algorithms becoming more tedious and complicated. Instead, we use *algorithm engineering* [69].

The term algorithm engineering was first used by Giuseppe F. Italiano

in 1997 who organized the "Workshop on Algorithm Engineering" in Venice, Italy. An early survey appeared two years later by Cattaneo *et al.* [35], and an attempt at a definition of algorithm engineering as a general methodology for algorithmic research was given by Peter Sanders [69] in 2009. The main goal in algorithm engineering is to bridge the *gap* that exists between algorithm theory and practice. We achieve this by focusing not just on the theory behind the algorithms but also the implementation and experimentation. Both theory and practice work together in a feedback loop fashion: algorithms are designed, then implemented, experiments are performed, and the practical results are used as guideline to change the original design, which, if and when changed, the implementations also change, experiments are rerun, and the same cycle is repeated until we get a satisfying theoretical and practical performance.

In this thesis, the main objective is to present novel algorithms for central problems in the area of algorithms and data structures that are efficient *both* in theory and in practice. Algorithms improving significantly upon previous approaches, their implementations, and experiments illustrating their practical performance are provided.

1.3 Comparing Phylogenetic Trees

The first problem we consider in this thesis is the computation of the *triplet distance* between trees, which is a distance measure for comparing rooted trees. Trees are data structures that are used in many different fields to represent relationships. One example comes from evolutionary biology, where a type of a tree called *phylogenetic tree* is used to represent evolutionary relationships between species. The leaves in such a tree can correspond to species that are currently alive today, and internal nodes to ancestor species that used to exist in the past. When studying the evolutionary history of a fixed set of n species, different available data and construction methods can result in phylogenetic trees that look structurally different. To make better inferences about the evolutionary relationships of the species, it becomes crucial to know whether the structural difference that exists between the trees is statistically significant or not. For this reason, several different tree distance measures have been proposed in the literature. A class of them includes distance measures that count how many times certain features do not appear in both input trees, e.g., the Robinson-Foulds distance [64], the triplet distance [25] for rooted trees, and the quartet distance [26] for unrooted trees. In this section, we formally define the triplet distance problem, we present a detailed survey of previous results, an overview of our contributions, and finally, some open problems.

1.3.1 Preliminaries and Problem Definition

In the problem of computing the triplet distance between two trees, we consider rooted unordered trees that are distinctly leaf labeled. For such a tree T,

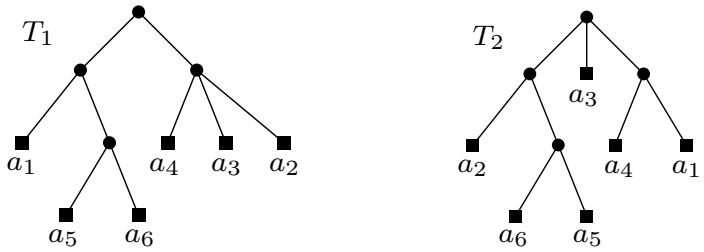


Figure 1.1: Two trees with leaf label set $\{a_1, a_2, a_3, a_4, a_5, a_6\}$. Examples of shared triplets are $a_5a_6|a_3$, $a_2|a_3|a_4$. Examples of triplets that are only induced by one tree are $a_6a_2|a_3$, $a_1|a_3|a_2$. Here, $D(T_1, T_2) = 15$.

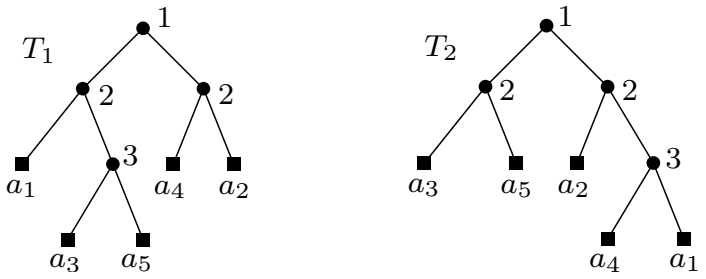
a *triplet* is defined by three leaf labels x, y, and z, as well as their induced tree topology in T. If the lowest common ancestor between any pair of leaves happens to be the same internal node in T, we say that x, y, and z form a *fan triplet*, and we represent that triplet with $x|y|z$. Some examples of fan triplets are $a_1|a_3|a_2$ and $a_4|a_3|a_5$ in T_2 of Figure 1.1. The remaining case of a triplet is called a *resolved triplet*, and is represented with $xy|z$ when the lowest common ancestor of x and y is further from the root than the lowest common ancestor of x (or y) and z. Some examples of resolved triplets are $a_2a_6|a_1$ and $a_5a_6|a_2$ in T_2 of Figure 1.1.

Given two trees T_1 and T_2 that are built on the same leaf label set of size n, the triplet distance $D(T_1, T_2)$ is defined as the total number of leaf triples whose tree topology differs in the two trees. Most algorithms in the literature compute the total number of shared triplets $S(T_1, T_2)$, i.e., total number triplets that appear in both trees. Then, we have $D(T_1, T_2) = \binom{n}{3} - S(T_1, T_2)$, where $\binom{n}{3}$ is the total number of subsets of size 3 from the leaf label set. Figure 1.1 contains examples of shared and non-shared triplets between two trees.

1.3.2 Previous Results

A naive algorithm to compute $S(T_1, T_2)$ would go through all the $4\binom{n}{3}$ triplets and for each triplet check whether it is shared or not. To check efficiently if a triplet appears in a tree, we can use the data structure in [7]. For a tree T with n leaves this data structure can, after spending $O(n)$ preprocessing time and space, answer lowest common ancestor queries between any pair of leaves in $O(1)$ time. The time taken by the naive algorithm would then be $O(n^3)$, and the space $O(n)$.

The first improvement over the naive approach was given in 1996 by Critchlow *et al.* [23], but only for the case where both T_1 and T_2 are binary trees. Their algorithm uses $O(n^2)$ time and space, and works by exploiting the information about the depth of the leaves' ancestors. More precisely, for T_1 a *generational table* A of size $n \times n$ is defined such that $A[i, j]$ stores the depth of



	a_1	a_2	a_3	a_4	a_5
a_1	-	1	(2)	1	(2)
a_2	1	-	(1)	2	(1)
a_3	2	(1)	-	(1)	3
a_4	1	2	(1)	-	(1)
a_5	2	(1)	3	(1)	-

A

	a_1	a_2	a_3	a_4	a_5
a_1	-	2	(1)	3	(1)
a_2	2	-	(1)	2	(1)
a_3	1	(1)	-	(1)	2
a_4	3	2	(1)	-	(1)
a_5	1	(1)	2	(1)	-

B

Figure 1.2: Two binary trees with leaf label set $\{a_1, a_2, a_3, a_4, a_5\}$. The corresponding generational tables [23] are depicted below each tree. The leaf a_1 associates with the generational pattern $(2, 1)$ twice due to the shared triplet $a_3a_5|a_1$. Similarly, a_2 associates with $(1, 1)$ because of $a_3a_5|a_2$, a_3 because of $a_2a_4|a_3$, a_4 because of $a_3a_5|a_4$, and a_5 because of $a_2a_4|a_5$. Hence, $S(T_1, T_2) = 5\binom{2}{2} = 5$ and $D(T_1, T_2) = \binom{5}{3} - S(T_1, T_2) = 10 - 5 = 5$.

the lowest common ancestor of the leaves with labels i and j. Similarly, a generational table B is defined for T_2. Then, we have that the resolved triplet $jk|i$ is shared between T_1 and T_2 if and only if $A[i, j] = A[i, k]$ and $B[i, j] = B[i, k]$. To compute $S(T_1, T_2)$ efficiently, a *generational pattern* that is associated with the leaf labels i and j is defined as the pair $(A[i, j], B[i, j])$. Note that if i is associated with the two generational patterns $p_1 = (A[i, j], B[i, j])$, $p_2 = (A[i, k], B[i, k])$, and $p_1 = p_2$, then $jk|i$ is a shared triplet between T_1 and T_2. Generally, if i associates with X generational patterns, then there are exactly $\binom{X}{2}$ shared triplets represented as $xy|i$, where x and y are leaf labels. Thus, if $f(m, i)$ is defined as the total number of times that a leaf with label i associates with the m-th generational pattern, we have $S(T_1, T_2) = \sum_i \sum_m \binom{f(m,i)}{2}$. An example can be found in Figure 1.2.

More than a decade later and in 2011, Bansal *et al.* [5], provided an algorithm that used $O(n^2)$ time and space, but now worked for both binary and non-binary trees. The information that this algorithm exploits is the common leaves that can exist between a subtree of T_1 and a subtree of T_2. To compute $D(T_1, T_2)$ in the given time and space bounds, the algorithm mainly relies on a subroutine that uses dynamic programming to find the total number of common leaves between any pair of subtrees, one from T_1 and one from T_2.

In 2013, Sand *et al.* [66] provided the first subquadratic algorithm for

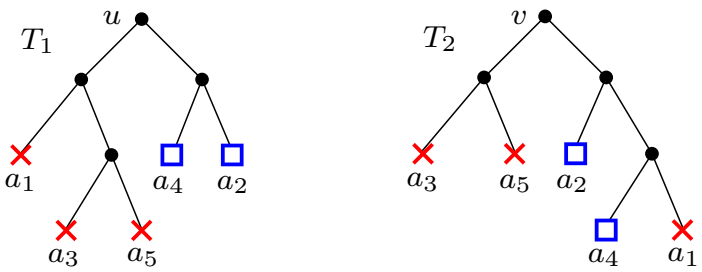


Figure 1.3: Counting the shared triplets anchored in a node u of T_1, according to [66]. The triplets anchored in u are $a_1a_3|a_4$, $a_1a_3|a_2$, $a_1a_5|a_4$, $a_1a_5|a_2$, $a_3a_5|a_4$, $a_3a_5|a_2$, $a_4a_2|a_1$, $a_2a_4|a_3$, and $a_2a_4|a_5$. The shared triplets $a_3a_5|a_2$, $a_3a_5|a_4$, $a_2a_4|a_3$, and $a_2a_4|a_5$ are captured in the node v of T_2. We have $v_{lr} = 2$, $v_{lb} = 0$, $v_{rr} = 1$, $v_{rb} = 2$, and $S_{uv}(T_1, T_2) = \binom{2}{2}2+\binom{0}{2}1+\binom{1}{2}0+\binom{2}{2}2 = 4$.

computing $D(T_1, T_2)$. The algorithm however works only for the case where both T_1 and T_2 are binary trees, and uses $\mathrm{O}(n \log^2 n)$ time and $\mathrm{O}(n)$ space. This improvement is achieved by extending a new algorithm that is also provided in [66] and uses $\mathrm{O}(n^2)$ time and $\mathrm{O}(n)$ space. This $\mathrm{O}(n^2)$-time algorithm is fairly simple and defines a coloring of the leaves in order to capture and count triplets. More precisely, every triplet $ij|k$ in T_1 and T_2 is anchored in the lowest common ancestor of i, j, and k. To capture the triplets that are anchored in a node u of T_1, the leaves in the left subtree of u are colored red and the leaves in the right subtree of u are colored blue. Then, every triplet $xy|z$ such that both x and y are red and z is blue, or both x and y are blue and z is red, is anchored in u. Let $S_{uv}(T_1, T_2)$ be the number shared triplets anchored in a node u of T_1 and a node v of T_2. To compute $S_{uv}(T_1, T_2)$, the coloring of the leaves in T_1 is transferred to the leaves in T_2. Let v_{lr} and v_{lb} be the number of red and blue leaves in the left subtree of v. Similarly, let v_{rr} and v_{rb} be the number of red and blue leaves in the right subtree of v. We then have:

$$S_{uv}(T_1, T_2) = \binom{v_{lr}}{2} v_{rb} + \binom{v_{lb}}{2} v_{rr} + \binom{v_{rr}}{2} v_{lb} + \binom{v_{rb}}{2} v_{lr} \,.$$

An example can be found in Figure 1.3. Note that given a coloring defined by a node u in T_1, the values $S_{uv}(T_1, T_2)$ for any node v in T_2 can be computed in $\mathrm{O}(n)$ time. Then, given that $S(T_1, T_2) = \sum_{u \in T_1} \sum_{v \in T_2} S_{uv}(T_1, T_2)$, i.e., for every node u in T_1 we can spend $O(n)$ time in T_2 to compute $S(T_1, T_2)$, the running time becomes $\mathrm{O}(n^2)$, with the space being only $\mathrm{O}(n)$ as we only store the two trees in memory plus $\mathrm{O}(1)$ additional counters in every node of T_2.

To reduce the time, Sand *et al.* [66] proposed an order of coloring the leaves of T_1, so that every leaf changes color $\mathrm{O}(\log n)$ times. This is achieved by using the *smaller half trick* as follows. During a depth first traversal of T_1, the following two invariants are maintained:

1. When exploring a node u, the leaves in the subtree of u are red, and every other leaf in T_1 has no color.
2. When returning from u, the leaves in the subtree of u have no color.

The algorithm starts by coloring every leaf of T_1 red. Then, it applies a depth first traversal starting from the root of T_1. For every node u that is explored, it chooses the smallest child subtree, denoted $S(u)$, to recursively color blue before computing $\sum_{v \in T_2} S_{uv}(T_1, T_2)$. Then, every leaf with color blue gets uncolored before recursing to the largest child subtree of u, denoted $L(u)$. When the processing of $L(u)$ is finished, by the second invariant the leaves in $L(u)$ have no color, so the algorithm colors $S(u)$ red and recurses to $S(u)$. For the analysis one observes that a leaf changes color when there is an ancestor node u' that belongs to the smallest child subtree of u, where u is a parent of u'. A leaf can only have $O(\log n)$ such ancestors, thus every leaf changes color $O(\log n)$ times.

So far, the algorithm has managed to reduce the time spent in T_1. More precisely, we went from $O(n^2)$ time, because a naive order of coloring the leaves of T_1 will in the worst case change a color of a leaf $O(n)$ times, down to $O(n \log n)$ by using the smallest half trick. However, if we act as before and spend $O(n)$ time in T_2 for every node u in T_1, the running time will still be $O(n^2)$. To reduce the time spent in T_2, Sand *et al.* [66] provided a tree-like data structure, called *hierarchical decomposition* (HDT), for storing T_2. From a high level perspective, the HDT of T_2 can be constructed in $O(n)$ time, while using $O(n)$ space, and it can maintain the value $\sum_{v \in T_2} S_{uv}(T_1, T_2)$ by only spending $O(\log n)$ time for every leaf color change in T_1. Since there are $O(n \log n)$ leaf color changes, the total running time becomes $O(n \log^2 n)$. An implementation of the algorithm is also provided in [66] and shown to be more efficient in practice compared to the $O(n^2)$-time algorithm by Critchlow *et al.* [23].

In the same year, Brodal *et al.* [12] managed to reduce the time of the algorithm in [66] from $O(n \log^2 n)$ to $O(n \log n)$. Moreover, in the same paper the first subquadratic algorithm was also presented for general non-binary trees, with the running time being $O(n \log n)$ but now with the space increased to $O(n \log n)$. This improvement in the running time was achieved by using a careful contraction scheme for the HDT of T_2. More specifically, every time a node u is explored during the depth first traversal of T_1, both T_2 and the HDT of T_2 are contracted so that their sizes are proportional to the size the subtree of u in T_1, while at the same time maintaining all necessary triplet information to properly detect future shared triplets (the exact details are presented in the paper). The algorithms in [12] were also implemented and added to the library tqDist [68].

Interestingly, in 2014 it was shown by Holt *et al.* [40] that for binary trees, in practice the $O(n \log^2 n)$ algorithm in [66] is faster than the $O(n \log n)$ algorithm in [12]. A year later and in 2015, Jansson and Rajaby [45] gave an even slower theoretically algorithm, using $O(n \log^3 n)$ time and $O(n \log n)$

Table 1.1: Previous and new results for computing $D(T_1, T_2)$, where T_1 and T_2 are trees that are built on the same leaf label set of size n. In the previous papers, no analysis in the cache oblivious model was provided, so here we give an upper bound for the number of I/Os incurred.

Year	Reference	Time	IOs	Space	Non-Binary Trees
1996	Critchlow *et al.* [23]	$O(n^2)$	$O(n^2)$	$O(n^2)$	No
2011	Bansal *et al.* [5]	$O(n^2)$	$O(n^2)$	$O(n^2)$	Yes
2013	Sand *et al.* [66]	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(n)$	No
2013	Brodal *et al.* [12]	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Yes
2015	Jansson & Rajaby [45]	$O(n \log^3 n)$	$O(n \log^3 n)$	$O(n \log n)$	Yes
2017	New [Chapter 2]	$O(n \log n)$	$O(\frac{n}{B} \log_2 \frac{n}{M})$	$O(n)$	Yes

space, for computing $D(T_1, T_2)$ for both binary and non-binary trees. However, experiments were also provided showing that their $O(n \log^3 n)$-time algorithm was actually the most efficient in practice. This was achieved by using a less complicated HDT for T_2, based on a heavy-light decomposition of T_2.

1.3.3 Our Contributions

In Table 1.1 we provide a list of previous and new results. From previous results, the theoretically fastest algorithm by Brodal *et al.* [12] is not the fastest algorithm in practice. On the other hand, the $O(n \log^3 n)$-time algorithm by Jansson and Rajaby [45], even though slower in theory by a $\log^2 n$ factor, in practice it was documented on a reasonable tree generation model to achieve the best performance. Thus, there is a theoretical and practical gap for the problem of computing $D(T_1, T_2)$.

We bridge this gap by providing two new algorithms, one optimized for the case where T_1 and T_2 are restricted to be binary trees and the other for the unrestricted case. Both algorithms have an $O(n \log n)$ running time, they use $O(n)$ space, and at the same time have the best practical performance. We achieve this by also making our algorithms cache oblivious. More precisely, in the cache oblivious model, the total number of I/Os incurred by both algorithms is $O(\frac{n}{B} \log_2 \frac{n}{M})$. We emphasize that our algorithms are also the first in the literature that can scale to external memory. To see why this is the case, consider any previous $O(n \operatorname{polylog} n)$-time algorithm. They all work by building an HDT for T_2 and then performing an operation to this HDT, which corresponds to a leaf to root path traversal of the HDT, every time a leaf changes color. However, a leaf changes color $O(n \log n)$ times because of the smallest half trick, thus for large enough trees the algorithms would need to access the disk at least once in order to update the HDT of T_2. Hence, the total number of I/Os incurred is $\Omega(n \log n)$.

Finally, we provide an implementation of our algorithms that is publicly available at `https://github.com/kmampent/CacheTD`, as well as extensive experiments that illustrate their practical performance. In the remaining part of this subsection, we give a high level overview of our algorithms. Details can be found in Chapter 2 and Appendix A.

Binary Trees. Our algorithm can be considered as an extended version of the $O(n^2)$-time algorithm in [66] that we also described above, with two main differences. First, we define a new order of visiting the nodes of T_1 that is not based on the smallest half trick, but on a hierarchical decomposition of T_1. As we show in Section 2.3.2, this decomposition can be computed recursively by scanning heavy paths of T_1. Second, we do not store T_2 in a data structure, and instead we process it by scanning it for two different reasons, one to contract T_2 into a smaller tree whose triplet topologies are all induced by T_2 and the other to count shared triplets that are anchored at the node that is currently visited in T_1. Notice that the word *scanning* is used to describe the main operations on both input trees. We manage to completely remove the need of data structures and scanning is the basic primitive in the algorithm. Scanning is very efficient in the external memory models, i.e., scanning s consecutive elements in memory in the cache oblivious model takes $O(s/B)$ I/Os, which is optimal.

The order of visiting the nodes of T_1 is specified by a depth first traversal of a decomposition of T_1, called *modified centroid decomposition* and denoted $MCD(T_1)$. $MCD(T_1)$ is a tree, the nodes of which, also called *centroids*, are the same as the nodes of T_1 and each node corresponds to a connected part of T_1, also called *component of* T_1 (see Figure 1.4(a)). We show that every such component has at most one edge crossing its boundary from below, and in Lemma 2 we prove that the height of the decomposition is $O(\log n)$. It is important to have only one edge going out from below, because the goal is for every node u in $MCD(T_1)$, to compute the number of shared triplets anchored in u. More than one edges from below, would make counting shared triplets complicated and potentially the running time of the algorithm much worse.

For every node u in $MCD(T_1)$, if we scan the entire T_2 to compute the shared triplets anchored in u, the running time of the algorithm will still be $O(n^2)$. We show that we can instead scan a smaller version of T_2. More precisely and as also illustrated in Figure 1.4(b), if L_u is the set of leaves belonging to the component of u in $MCD(T_1)$, we scan the contracted version $T_2(u)$, which is a tree on L_u with all triplet topologies induced by L_u in T_2 maintained. As shown in Section 2.3.3, for a given level of $\mathrm{MCD}(T_1)$ and every node in that level, the corresponding contracted versions of T_2 are disjoint. Moreover, for two nodes u, u_p in $MCD(T_1)$ where u_p is the parent of u in $MCD(T_1)$, $T_2(u)$ can be computed by scanning $T_2(u_p)$. Hence, we only spend $O(n)$ time for every level of $MCD(T_1)$, giving an $O(n \log n)$ time overall. For the space, we only need to maintain in memory for a root to leaf path $u_1, \ldots, u_k$ in $MCD(T_1)$ the corresponding contracted versions $T_2(u_1), \ldots, T_2(u_k)$ (see Figure 1.4(c)).

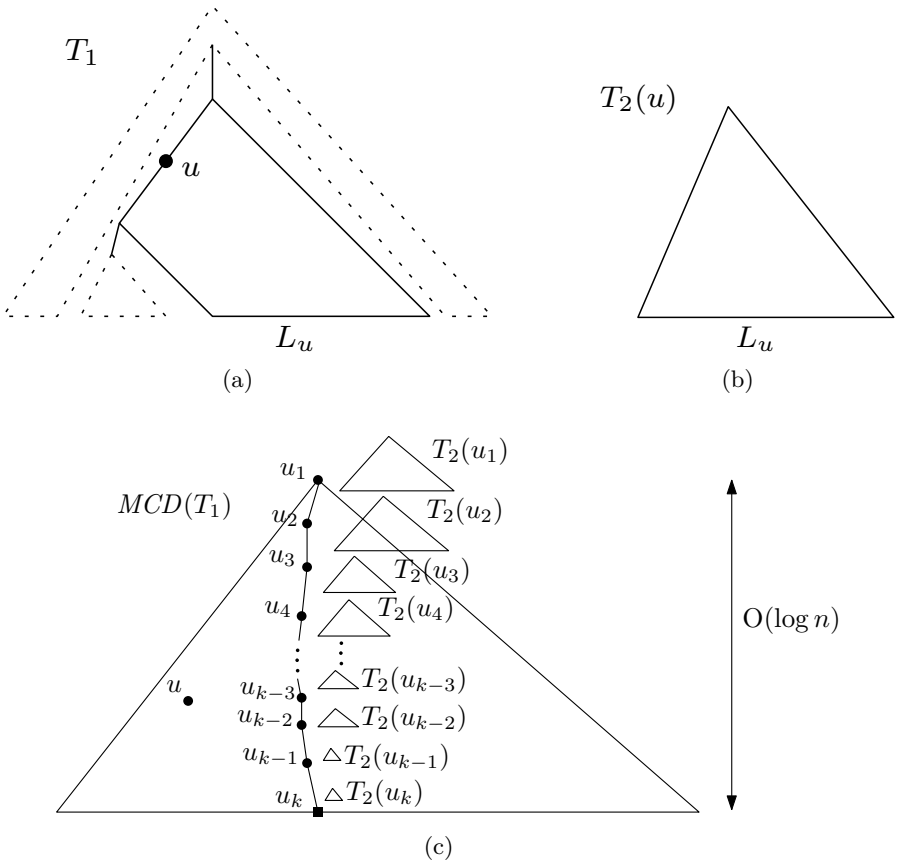


Figure 1.4: (a) The corresponding component (black polygon) in T_1 of a node u in $MCD(T_1)$. (b) The contracted version of T_2, which is a tree on the same leaf label set L_u defined by the component of u in T_1, with all triplet topologies induced by L_u in T_2 maintained. (c) A root to leaf path in $MCD(T_1)$ and the corresponding contracted versions of T_2. We have $\sum_{i=1}^{k} |T_2(u_i)| = \mathrm{O}(n)$.

If for a given tree T, we denote by $|T|$ the size of tree (internal nodes and leaves), we prove in Lemma 3 that $\sum_{i=1}^{k} |T_2(u_i)| = \mathrm{O}(n)$, thus making the space of our algorithm $O(n)$.

To make our algorithm scale to external memory, we have a preprocessing step where we make T_1 left-heavy, i.e., for every node u of T_1 the children are swapped if necessary in order to make the left child's subtree larger than the right child's subtree. Moreover, we store the left-heavy version of T_1 following the preorder layout in main memory, so that later when scanning a heavy path of size s during the construction of $MCD(T_1)$, we only spend $\mathrm{O}(s/B)$ I/Os. Afterwards, we relabel the leaves in T_1 according to their visiting time in a depth first traversal of T_1. This relabeling is also transferred to the leaves in T_2. Then, because any subtree of T_1 will contain a continuous range of

leaf labels, the coloring of any subtree in T_1 becomes very easy to transfer between T_1 and T_2. The tree T_2 and its contractions are stored in memory following the postorder layout. In Section 2.3.4 we provide a formal analysis of the algorithm to argue why it only performs $O(\frac{n}{B} \log_2 \frac{n}{M})$ I/Os in the cache oblivious model.

Non-binary Trees. When T_1 and T_2 have no restrictions on their degrees, fan triplets can appear in the trees, which must also be captured and properly counted. In Section 2.4.1, we extend the $O(n^2)$-time algorithm for the binary case, to the non-binary case. The time and space bounds remain the same, i.e., $O(n^2)$ and $O(n)$ respectively. This is achieved by anchoring the triplets in edges instead of nodes, and using four colors to capture every triplet instead of two. In Section 2.4.2 we show how to reduce the time to $O(n \log n)$. To achieve this, we binarize T_1 to obtain $b(T_1)$, by having for every node u with k children in T_1, a binary tree below u with the k children as leaves. Each node in $b(T_1)$ corresponds to one edge of T_1. We then use the same ideas as before, i.e., we build the $MCD(b(T_1))$ and apply a depth first traversal to find an order in which to visit the edges of T_1. Every node of $b(T_1)$ corresponds to a contracted version of T_2 that we use to count shared triplets that are anchored in the corresponding edge of T_1. To make the algorithm scale to external memory, we make sure that $b(T_1)$ is a left-heavy tree, we store it in memory following a preorder layout, and we also relabel the leaves similarly to the case of binary trees. Like in the binary case, T_2 and its contractions are stored in memory following the postorder layout. Finally, similarly to Section 2.3.4, in Section 2.4.3 we provide a formal analysis in the cache oblivious model.

1.3.4 Open Problems and Future Work

A major open problem is whether or not it is possible to create an algorithm for computing $D(T_1, T_2)$ with a better running time than $O(n \log n)$ in the RAM model. An approach that might work is to exploit the size of the word by storing additional information in it with word packing techniques. A possible extension for our algorithm is the following: If there exists an algorithm that can in $O(n)$ time count shared triplets anchored in $O(\log^{\varepsilon} n)$ centroids and create all contractions of T_2 for $O(\log^{\varepsilon} n)$ centroids, where ε is a small enough non-negative constant, then the running time of the resulting algorithm will be $O(n \frac{\log n}{\log \log n})$. Another open problem is to prove a non-trivial lower bound. The trivial lower bound for this problem is $O(n)$. Finally, in the I/O and cache oblivious models, is it possible to achieve the sorting bound, i.e., create an algorithm that requires $O(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{M})$ I/Os?

1.4 Comparing Phylogenetic Networks

In this section we consider the second problem studied in this thesis, which is about computing the triplet distance between more complex structures than

trees, called *phylogenetic networks* [43]. A phylogenetic tree over a set of species only models and captures the transfer of genes from ancestors to their direct descendants, i.e., from parents to their children. However, it has been observed that more complex evolutionary events, called *reticulation events* [43], can even make it possible for genes to transfer between unrelated species. One such event is called *horizontal gene transfer* [14], where genetic material is passed to an unrelated organism, e.g., via organ transplantation, viruses, and bacteria. Another is called *hybridization* [6], which happens when genes are transferred between two different breeds or species e.g., through interbreeding. Other examples include *genetic recombination* [71], when genetic information is transfered between two different chromosomes, and *gene duplication and loss* [22] through which new genetic material is created.

A tree cannot represent reticulation events, since a reticulation event implies species with potentially more than one direct ancestors, i.e., the existence of cycles in the undirected version of the tree. In the previous section, we considered rooted trees with implied directions on the edges going from ancestors to descendants. In this section, we will be more explicit with these directions. More precisely, a rooted phylogenetic network uses a *rooted directed acyclic graph (rooted DAG)* to represent both the transfer of genes from the parents to their children and the reticulation events. Like with phylogenetic trees, we can have unrooted phylogenetic networks as well, but since here we consider the triplet distance measure that can only be defined for *rooted* phylogenetic trees and networks, we only focus on the rooted case of phylogenetic networks.

In 2009, Cardona *et al.* [18] gave a generalization of the rooted triplet distance from phylogenetic trees to phylogenetic networks. However in 2012, Gambette and Huber [33] gave another generalization that is closer to the definition of the extensively studied triplet distance for trees. Moreover and as we will see below, the latter has already been used to create efficient algorithms for computing the triplet distance between phylogenetic networks, which is why in this thesis we consider the generalization in [33]. From here on in this section, when we use the word "tree" we imply a "rooted phylogenetic tree", and similarly when we use the word "network" we imply a "rooted phylogenetic network". In the remaining part of this section, we formally define the problem, we give a survey of previous results, and finally we provide our contributions.

1.4.1 Preliminaries and Problem Definition

A network $N' = (V, E)$ is a rooted directed acyclic graph that is unordered, with distinctly labeled leaves and with no nodes that have both in-degree 1 and out-degree 1. For a node u in a given network N', let $in(u)$ and $out(u)$ be the in-degree and out-degree of u. We distinguish the nodes of N' into the following types:

1. A node u is a *root node* if $in(u) = 0$ and $out(u) \geq 1$. There is exactly one such node in N'.

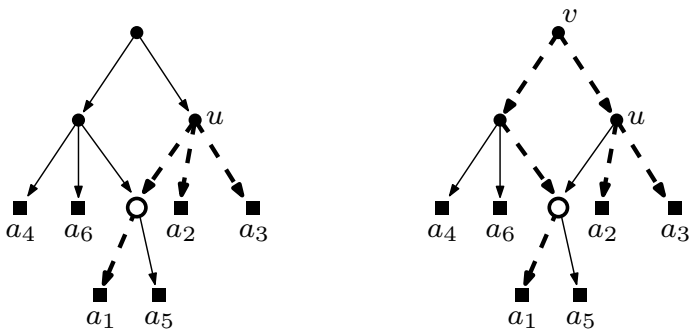


Figure 1.5: A network on the leaf label set $\{a_1, a_2, a_3, a_4, a_5, a_6\}$. The node illustrated with a circle is a reticulation node. (a) The leaves a_1, a_2, a_3 can be used to induce the fan triplet $a_1|a_2|a_3$ (indicated by the bold dashed lines). (b) The same leaves can also be used to induce the resolved triplet $a_2a_3|a_1$ (indicated by the bold dashed lines).

2. A node u is an *internal node* if $out(u) \geq 1$. Note that the root node is also an internal node.
3. A node u is a *leaf node*, if $out(u) = 0$ and $in(u) = 1$.
4. A node u is a *reticulation node*, if $in(u) \geq 2$ and $out(u) \geq 1$.

By definition, N' cannot have a node u with $in(u) > 1$ and $out(u) = 0$.

We say that the fan triplet $x|y|z$ is consistent with N', if there exists a tree topology induced by the leaves x, y, and z in N' with a node u in N' being the only internal node of this tree. In other words, there exists an internal node u in N' and three directed paths of non-zero length from u to x, y, and z that are node-disjoint except for in u. Similarly, we say that a resolved triplet $xy|z$ is consistent with N', if there exists a binary tree topology induced by the leaves x, y, and z in N' with u and v in N' being the two internal nodes of this tree. In other words, there exist two different internal nodes u and v in N', such that there are four directed paths of non-zero length from u to v, from v to x, from v to y, and from u to z that are node-disjoint except for in u and v, and furthermore, the path from u to z does not pass through v. Note that if N' is a tree, the definition of a triplet is equivalent to the one in Section 1.3.1. In Figure 1.5 we have an example network, as well as two different triplets, a fan triplet and a resolved triplet, that are induced by the same set of three leaves. Observe that this is impossible in a tree, i.e., in a tree any set of three leaves can be used to form either a resolved triplet or a fan triplet.

Given two networks N_1 and N_2 that are built on the same leaf label set of size n, the triplet distance $D(N_1, N_2)$ is defined as the number of triplets that are induced by only one of the two networks. More formally, if $S(N_1, N_2)$ is the total number of shared triplets between N_1 and N_2, we have:

$$D(N_1, N_2) = S(N_1, N_1) + S(N_2, N_2) - 2S(N_1, N_2) \ . \tag{1.1}$$

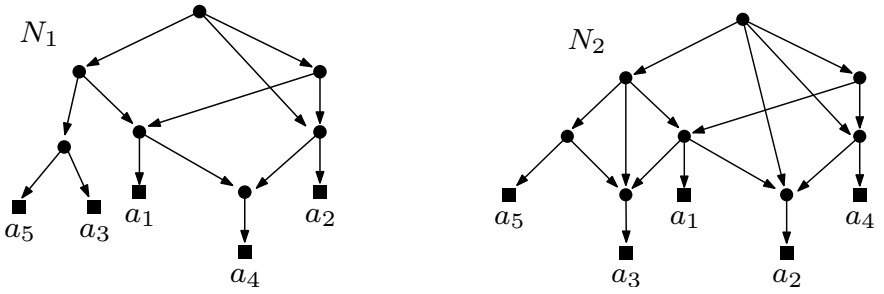


Figure 1.6: Two networks N_1 and N_2 that are built on the same leaf label set $\{a_1, a_2, a_3, a_4, a_5\}$. Here, $D(N_1, N_2) = 11$. Some examples of shared triplets are $a_1a_2|a_5$, $a_1|a_2|a_5$, $a_1a_4|a_3$, $a_3a_4|a_2$, and $a_2|a_4|a_5$. All the 11 triplets consistent with only one network are $a_1|a_2|a_4$, $a_1a_2|a_4$, $a_1|a_3|a_5$, $a_1a_3|a_5$, $a_2a_3|a_4$, $a_2|a_3|a_5$, $a_2a_3|a_5$, $a_4a_5|a_2$, $a_2a_5|a_4$, $a_3|a_4|a_5$, and $a_3a_4|a_5$.

This definition of the triplet distance differs from the definition we used for trees in Section 1.3.1. More precisely, when N_1 and N_2 are trees, the value for $D(N_1, N_2)$ is twice the value we would get if we were to use the definition in Section 1.3.1. However, as shown in Figure 1.5 and also discussed in more detail in [44, Section 3.2], three leaves can appear as both a resolved triplet and a fan triplet in a given network, and Equation 1.1 takes that into account. An example can be found in Figure 1.6.

Gambette and Huber, in their extension of the triplet distance from trees to networks in [33], used a parameter called *level* to measure the tree-likeness of a network, i.e., a parameter to measure how close a network is to being a tree. As the algorithms below use this parameter in their analysis, we formally define the level of a network here. We call an undirected graph H *biconnected* if it is not possible to remove exactly one node from H to make it disconnected. Let $U(N')$ be the graph created by removing the directions of the edges in N'. We say that H' is a *biconnected component* of $U(N')$ if H' is a maximal biconnected subgraph of $U(N')$. Finally, we say that the *level of N' is k*, or *N' is a level-k network*, if there are at most k reticulation nodes in any biconnected component of $U(N')$. By definition a level-0 network corresponds to a tree and a level-1 network corresponds to a *galled tree* [36]. In the example of Figure 1.6, the level of N_1 is 3 and the level of N_2 is 4.

In all algorithms for computing $D(N_1, N_2)$, the main focus is time. To help with the overview provided below, the following notation is used. We assume that the input networks are $N_1 = (V_1, E_1)$ and $N_2 = (V_2, E_2)$. The time analysis is then performed on the following parameters. N_1 and N_2 have the same leaf label set Λ of size n. Let $N = \max(|V_1|, |V_2|)$ and $M = \max(|E_1|, |E_2|)$. If k_1 is the level of N_1 and k_2 is the level of N_2, we have $k = \max(k_1, k_2)$. Finally, if d_1 is the maximum in-degree of any node in N_1 and similarly we have d_2 for N_2, then $d = \max(d_1, d_2)$.

1.4.2 Previous Results

When $k = 0$, both N_1 and N_2 are trees, and we considered this case in Section 1.3. As a reminder, we managed to bridge the theoretical and practical gap that existed for this problem before in the literature, by providing two new algorithms in Chapter 2, one optimized for the case where the two trees are binary, and the other for the general unrestricted case. Both algorithms run in $O(n \log n)$ time and are the first to scale to external memory by incurring $O(\frac{n}{B} \log_2 \frac{n}{M})$ I/Os in the cache oblivious model. We also provided an implementation of the algorithms, as well as extensive experiments illustrating their practical performance.

When $k = 1$, we have that N_1 and N_2 are galled trees. Moreover, we have $N = \Theta(n)$ and $M = \Theta(n)$. There are two major results for this case in the literature. In 2014, Jansson and Lingas [44] gave the first non-trivial algorithm that has an $O(n^{2.687})$ running time, by reducing the problem of computing $D(N_1, N_2)$ to the problem of counting triangles in a graph. A few years later in 2017, Jansson *et al.* [49], gave an $O(n \log n)$-time algorithm, by reducing the problem to combining the outputs of an algorithm on a constant number of instances when $k = 0$. An implementation of the $O(n \log n)$-time algorithm was given in the journal version of the paper in [53]. Note that both algorithms do not impose any restrictions on the degrees of the nodes in the input networks.

When $k > 1$, Byrka *et al.* [15] considered networks with restrictions on the degrees of their nodes. More precisely, according to [15, Section 2], a network $N' = (V, E)$ is defined such that there is one root node with in-degree 0 and out-degree 2, and all other nodes have either in-degree 1 and out-degree 2, or in-degree 2 and out-degree 1, or in-degree 1 and out-degree 0. By definition, nodes with either in-degree or out-degree larger than 2 are not allowed, i.e., N' cannot have any fan triplets. Moreover, nodes with both in-degree 2 and out-degree 2 are not allowed either. Because of the degree constraints in N', we have $|V| = \Theta(|E|)$. For such a network N', a data structure is given in [15, Lemma 2], call it D_s, that can be constructed in $O(|V|^3)$ time, and then be used to answer any query about the consistency of a resolved triplet with N' in $O(1)$ time. This result was then extended to exploit the level of N'. More precisely, if the level of N' is k, a new data structure is given in [15, Lemma 3], call it D'_s, that can support the same queries while only requiring $O(|V| + |V|k^2)$ preprocessing time. Algorithm 1 shows how we can use these data structures to compute $D(N_1, N_2)$. The input is the two networks N_1 and N_2. There is a preprocessing step (lines 2–4), where we construct for both N_1 and N_2, either D_s or D'_s. In lines 6–13 we use either D_s or D'_s to count the total number of shared triplets. The main procedure is $D()$, and it uses Equation 1.1 to compute $D(N_1, N_2)$. Notice that the procedure $S()$ only focuses on resolved triplets, since as we also mentioned above, fan triplets cannot be induced by the networks considered in [15]. As

Algorithm 1 Computing $D(N_1, N_2)$ using the data structures presented by Byrka *et al.* [15].

```
1: procedure PREPROCESSING(N_1, N_2)          ▷ Building the data structures
2:     build the data structure of [15, Lemma 2 or Lemma 3] for N_1 and N_2
3:     let D_1 and D_2 be the data structures for N_1 and N_2
4:     return (D_1, D_2)

5: procedure S(N_1, N_2, D_1, D_2)            ▷ Finding the shared triplets
6:     shared = 0
7:     for three different leaves x, y and z do
8:         for τ ∈ {xy|z, xz|y, yz|x} do
9:             Using D_1 check the consistency of τ with N_1
10:            Using D_2 check the consistency of τ with N_2
11:            if τ is consistent with both N_1 and N_2 then
12:                shared = shared + 1
13:    return shared

14: procedure D(N_1 = (V_1, E_1), N_2 = (V_2, E_2))   ▷ Computing D(N_1, N_2)
15:    (D_1, D_2) = PREPROCESSING(N_1, N_2)
16:    return S(N_1, N_1, D_1, D_1) + S(N_2, N_2, D_2, D_2) − 2S(N_1, N_2, D_1, D_2)
```

for the running time of the algorithm, first, there is an $\mathrm{O}(n^3)$ term because of the procedure $S()$. If D_s is used in the preprocessing step, the total running time becomes $\mathrm{O}(|V_1|^3 + |V_2|^3 + n^3)$. By the definition of N, the total running time is then simplified to $\mathrm{O}(N^3 + n^3)$. If D'_s is used in the preprocessing step, the total running time becomes $\mathrm{O}(|V_1| + k_1^2|V_1| + |V_2| + k_2^2|V_2| + n^3)$. By the definition of k and N, the total running time becomes $\mathrm{O}(N + k^2N + n^3)$. To the best of our knowledge, no implementations of the above algorithms currently exist.

When $k > 1$ and without having any restrictions on the degrees of the nodes in N_1 and N_2, no non-trivial algorithms are available in the literature. Even an algorithm that is based on enumerating over all $4\binom{n}{3}$ triplets and checking their consistency with N_1 and N_2, turns out to be not so trivial. As also shown in Section 1.3.2, the main reason why it is easy to naively check the consistency of a triplet with a tree is because of how simple a tree as a structure is, e.g., any two leaves have exactly one lowest common ancestor, and that ancestor has exactly one path to each leaf. In a network and following [44], a lowest common ancestor between two nodes u and v is defined to be a node w that is an ancestor of both u and v, and there does not exist any descendant of w that is also an ancestor of u and v. Then, it is trivial to see that two nodes can have more than one lowest common ancestors (see Figure 1.7). However, it turns out there is a result from 1980 by Fortune *et al.* [29] that, for a given directed acyclic graph G can check in polynomial time if a pattern graph P, that is also acyclic and directed, is homeomorphic

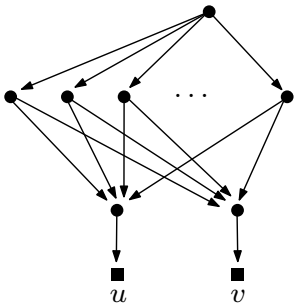


Figure 1.7: An example of a network. Every internal node, except the root, is a lowest common ancestor of the leaves u and v.

to a subgraph of G. The time complexity of the algorithm depends on the structure of G and P. Using this result on our problem, we can check the consistency of a fan triplet in $\Omega(N^5)$ time and the consistency of a resolved triplet in $\Omega(N^7)$ time. Thus, the total running time of the naive algorithm that enumerates over all possible triplets and checks their consistency with N_1 and N_2 becomes $\Omega(N^7n^3)$.

1.4.3 Our Contributions

In Table 1.2 we have a summary of previous and new results. Our contributions are two algorithms for computing the triplet distance between two phylogenetic networks that work for any $k \geq 0$ while not imposing any restrictions on the degrees of the input networks. This means that unlike the result in [15], our algorithms can also detect fan triplets that might exist in the input networks. The running time of the first algorithm is $O(N^2M + n^3)$, and the second algorithm $O(M + k^3d^3n + n^3)$. Finally, we provide implementations of the two algorithms in C++, that is publicly available at `https://github.com/kmampent/ntd`, as well as extensive experiments on both simulated and real datasets illustrating their practical performance.

Table 1.2: Previous and new results for computing $D(N_1, N_2)$, where N_1 and N_2 are two level-k networks built on the same leaf label set of size n.

Year	Reference	k	Degrees	Time
1980	Fortune *et al.* [29]	arbitrary	arbitrary	$\Omega(N^7n^3)$
2010	Byrka *et al.* [15]	arbitrary	binary	$O(N^3 + n^3)$
2010	Byrka *et al.* [15]	arbitrary	binary	$O(N + k^2N + n^3)$
2017	Brodal *et al.* [11, 12]	0 (trees)	arbitrary	$O(n \log n)$
2019	Jansson *et al.* [53]	1 (galled trees)	arbitrary	$O(n \log n)$
2019	new [Chapter 3]	arbitrary	arbitrary	$O(N^2M + n^3)$
2019	new [Chapter 3]	arbitrary	arbitrary	$O(M + k^3d^3n + n^3)$

In the remaining part of this subsection, we give a high level overview of our algorithms. Details can be found in Chapter 3 and Appendix B.

The First Algorithm. The first algorithm is described in Section 3.2. It consists of a preprocessing step and a triplet distance computation step. In the preprocessing step, for a network $N_i = (V_i, E_i)$ we construct another graph N_i^f in $O(|V_i|^2|E_i|)$ time. This graph has the property that the fan triplet $x|y|z$ is consistent with N_i if and only if there is a path from the node s to the node (x, y, z) in N_i^f. Similarly, we construct another graph called N_i^r in $O(|V_i|^2|E_i|)$ time that has a similar property but now for resolved triplets. More precisely, the resolved triplet $x|yz$ is consistent with N_i if and only if there is a path from the node s to the node (x, y, z) in N_i^r. Given these two graphs, we build two $n \times n \times n$ tables A_i^f and A_i^r, such that $A_i^f[x][y][z] = 1$ if there is a path from the node s to the node (x, y, z) in N_i^f, and 0 otherwise. Similarly, $A_i^r[x][y][z] = 1$ if there is a path from the node s to the node (x, y, z) in N_i^r, and 0 otherwise. Building these two tables is done by a depth first traversal of the two graphs N_i^f and N_i^r, thus requiring $O(|V_i|^2|E_i|)$ time as well. Finally and in the triplet distance computation step, for two given networks N_1 and N_2, the algorithm enumerates over all $4\binom{n}{3}$ triplets and uses the A_1^f, A_1^r, A_2^f, and A_2^r tables to determine in $O(1)$ time the consistency of a triplet with N_1 and N_2.

For the time analysis, the preprocessing step takes $O(|V_1|^2|E_1| + |V_2|^2|E_2|)$ time and the triplet distance computation step $O(n^3)$. By the definition of N and M, the total running time becomes $O(N^2M + n^3)$. The space is also $O(N^2M + n^3)$.

The Second Algorithm. In Section 3.3, we extend the algorithm from Section 3.2 to also take into consideration the level of the input networks and thus make the running time adaptive to the levels of the networks. The general approach is the same, i.e., there is a preprocessing step, where all the necessary data structures are built that let us later in the triplet distance computation step determine in $O(1)$ time the consistency of any triplet with either of the two input networks.

In Lemma 8 we prove that the biconnected components of a network N_i are edge-disjoint but not node-disjoint. The biconnected components of a network following the degree constraints in [15] would not only be edge-disjoint, but node-disjoint as well. This extra feature of the biconnected components being node-disjoint was directly exploited in [15, Lemma 3] when building the data structure D'_s, which is why D'_s cannot be used to work for the unrestricted networks that we consider in our paper.

In the preprocessing step, for a network $N_i = (V_i, E_i)$ we build a tree called *block tree* with leaf label set Λ. This tree is created by carefully contracting every biconnected component (a "biconnected component" is referred to as a "block" in the paper, hence the name of the tree) of N_i into a node. An example can be found in Figure 3.5. Every node of this tree corresponds to

a biconnected component of N_i and vice versa, following a 1-to-1 correspondence. Roughly speaking, for every biconnected component of N_i we create a small network, called a *contracted block network*, that has almost the same structure as the biconnected component (e.g., see Figure 3.6). For every contracted block network, we then build the data structures that we had in the preprocessing step of the first algorithm. In Lemma 10 we show that building the block tree can be done in $O(|E_i|)$ time. In Lemma 12 we show how to build every contracted block network of N_i in $O(|E_i|+n^2)$ time. We pay the extra n^2 factor so that we can then in Lemma 13 have an algorithm that builds all data structures for every contracted block network of N_i in $O(n(k_i^3d_i^3+1))$ time. Finally and in the triplet distance computation step, in Lemmas 14, 15, 16, and 17 we show how given the data structures from the preprocessing step, we can determine in $O(1)$ time the consistency of any fan and resolved triplet with N_i. From a high level perspective, if x, y, and z are the leaves of the triplet, we check the lowest common ancestors of every pair (x, y), (x, z), and (y, z) in the block tree. To be able to find these lowest common ancestors efficiently, we also build an $n \times n$ table in $O(n^3)$ time in the preprocessing step. There are ways in the literature to perform this step faster, e.g., by only using the LCA data structure in [7] and thus only spending $O(n)$ time and space. However as we will see below, this approach would not improve the final time bound of our algorithm. Depending on the lowest common ancestors in the block tree, we can then either directly report whether or not the triplet is consistent with the network, or we have to make queries to the data structures of the contracted block networks that correspond to the lowest common ancestors.

For the time analysis, just like with the first algorithm we build the data structures for both N_1 and N_2. Hence, the total preprocessing time is bounded by $O(|E_1| + |E_2| + n(k_1^3d_1^3 + k_2^3d_2^3) + n^3)$. In the triplet distance computation step, we get an extra n^3 factor which does not make the previous bound worse. By the definition of N, M, k, and d the total running time then becomes $O(M + k^3d^3n + n^3)$. Similarly to the first algorithm, the space does not change, i.e., it is $O(M + k^3d^3n + n^3)$.

1.4.4 Open Problems and Future Work

Future work involves creating algorithms that are more efficient than the ones we presented in this thesis. This could mean coming up with more efficient data structures that can still answer triplet consistency queries in $O(1)$ time, or use a completely different approach. Iterating over every possible triplet will immediately incur a $O(n^3)$ term in the running time, which is very slow for many practical purposes. For the case of trees, i.e., when $k = 0$, the $O(n \log n)$-time algorithms are based on completely different techniques. It would be useful to see if we could extend those techniques to the case of networks. A main reason why this is not trivial is because the algorithms for trees exploit the fact that the lowest common ancestor between a pair of leaves

in a tree always corresponds to exactly one node. As we argued above and gave an example in Figure 1.7, this is not necessarily the case for networks, i.e., in a given network N_i two leaves can even have an arbitrary number of lowest common ancestors.

Finally, it might be useful to create more biologically meaningful definitions for the rooted triplet distance in networks. As discussed in the concluding remarks of Section 3.5, the definition of $D(N_1, N_2)$ does not consider the total number of different occurrences of a given triplet in a network, i.e., a triplet appearing once in a network or 1000 times has the exact same meaning. However, the number of occurrences says something about the structure of the two networks that is currently not captured by the definition of $D(N_1, N_2)$. One even needs to carefully define the number of occurrences of a triplet in a network, since different definitions can lead to different outcomes (for a more detailed discussion on this see Section 3.5).

1.5 Building Small and Informative Supertrees

In the previous two sections, we assumed that the phylogenetic trees to be compared are available. However when studying the evolutionary history of species, usually during the early stages of a study only basic biological data is available (e.g., DNA or morphological data, triplets). This data is then used by algorithms to construct a phylogenetic tree, commonly referred to in the literature as a *supertree* [9], that represents the data as much as possible. The third and final problem considered in this thesis is about algorithms that can construct such supertrees efficiently.

In a supertree problem the input can be a set of small, accurate trees over overlapping subsets of n species, and the goal is to construct one large tree that has the n species as leaves and at the same time preserves the most

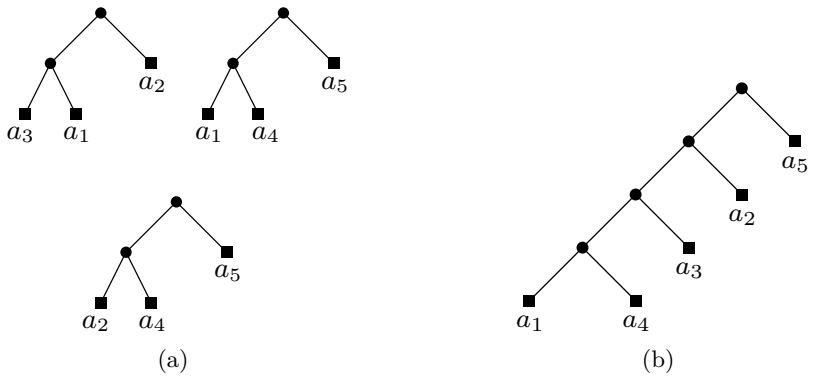


Figure 1.8: An input/output example for a supertree problem. (a) A set $\mathcal{R} = \{a_3a_1|a_2, a_1a_4|a_5, a_2a_4|a_5\}$ of resolved triplets. (b) A tree inducing $\mathcal{R}$.

information possible induced by the small trees in the input. Depending on the type of input and information that we want to preserve in the output tree, there are several different versions of this problem. For example, in one version the input can be a set of rooted binary trees with three leaves, i.e., resolved triplets, and the output a tree, if it exists, that induces every resolved triplet in the input. Figure 1.8 contains a concrete example. In another version, we could have the same requirements but now with fan triplets also allowed in the input. Efficient algorithms that build these supertrees are necessary, since the trees produced can then be used as input for the tree comparison algorithms presented in the previous two sections, which in turn can help with evolutionary inferences. In the remaining part of this section, we provide a precise definition of the supertree problem considered in this thesis, we give a survey of previous results, and finally we provide our contributions.

1.5.1 Preliminaries and Problem Definition

We consider *rooted phylogenetic trees*, i.e., rooted unordered trees that are distinctly leaf labeled. In this section, when referring to a "tree" or a "supertree" we imply a "rooted phylogenetic tree". The definition of a resolved and fan triplet follows from Section 1.3. For a tree T, let $rt(T)$ be the set of all triplets induced by T. We say that T is consistent with a set of triplets $\mathcal{R}$, or equivalently the set of triplets $\mathcal{R}$ is consistent with T, if every triplet in $\mathcal{R}$ is induced by T. By definition, if $\mathcal{R}$ is consistent with T we have $\mathcal{R} \subseteq rt(T)$. Finally, we assume that any set $\mathcal{R}$ contains triplets over a leaf label set of size n.

We consider the supertree problem called *q-maximum rooted triplets consistency* (q-MAXRTC). The input is a set $\mathcal{R}$ of *resolved* triplets, as well as a parameter q, and the objective is to find a tree with exactly q internal nodes that is consistent with the maximum number of resolved triplets from $\mathcal{R}$. Part of the motivation behind q-MAXRTC is a new algorithm for computing the triplet distance $D(T_1, T_2)$ between two trees T_1 and T_2 that are built on the same leaf label set of size n (for a formal definition see Section 1.3), so we will revisit this problem again below.

Below, we present several polynomial-time approximation algorithms for q-MAXRTC. As a reminder, let P be a maximization problem and I any input instance for P. Let A be an algorithm that solves P, $V_A(I)$ the value of the solution returned by A on the input I, and $OPT(I)$ the value of the optimal solution on the input I. Let $0 \leq r \leq 1$. We say that A is an r-approximation algorithm with *relative ratio* r, if $V_A(I) \geq r \cdot OPT(I)$. We also say that A is an r-approximation algorithm with *absolute ratio* r, if $V_A(I) \geq r|I|$ for any input I. By definition, an approximation algorithm with absolute ratio r, immediately implies an approximation algorithm with relative ratio r. For the q-MAXRTC problem we have $I = \mathcal{R}$. In our results below, we mainly present approximation algorithms with absolute ratios, thus unless otherwise stated, when we refer to any ratio r we imply an absolute ratio.

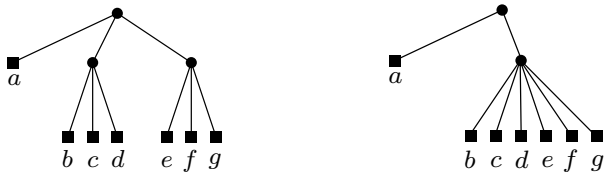


Figure 1.9: Let $\mathcal{R} = \{bc|a, bd|a, ef|a, eg|a\}$ be the example from [13]. (a) The tree with three internal nodes returned by the BUILD algorithm [2]. (b) A tree with two internal nodes that is consistent with $\mathcal{R}$.

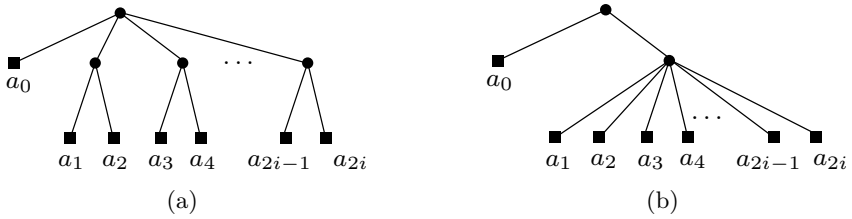


Figure 1.10: Let $\mathcal{R} = \{a_1a_2|a_0, a_3a_4|a_0, \ldots, a_{2i-1}a_{2i}|a_0\}$ be the example from [48]. (a) The tree with $i + 1$ internal nodes returned by the BUILD algorithm [2]. (b) A tree with two internal nodes that is consistent with $\mathcal{R}$.

1.5.2 Survey of Previous Results

The q-MAXRTC problem is essentially a combination of the *minimally resolved supertree* (MINRS) [48] problem and the *maximum rooted triplets consistency* (MAXRTC) [13] problem. In MINRS, for a set of resolved triplets $\mathcal{R}$ the objective is to find, if it exists, a tree with the minimal number of internal nodes that is consistent with the entire set $\mathcal{R}$. In MAXRTC we are given a set $\mathcal{R}$ of resolved triplets and the goal is to find a tree that is consistent with the largest number of triplets from $\mathcal{R}$.

For the version of MINRS without the minimality constraint on the internal nodes, Aho *et al.* [2] gave a polynomial-time algorithm called BUILD. The BUILD algorithm receives as input a set $\mathcal{L}$ of lowest common ancestor constraints on pairs of leaves, e.g., the pair $\langle i, j\rangle < \langle i, k\rangle$ means that the lowest common ancestor of the leaves with labels i and j is a proper descendant of the lowest common ancestor of the leaves with labels i and k. We can easily translate $\mathcal{R}$ into such a set $\mathcal{L}$, for which $|\mathcal{L}| = 2|\mathcal{R}|$, by adding to $\mathcal{L}$ for every triplet $ij|k$ in $\mathcal{R}$ the pairs $\langle i, j\rangle < \langle i, k\rangle$ and $\langle i, j\rangle < \langle j, k\rangle$. Bryant [13] showed that for $\mathcal{R} = \{bc|a, bd|a, ef|a, eg|a\}$ the BUILD algorithm will return a tree with three internal nodes, even though it is possible to construct a tree with only two internal nodes that induces every triplet from $\mathcal{R}$ (see Figure 1.9). Jansson *et al.* [48] simplified the example to $\mathcal{R} = \{bc|a, ef|a\}$, and extended

it to show that for some input instances the tree returned by BUILD can have $\Omega(n)$ internal nodes more than necessary (see Figure 1.10).

An internal node in a tree implies that at the point of time indicated by the internal node, something happened in the evolutionary history of species. A common criticism on the supertree construction algorithms is that they tend to make statements about the evolutionary history of species which is not directly supported by the data, thus creating false groupings of the leaves, commonly referred to as "spurious novel clades" [9]. Moreover, in science we generally prefer simple explanations for a set of given observations, thus if we can create a smaller tree that induces all triplets from a triplet set $\mathcal{R}$, then due to its simplicity this tree is commonly more preferred over another larger, more complex, tree. MINRS tried to address these concerns by having the minimality constraint on the internal nodes. Jansson *et al.* showed in [48] that the decision version of MINRS is NP-complete for $q \geq 4$ and polynomial-time solvable for $q \leq 3$, where q denotes the number of internal nodes in the produced output tree.

Both MINRS and its version without the minimality constraint have the problem that they are very sensitive to outliers. For example, it could be that every triplet from $\mathcal{R}$ except for one triplet can be induced by a single tree. However, in both problems the corresponding algorithms would not report any tree in the output. This is what motivated MAXRTC. Bryant [13] proved that MAXRTC is NP-hard. In 1999 Gąsieniec *et al.* [34] gave the first polynomial-time approximation algorithm that for any input set $\mathcal{R}$ returns a caterpillar tree, i.e., a tree in which every internal node has at most one non-leaf child, that induces at least $\frac{1}{3}|\mathcal{R}|$ triplets from $\mathcal{R}$. Hence, their algorithm is a polynomial-time $\frac{1}{3}$-approximation algorithm. The number of internal nodes in the output tree is arbitrary and in the worst case it can be as big as $n-1$, i.e., the supertree produced is a binary tree. In 2004, Wu [77] gave a bottom-up algorithm for MAXRTC and showed that it performs well in practice. However, the approximation ratio of this algorithm was left as an open problem. Several years later and in 2010, Byrka *et al.* [15] modified the algorithm by Wu [77] to create an polynomial-time algorithm that has an approximation ratio of $\frac{1}{3}$. This algorithm always produces a binary tree, meaning that the number of internal nodes of the supertree produced is exactly $n-1$. In the same year of 2010, Byrka *et al.* [15] gave another polynomial-time $\frac{1}{3}$-approximation algorithm, based on a randomized approach, that also always returns a binary tree as an output.

1.5.3 Our Contributions

Note that every previous algorithm for MAXRTC produces trees with an arbitrary number of internal nodes, some even always produce binary trees, i.e., trees with the maximum number of internal nodes. However, due to the issue of spurious clades described above, we might still want to have small and sim-

Table 1.3: Previous and new results for computing q-MAXRTC. The abbreviations "abs." and "rel." correspond to "absolute" and "relative" respectively.

Year	Reference	Deterministic	q	Approximation	Type
1999	Gąsieniec *et al.* [34]	yes	unbounded	1/3	abs.
2010	Byrka *et al.* [15, 16]	yes	$n-1$	1/3	abs.
2019	new [Section 4.3.1]	no	2	1/2	rel.
2019	new [Section 4.3.1]	yes	2	1/4	rel.
2019	new [Theorem 5]	yes	2	4/27	abs.
2019	new [Theorem 7]	yes	$q \geq 3$	$\frac{1}{3} - \frac{4}{3(q+q \bmod 2)^2}$	abs.

ple trees that induce a large number of triplets from the input set $\mathcal{R}$. Finally, as we will see below, in the design of phylogenetic tree comparison algorithms, trees with fewer internal nodes can admit faster running times. More precisely, in Section 4.5 we give an algorithm for computing the rooted triplet distance between two rooted trees that runs in $O(qn)$ time, where q is the number of internal nodes in the smaller tree. Note that this new algorithm is faster than the algorithm we presented in Chapter 2, whenever $q = o(\log n)$. Interestingly, this algorithm even happens to be optimal when $q = O(1)$. In Table 1.3 we list previous and new results for computing q-MAXRTC. An implementation of q-MAXRTC is publicly available at `https://github.com/kmampent/qMAXRTC`. An implementation of our new $O(qn)$-time triplet distance algorithm is available at `https://github.com/kmampent/qtd`. Finally, we provide extensive experiments illustrating the practical performance of both algorithms. In the remaining part of this subsection, we give a more detailed overview of our paper and results. Additional details can be found in Chapter 4 and Appendix C.

In Section 4.2 we study the computational complexity of q-MAXRTC, and present some inapproximability results. More precisely, in Theorem 4 we prove that q-MAXRTC is NP-hard for any fixed $q \geq 2$. We achieve this by a reduction from MAX q-CUT [55]. This in turn yields some inapproximability results, presented in Corollaries 4 and 5.

In Section 4.3 we present our approximation algorithms for q-MAXRTC. We consider the problem MAX 3-AND, where we are given a logical formula consisting of a set of clauses S, each being a conjunction of three literals from a set of Boolean variables, and the goal is to find an assignment of values to the variables that satisfies the maximum number of clauses from S. We then prove in Lemma 19 that if MAX 3-AND can be approximated within a factor of r, then 2-MAXRTC can also be approximated within a factor of r. Because of Lemma 19, we can then use any approximation algorithm for MAX 3-AND to directly obtain an approximation algorithm for 2-MAXRTC. Hence, using the result in [78] we obtain a randomized polynomial-time $\frac{1}{2}$-approximation algorithm with relative ratio for 2-MAXRTC. Moreover, using the result in [74] we obtain a deterministic polynomial-time $\frac{1}{4}$-approximation

algorithm with relative ratio for 2-MAXRTC. In Lemma 20 we give a randomized polynomial-time $\frac{4}{27}$-approximation algorithm for q-MAXRTC and then show in Theorem 5 how to derandomize in order to obtain a deterministic $O(|\mathcal{R}|)$-time algorithm that achieves the same $\frac{4}{27}$ approximation ratio. Interestingly, we show in Theorem 6 that this algorithm is optimal for 2-MAXRTC when considering absolute approximation ratios. Finally, in Theorem 7 we show how to extend the algorithm from Theorem 5 to obtain a deterministic $O(q|\mathcal{R}|)$-time algorithm that scales with q and achieves the approximation ratio $(\frac{1}{3} - \frac{4}{3(q+q \bmod 2)^2})$.

In Section 4.4 we provide an implementation of our new algorithms for q-MAXRTC with absolute approximation ratios, as well as experiments, on both simulated and real datasets, illustrating their practical performance. As an extreme example, we show that with only nine internal nodes our algorithms can capture on average 80% of the rooted triplets from some recently published trees, each having between 760 and 3081 internal nodes.

Finally, in Section 4.5 we give our new $O(qn)$-time algorithm for computing the rooted triplet distance between two trees, built on the same leaf label set of size n and with q being the number of internal nodes in the smaller tree. Let T_1 be the tree with the q internal nodes. From a high level, this algorithm has a preprocessing step and a counting of shared triplets step. In the preprocessing step, we change the labels of the leaves in T_2 according to their discovery time in a depth first traversal of T_2. We then transfer this change of labels to the leaves in T_1. Then for T_1, we build an $q \times n$ table C in $O(qn)$ time that can later be used to answer in $O(1)$ time any query asking for the total number of leaves in a subtree defined by a node v in T_2 that are also in a subtree defined by an internal node u in T_1. In the counting step, we use the idea of anchoring the triplets in edges, introduced in Chapter 2 and Section 2.4.2 for high degree trees. In Lemma 22 we show that we can use the C table to compute all shared triplets that are anchored in a fixed edge of T_2 in $O(q)$-time. Since there are $O(n)$ edges in T_2, the running time of the algorithm becomes $O(qn)$. To conclude, in Section 4.5.3 we provide some experimental results, where we compare the space and time performance of our algorithm against previous algorithms. Our experiments indicate that our implementation uses less space and is faster than the state-of-the-art algorithms [11, 46] for this problem for large inputs, e.g., when $n = 1,000,000$ and $q \leq 50$.

1.5.4 Open Problems and Future Work

In Theorem 6 we prove that when $q = 2$, the optimal absolute approximation ratio is $\frac{4}{27}$. In Theorem 5 we give a polynomial-time deterministic algorithm achieving that ratio. The optimal absolute approximation ratio for $q \geq 3$ is an open problem, as well as the existence of corresponding algorithms achieving the optimal ratio. Another open problem and possible direction for future work, is the existence of approximation algorithms in the weighted case

of q-MAXRTC, where a weight is assigned to every triplet in the input and the objective is to build a tree that maximizes the total weight of the triplets induced from $\mathcal{R}$. This would address the case where some triplets in $\mathcal{R}$ are more important than others. Another possible interesting future research direction would be to consider the following combination of the problem studied in [47] and MAXRTC: Given a set of triplets $\mathcal{R}$ on a leaf label set of size n and a parameter ℓ, build a tree T with ℓ leaves such that $|rt(T) \cap \mathcal{R}|$ is maximized. Finally, it would be interesting to know whether or not it is possible to create an algorithm for computing the rooted triplet distance between two trees that are built on the same leaf label set of size n in $\mathrm{O}(q_1 q_2 + n)$ time, where q_1 is the number of internal nodes in T_1 and q_2 the number of internal nodes in T_2.

Chapter 2

Cache Oblivious Algorithms for Computing the Triplet Distance Between Trees

[11] Gerth Stølting Brodal and Konstantinos Mampentzidis. Cache Oblivious Algorithms for Computing the Triplet Distance Between Trees. In *25th Annual European Symposium on Algorithms (ESA 2017)*, volume 87 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. 2017.

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

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

2.1 Introduction

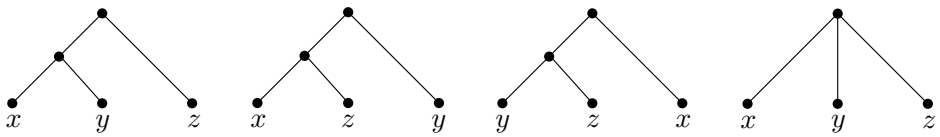
Trees are data structures that are often used to represent relationships. For example in the field of Biology, a tree can be used to represent evolutionary relationships, with the leaves corresponding to species that exist today, and internal nodes to ancestor species that existed in the past. For a fixed set of n species, different data (e.g., DNA, morphological) or construction methods (e.g., Q* [8], neighbor joining [65]) can lead to trees that look structurally different. An interesting question that arises then is, given two trees T_1 and T_2 over n species, how different are they? An answer to this question could potentially be used to determine whether the difference is statistically significant or not, which in turn could help with evolutionary inferences.

Several distance measures have been proposed in the past to compare two trees that are *unordered*, i.e., trees in which the order of the siblings is not taken into account. A class of them includes distance measures that are based on how often certain features are different in the two trees. Common distance measures of this kind are the Robinson-Foulds distance [64], the triplet distance [25] for rooted trees and the quartet distance [26] for unrooted trees. The Robinson-Foulds distance counts how many leaf bipartitions are different, where a bipartition in a given tree is generated by removing a single edge from the tree. The triplet distance is only defined for rooted trees, and counts how many leaf triples induce different topologies in the two trees. The counterpart of the triplet distance for unrooted trees, is the quartet distance, which counts how many leaf quadruples induce different topologies in the two trees.

Algorithms exist that can efficiently compute these distance measures. The Robinson-Foulds distance can be optimally computed in $O(n)$ time [24]. The triplet distance can be computed in $O(n \log n)$ time [12]. The quartet distance can be computed in $O(dn \log n)$ time [12], where d is the maximal degree of any node in the two input trees.

The above bounds are in the RAM model. Previous work did not consider any other models, for example external memory models like the I/O model [1] and the cache oblivious model [32]. Typically when hearing about algorithms for external memory models, one might (sometimes incorrectly) think of only algorithms that have to deal with large amounts of data. Hence, any practical improvement that comes from an algorithm that scales to external memory compared to an equivalent that does not, can only be noticed if the inputs are large. However, this is not necessarily the case for cache oblivious algorithms. A cache oblivious algorithm, if built and implemented correctly, can take advantage of the L1, L2, and L3 caches that exist in the vast majority of computers and give a significant performance improvement even for small inputs.

A trivial modification of the algorithm in [24], can give a cache oblivious algorithm for computing the Robinson-Foulds distance that achieves the sorting bound, by requiring $O(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{M})$ I/Os instead of $O(n)$ I/Os for the



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

Figure 2.1: All possible topologies of a triplet with leaves x, y, and z.

standard implementation. For the triplet and quartet distance measures, no such trivial modifications exist.

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

2.1.1 Problem Definition

For a given rooted unordered tree T where each leaf has a unique label, a *triplet* is defined by a set of three leaf labels x, y, and z and their induced topology in T. The four possible topologies are illustrated in Figure 2.1. The notation $xy|z$ is used to describe a triplet where the lowest common ancestor of x and y is at a lower depth than the lowest common ancestor of z with either x or y. Note that the triplet $xy|z$ is the same as the triplet $yx|z$ because T is considered to be unordered. Similarly, notation xyz is used to describe a triplet for which every pair of leaves has the same lowest common ancestor. This triplet can only appear if we allow nodes with degree three or larger in T. From here on, when using the word "tree" we imply a "rooted unordered tree".

For two given trees T_1 and T_2 that are built on n identical leaf labels, the *triplet distance* $D(T_1, T_2)$ is the number of triplets the leaves of which induce different topologies in T_1 and T_2. Let $S(T_1, T_2)$ be the number of *shared* triplets in the two trees, i.e., leaf triples with identical topologies in the two trees. We then have the relationship $D(T_1, T_2) + S(T_1, T_2) = \binom{n}{3}$.

Previous and new results for computing the triplet distance are shown in Table 2.1. Note that the papers [5, 12, 23, 45, 66] do not provide an analysis of the algorithms in the cache oblivious model, so here we provide an upper bound. From here on and unless otherwise stated, any asymptotic bound refers to time.

2.1.2 Related Work

The triplet distance was first suggested as a method of comparing the shapes of trees by Dobson in 1975 [25]. The first non-trivial algorithmic result dates back to 1996, when Critchlow *et al.* [23] proposed an $O(n^2)$ algorithm that however works only for binary trees. Bansal *et al.* [5] introduced an $O(n^2)$

Year	Reference	Time	IOs	Space	Non-Binary Trees
1996	Critchlow *et al.* [23]	$O(n^2)$	$O(n^2)$	$O(n^2)$	no
2011	Bansal *et al.* [5]	$O(n^2)$	$O(n^2)$	$O(n^2)$	yes
2013	Sand *et al.* [66]	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(n)$	no
2013	Brodal *et al.* [12]	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	yes
2015	Jansson & Rajaby [45]	$O(n \log^3 n)$	$O(n \log^3 n)$	$O(n \log n)$	yes
2017	new	$O(n \log n)$	$O(\frac{n}{B} \log_2 \frac{n}{M})$	$O(n)$	yes

Table 2.1: Previous and new results for computing the triplet distance between two trees that are built on the same leaf label set of size n.

algorithm that works for general (binary and non-binary) trees. Both of these algorithms use $O(n^2)$ space. Sand *et al.* [66] introduced a new $O(n^2)$ algorithm using only $O(n)$ space for the case of binary trees, that they showed how to optimize to reduce the time to $O(n \log^2 n)$. This algorithm was also implemented and shown to be the most efficient in practice. Soon after, Brodal *et al.* [12] managed to extend the $O(n \log^2 n)$ algorithm to work for general trees, and at the same time brought the time down to $O(n \log n)$ but now with the space increased to $O(n \log n)$. The space for binary trees was still $O(n)$. The algorithms from [66] and [12] were implemented and added to the library tqDist [68]. Interestingly, it was shown in [40] that for binary trees the $O(n \log^2 n)$ algorithm had a better practical performance than the $O(n \log n)$ algorithm. Jansson and Rajaby [45, 46] showed that an even slower theoretically algorithm requiring worst case $O(n \log^3 n)$ time and $O(n \log n)$ space could give the best practical performance, both for binary and non-binary trees. A survey of previous results until 2013 can be found in [67].

2.1.3 Contribution

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

2.1.4 Outline of the Article

In Section 2.2 we provide an overview of previous approaches. In Section 2.3 we describe the new algorithm for the case where T_1 and T_2 are binary trees. In Section 2.4 we extend the algorithm to also work for general trees. In Section 2.5 we provide some implementation details. Section 2.6 contains our experimental evaluation. Appendix A contains more experimental results. Finally, in Section 2.7 we provide our concluding remarks.

2.2 Previous Approaches

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

Critchlow *et al.* [23] exploit the depth of the leaves' ancestors to achieve the first improvement over the naive approach. Bansal *et al.* [5] exploit the shared leaves between subtrees and reduce the problem to computing the intersection size (number of shared leaves) of all pairs of subtrees, one from T_1 and one from T_2, which can be solved with dynamic programming.

2.2.1 The $O(n^2)$ Algorithm for Binary Trees in [66]

The algorithm for binary trees in [66] is the basis for all subsequent improvements [12, 45, 66], including ours as well, so we will describe it in more detail here. The dependency that was exploited is the same as in [5] but the procedure for counting the shared triplets is completely different. More specifically, each triplet in T_1 and T_2, defined by three leaf labels i, j, and k, is implicitly *anchored* in the lowest common ancestor of i, j, and k. For two nodes u in T_1 and v in T_2, let $s(u)$ and $s(v)$ be the set of triplets that are anchored in u and v respectively. For the number of shared triplets $S(T_1, T_2)$ we then have:

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

For the algorithm to be $O(n^2)$ the value $|s(u) \cap s(v)|$ must be computed in $O(1)$ time. This is achieved by a leaf colouring procedure as follows: Fix an internal node u in T_1 and color the leaves in the left subtree of u *red*, the leaves in the right subtree of u *blue*, let every other leaf have no color and then transfer this coloring to the leaves in T_2, i.e., identically labeled leaves get the same color. The triplets anchored at u are exactly the triplets $xy|z$ where x, y

are blue and z is red, or x, y are red and z is blue. To compute $|s(u) \cap s(v)|$ we do as follows: let l and r be the left and right children of v, and let $w_{\texttt{red}}$ and $w_{\texttt{blue}}$ be the number of red and blue leaves in a subtree rooted at a node w in T_2. We then have:

$$|s(u) \cap s(v)| = \binom{l_{\texttt{red}}}{2} r_{\texttt{blue}} + \binom{l_{\texttt{blue}}}{2} r_{\texttt{red}} + \binom{r_{\texttt{red}}}{2} l_{\texttt{blue}} + \binom{r_{\texttt{blue}}}{2} l_{\texttt{red}} \quad (2.1)$$

2.2.2 Subquadratic Algorithms

To reduce the time, Sand *et al.* [66] applied the *smaller half trick*, which specifies a depth first order to visit the nodes u of T_1, so that each leaf in T_1 changes color at most $\mathrm{O}(\log n)$ times. To count shared triplets efficiently without scanning T_2 completely for each node u in T_1, the tree T_2 is stored in a data structure denoted a *hierarchical decomposition tree* (*HDT*). This HDT of T_2 maintains for the current visited node u in T_1, according to (2.1) the sum $\sum_{v \in T_2} |s(u) \cap s(v)|$, so that each leaf color change in T_1 can be updated efficiently in T_2. In [66] the HDT is a binary tree of height $\mathrm{O}(\log n)$ and every update can be done by a leaf to root path traversal in the HDT, which in total gives $\mathrm{O}(n \log^2 n)$ time. In [12] the HDT is generalized to also handle non-binary trees, each query operates the same, and now due to a contraction scheme of the HDT the total time is reduced to $\mathrm{O}(n \log n)$. Finally, in [45] as an HDT the so called *heavy-light tree decomposition* is used. Note that the only difference between all $\mathrm{O}(n \operatorname{polylog} n)$ results that are available right now is the type of HDT used.

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

2.3 The New Algorithm for Binary Trees

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

2.3.1 Overview

We use the $\mathrm{O}(n^2)$ algorithm from Section 2.2.1 as a basis. The main difference between this algorithm and our new algorithm is in the order that we visit the nodes of T_1, and how we process T_2 when we count. We propose a new order of visiting the nodes of T_1, which is found by applying a hierarchical decomposition on T_1. Every component in this decomposition corresponds to a connected part of T_1 and a contracted version of T_2. In simple terms, if Λ is

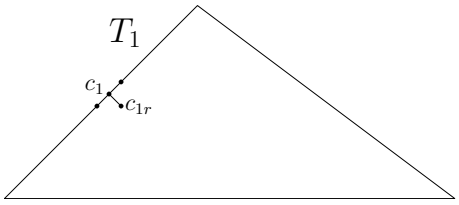
the set of leaves in a component of T_1, the contracted version of T_2 is a binary tree on Λ that preserves the topologies induced by Λ in T_2 and has size $O(|\Lambda|)$. To count shared triplets, every component of T_1 has a representative node u that we use to scan the corresponding contracted version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

2.3.2 Modified Centroid Decomposition

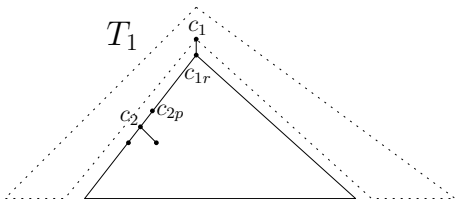
For a given binary tree T let $|T|$ denote the number of nodes in T (internal nodes and leaves). For a node u in T let l and r be the left and right children of u, and p the parent of u. Removing u from T partitions T into three (possibly empty) *connected components* T_l, T_r, and T_p containing l,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

Lemma 1. *For any given binary tree T with n leaves, there exists an algorithm that builds $CD(T)$ using $O(n \log n)$ time and $O(n)$ space in the RAM model.*

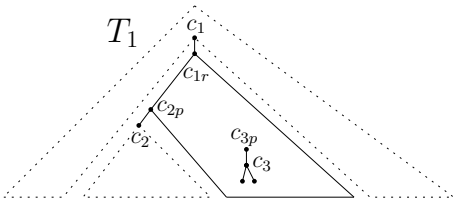
A component in a centroid decomposition $CD(T)$, might have several edges crossing its boundaries (connecting nodes inside and outside the component). An example of creating a component that has two edges from below can be found in Figure 2.2. It is trivial to see that by following the same pattern of generating components as depicted in Figure 2.2(d), $CD(T)$ can have a component with an arbitrary number of edges from below. The below *modified centroid decomposition*, denoted $MCD(T)$, generates components with at most two edges crossing the boundary, one going towards the root and one down to exactly one subtree.



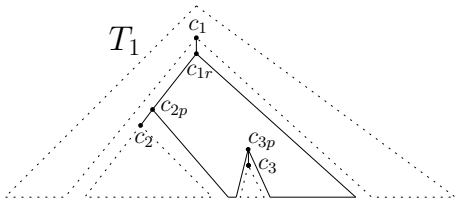
(a) Picking the first centroid c_1 of T_1.



(b) Recursing to the component defined by c_{1r} and picking the centroid c_2 of that component.



(c) Recursing to the component defined by c_{2p} and picking the centroid c_3 of that component.



(d) , Recursing to the component defined by c_{3p}.

Figure 2.2: Generating a component in $CD(T_1)$ that has two edges from below. The black polygon is the component.

An $MCD(T)$ is built as follows: The first component is defined by T, just like in $CD(T)$. To find recursively the rest of the components, if a component C has no edge from below, we select the centroid c of C as a splitting node, just like when building $CD(T)$. Otherwise, let (x, y) be the edge that crosses the boundary from below, where x is in C, y is a child of x and y is not in C, and let c be the centroid of C (possibly $x = c$). As a splitting node choose the lowest common ancestor of x and c (possibly x or c). By induction every component has at most one edge from below and one edge from above. A useful property of $MCD(T)$ is captured by the following lemma:

Lemma 2. *For any given binary tree T, we have $h(MCD(T)) \leq 2+2\log_2|T|$, where $h(MCD(T))$ denotes the height of $MCD(T)$.*

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

i.e., we have $|C_l| + |C_p| \leq \frac{|C|}{2}$ and $|C_r| \geq \frac{|C|}{2}$. From the first inequality, we have that $|C_l| \leq \frac{|C|}{2}$ and $|C_p| \leq \frac{|C|}{2}$. Notice that C_r is going to be a component corresponding to a complete subtree of T, so it will have no edges from below. This means that in the next recursion level when working with C_r the actual centroid of C_r will be chosen as a splitting node, thus in the following recursion level the three components produced from C_r will be such that their sizes are at most half the size of C. From the analysis given so far, it becomes clear that when we have a component of size $|C|$ with one edge from below, then we will need at most 2 levels in $MCD(T)$ before producing components all of which will have a guaranteed size of at most $\frac{|C|}{2}$. Hence, the statement follows. □

Since every level of $MCD(T)$ can be constructed in $O(|T|)$ time and we have that $|T| = 2n - 1$, we obtain the following:

Theorem 1. *For any given binary tree T with n leaves, there exists an algorithm that constructs $MCD(T)$ using $O(n \log n)$ time and $O(n)$ space in the RAM model.*

2.3.3 The Main Algorithm

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

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

In the counting step, we visit the nodes of T_1, given by the depth first traversal of the ternary tree $MCD(T_1)$, where the children of every node u in $MCD(T_1)$ are visited from left to right. For every such node u we compute $\sum_{v \in T_2} |s(u) \cap s(v)|$. We achieve this by processing T_2 in two phases, the *contraction* phase and the *counting* phase.

Contraction Phase of T_2. Let $L(T_2)$ denote the set of leaves in T_2 and $\Lambda \subseteq L(T_2)$. In the contraction phase, T_2 is compressed into a binary tree of size $O(|\Lambda|)$ whose leaf set is Λ. The contraction is done in a way so that all the topologies induced by Λ in T_2 are preserved in the compressed binary tree. This is achieved by the following three sequential steps:

– Prune all leaves of T_2 that are not in Λ,

- Repeatedly prune all internal nodes of T_2 with no children, and
- Repeatedly contract unary internal nodes, i.e., nodes having exactly one child.

Let u be a node of $MCD(T_1)$ and C_u the corresponding component of T_1. For every such node u we have a contracted version of T_2, from now on referred to as $T_2(u)$, where $L(T_2(u)) = L(C_u)$. The goal is to augment $T_2(u)$ with counters (see counting phase below), so that we can find $\sum_{v \in T_2} |s(u) \cap s(v)|$ by scanning $T_2(u)$ instead of T_2. One can imagine $MCD(T_1)$ as being a tree where each node u is augmented with $T_2(u)$. To generate all contractions of T_2 for level i of $MCD(T_1)$, which correspond to a set of disjoint connected components in T_1, we can reuse the contractions of T_2 at level $i-1$ in $MCD(T_1)$. This means that we can generate the contractions of level i in $O(n)$ time, thus we can generate all contractions of T_2 in $O(n \log n)$ time. Note that by explicitly storing all contractions, we will also need to use $O(n \log n)$ space. For our problem, because we traverse $MCD(T_1)$ in a depth first manner, we only need to store the contractions corresponding to the stack of nodes of $MCD(T_1)$ that we have to remember during the traversal of $MCD(T_1)$. Since the components at every second level of $MCD(T_1)$ have at most half the size of the components two levels above, Lemma 3 states that the size of this stack is always $O(n)$.

Lemma 3. *Let T_1 and T_2 be two binary trees with n leaves and $u_1, u_2, \ldots, u_k$ a root to leaf path of $MCD(T_1)$. For the sizes of the corresponding contracted versions $T_2(u_1), T_2(u_2), \ldots, T_2(u_k)$ we have that $\sum_{i=1}^{k} |T_2(u_i)| = O(n)$.*

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

Counting Phase of T_2. In the counting phase, we find the value of $\sum_{v \in T_2} |s(u) \cap s(v)|$ by scanning $T_2(u)$ instead of T_2. This makes the total time of the algorithm in the RAM model $O(n \log n)$, with the space being $O(n)$ because of Lemma 3. We consider the following two cases:

- *C_u has no edges from below.*

  In this case C_u corresponds to a complete subtree of T_1. We act exactly like in the $O(n^2)$ algorithm (Section 2.2) but now instead of scanning T_2 we scan $T_2(u)$.
- *C_u has one edge from below.*

  In this case C_u does not correspond to a complete subtree of T_1, since the edge from below C_u, points to a subtree X_u, that is located outside of C_u (see Figure 2.3). Note that because in the preprocessing step T_1 was made to be left-heavy, X_u is always rooted at a node on the leftmost path from u. The leaves in X_u are important because they can be used to form triplets that are anchored in u. Acting in the exact

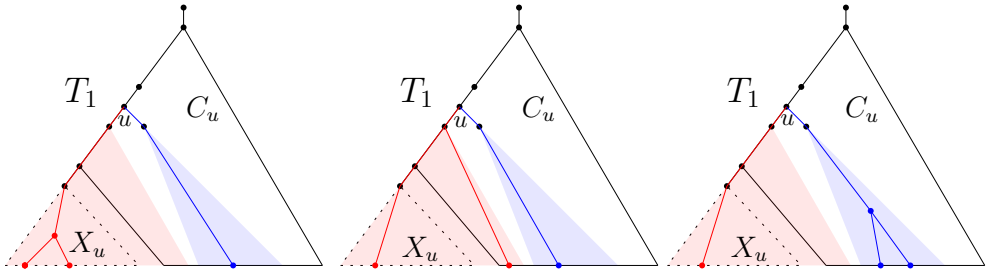


Figure 2.3: $MCD(T_1)$: Triplets (red and blue) that can be anchored in u with the leaves not being in the component C_u.

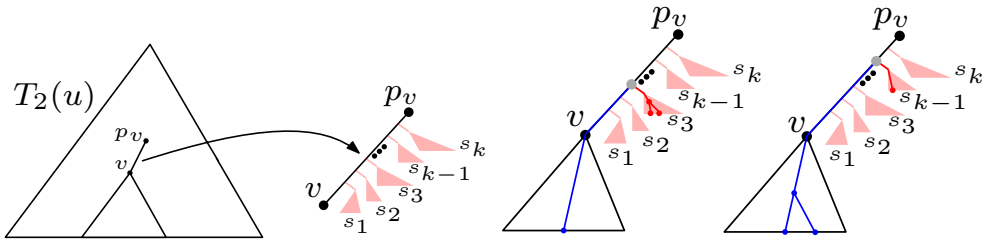


Figure 2.4: Contracted subtrees on edges in $T_2(u)$ and shared triplets rooted at contracted nodes.

same manner as in the previous case is not sufficient because we need to count these triplets as well.

To address this problem, every edge (p_v, v) in $T_2(u)$ between a node v and its parent p_v, is augmented with some counters about the leaves from X_u that were contracted away in T_2. If v is the root of $T_2(u)$, we add an extra edge to store this information. For every such edge (p_v, v), let $s_1, s_2, \ldots, s_k$ be the contracted subtrees rooted at the edge (see Figure 2.4). Every such subtree contains either leaves with no color or leaves from X_u that have the color red (the color cannot be blue because T_1 was made to be left-heavy). For every node v in $T_2(u)$ the counters that we have are the following:

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

The number of shared triplets that are anchored in a non-contracted node v of $T_2(v)$ can be found like in the $O(n^2)$ algorithm using the counters $v_{\texttt{red}}$ and $v_{\texttt{blue}}$ in (2.1). As for the number of shared triplets

that are anchored in a contracted node on edge (p_v, v), this value is exactly $\binom{v.\texttt{blue}}{2} \cdot v_{ts} + v_{\texttt{blue}} \cdot v_{ps}$.

2.3.4 Scaling to External Memory

We now describe how to make the algorithm scale to external memory. The tree T_1 is stored in an array of size $2n - 1$ by following a preorder layout, i.e., if a node w of T_1 is stored in position p, the left child of w is stored in position $p + 1$ and if x is the size of the left subtree of w, the right child of w is stored in position $p + x + 1$. The components of T_1 are connected parts of T_1, so they can be identified in T_1 without having to make a unique copy for each one of them. For T_2 and its contractions, we use the proof of Lemma 3 to initialize a large enough array that can fit T_2 and every contraction of T_2 that we need to remember while traversing $MCD(T_1)$. This array is used as a stack that we use to push and pop the contractions of T_2. The tree T_2 and its contractions are stored in memory following a post order layout, i.e., if a node w is stored in position p and y is the size of the right subtree of w, the left child of w is stored in position $p - y - 1$ and the right child of w is stored in position $p - 1$.

In the preprocessing step, T_1 can be made left-heavy with two depth first traversals. The first traversal computes for every node u in T_1 the size of the subtree rooted at u. The second traversal starts from the root of T_1, recursively visits the children by first visiting a largest child, and prints all nodes visited along the way to an output array. This output array will at the end of the traversal contain the left-heavy version of T_1 in a preorder layout. From the following Lemma 4 we have that both the first and second depth first traversal of T_1 require $O(n/B)$ I/Os in the cache oblivious model, i.e., making T_1 left-heavy requires $O(n/B)$ I/Os in the cache oblivious model.

In Lemma 4 we consider the I/Os required to apply a depth first traversal on a binary tree T that is stored in memory following a local layout, i.e., the nodes of every subtree of T are stored consecutively in memory and every node has $O(1)$ occurrences in memory. From here on, when we refer to an edge (u, v), we imply that u is the parent of v in T. During a depth first traversal of T, an edge (u, v) is either *processed* to discover v or to backtrack from v to u. In any case, w.l.o.g. we assume that when an edge is processed, both u and v are visited, i.e., both u and v are accessed in memory.

Lemma 4. *Let T be a binary tree with n leaves that is stored in an array following a local layout, i.e., the nodes of every subtree of T are stored consecutively in memory and every node has $O(1)$ occurrences in memory. Any depth first traversal that starts from the root of T, and in which for every internal node u in T the children of u are discovered in any order, requires $O(n/B)$ I/Os in the cache oblivious model.*

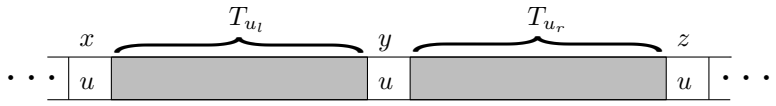


Figure 2.5: Position of a node u in memory with respect to the two children subtrees of u.

Proof. For a node u in T, let T_u denote the set of nodes in the subtree defined by u. From here on, T_u will be referred to as a *subtree of* T. Let u_l and u_r be the two children of u. In Figure 2.5 we illustrate the three possibilities for the position of u in memory with respect to T_{u_l} and T_{u_r}. W.l.o.g. and to simplify the presentation of the proof, in our analysis we assume that u is stored in all these three possible positions, denoted x, y, and z. This assumption is w.l.o.g. because in any local layout one or more of these positions is used, thus the number of I/Os is upper bounded by the number of I/Os incurred if we follow our assumption. This placement of u in memory implies that when u is visited in a depth first traversal of T, all the three copies of u are accessed in memory. Note that according to the definition of a local layout, T_{u_l} and T_{u_r} can be interchanged in Figure 2.5. In the following, the aim is to bound the number of I/Os implied.

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

S_1: Every B-light node
S_2: Every B-heavy node with only B-light children
S_3: Every B-heavy node with two B-heavy children
S_4: Every B-heavy node with one B-heavy child and one B-light child.

For a B-light node u in T, let w be the first B-heavy node we reach in the path from u to the root of T. An I/O incurred by visiting the node u in T is charged to w. This node w can be either in S_2 or S_4. Let w' be the child of w such that $T_{w'}$ contains u. Since $3|T_{w'}| \leq B-2$, at most 1 I/O is sufficient to visit all nodes in $T_{w'}$. We say that $T_{w'}$ is a subtree that is B-light. In Figure 2.6(a) we have an example of a tree, where the gray subtrees denote B-light subtrees.

We now argue that $|S_2| = \mathrm{O}(n/B)$ and $|S_3| = \mathrm{O}(n/B)$. Let T' be the binary tree created by pruning every B-light node from T and their incident edges, and subsequently contracting nodes with in-degree of 1 and out-degree of 1. An example for T and the corresponding tree T' can be found in Figures 2.6(a) and 2.6(b). Let $l_1, l_2, \ldots, l_k$ be the leaves of T' and $T_{l_1}, \ldots, T_{l_k}$ the corresponding subtrees in T. Since all these subtrees are disjoint and for every $1 \leq i \leq k$ we have $|T_{l_i}| > \frac{B-2}{3}$, for the total number of leaves x in T' we have $x \leq 3|T|/(B-2)$. Hence, we have $|S_2| = x = \mathrm{O}(n/B)$. By construction T' is a binary tree, thus we have that $|T'| \leq 2x \leq 6|T|/(B-2) = \mathrm{O}(n/B)$. Since

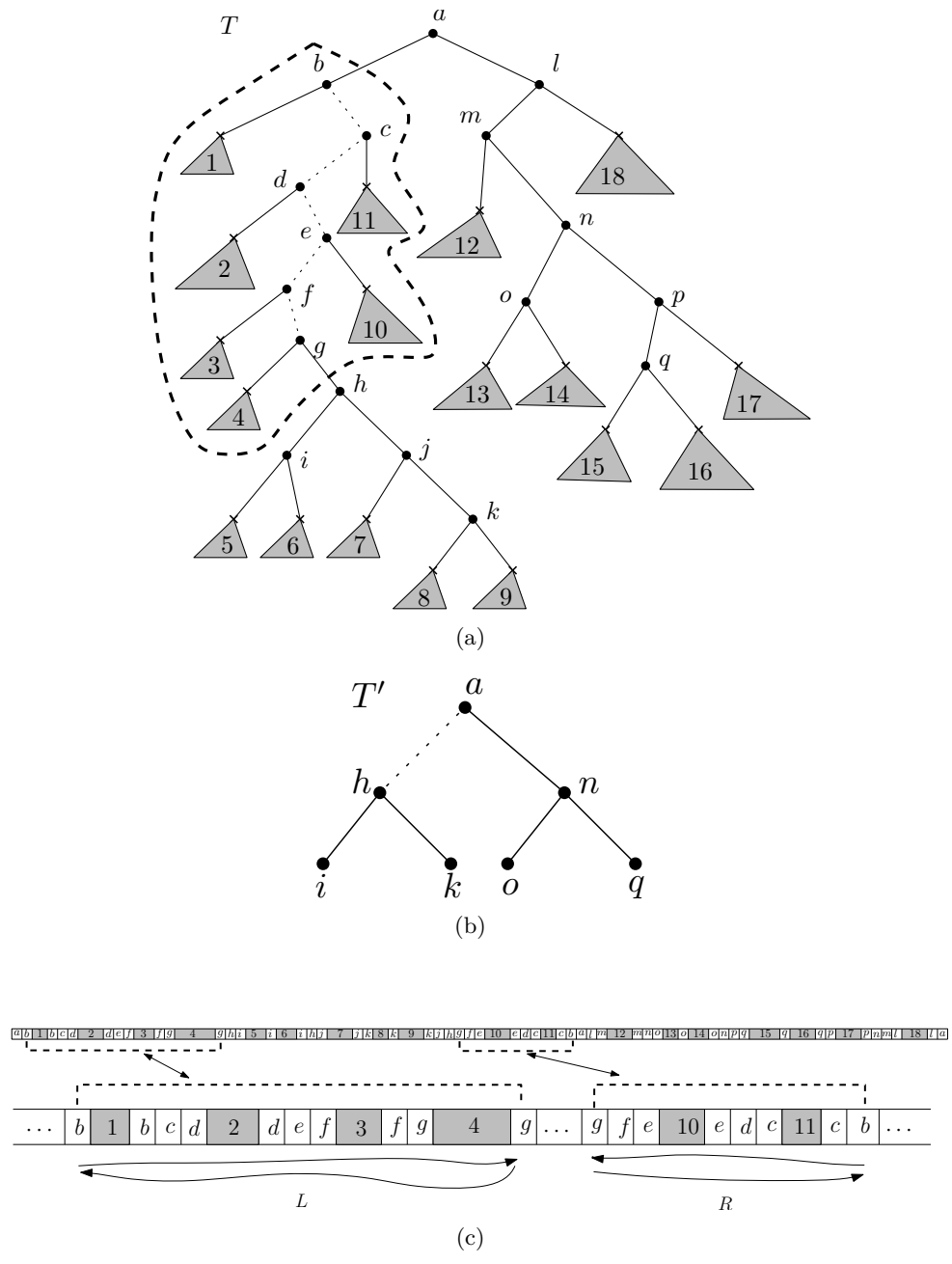


Figure 2.6: (a) A tree T. The gray subtrees are B-light subtrees and every node not in a B-light subtree is a B-heavy node. (b) The corresponding tree T' according to the proof of Lemma 4. (c) How T is stored in memory, the two segments of memory (in dashed lines) that correspond to the edge (a, h) in T' and how the nodes in $P_{(a,h)}$ are visited (defined by the one directional lines) during a depth first traversal of T.

the nodes in S_3 correspond to internal nodes in T', we have $|S_3| = O(n/B)$.

We now argue that the total number of I/Os incurred by the nodes in S_4 is $O(n/B)$, thus proving the statement. Let (u, v) be an edge in T'. This edge corresponds to a unique path, denoted $P_{(u,v)}$ in T that contains every B-heavy node, except u and v, that is in the path from u to v. For example the edge (a, h) in Figure 2.6(b) corresponds to $P_{(a,h)} = b \to c \to d \to e \to f \to g$. Let $C_{(u,v)}$ contain all B-light and B-heavy nodes, except u and v, rooted at the path from u to v in T. By the local layout followed to store T in memory, the nodes in $C_{(u,v)}$ are stored in two segments of memory (e.g., see Figure 2.6(c)). Let L be the left segment and R the right segment. During a depth first traversal of T, visiting all nodes in $P_{(u,v)}$ corresponds to visiting L from left to right and then from right to left, and visiting R from right to left and then from left to right. Since each of the B-light subtrees in L and R use at most $B - 2$ positions in memory, by accessing all three copies of a node w in $P_{(u,v)}$ every time w is visited in a depth first traversal of T, we guarantee that the corresponding B-light subtree rooted at w is in cache, i.e., it can be accessed in memory for free. Hence, the total number of I/Os that are sufficient to pay for traversing all nodes in $C_{(u,v)}$ is $4+\lceil 3|C_{(u,v)}|/B\rceil$, where the $+4$ comes from the 4 I/Os we need to pay (in the worst case) to visit the first and last node of L and R. In total, the total number of I/Os we need to spend for all paths of T that correspond to edges of T' is $\sum_{(u,v)\in T'}(4 + \lceil 3|C_{(u,v)}|/B\rceil) = O(n/B)$. Together with the fact that for every node of T that corresponds to a node of T' we only spend $O(1)$ I/Os and there are $O(n/B)$ such nodes, the statement follows. □

Changing the labels of T_1 can be done in $O(\frac{n}{B}\log_2\frac{n}{M})$ I/Os with a cache oblivious sorting routine, e.g., with merge sort. Overall, the preprocessing step requires $O(\frac{n}{B}\log_2\frac{n}{M})$ I/Os.

When building $MCD(T_1)$, by scanning the leftmost path that starts from the root of a component C_u, we can find the splitting node of C_u in $1 + \lceil|C_u|/B\rceil$ I/Os. In $T_2(u)$ we spend $1 + \Theta(\lceil|T_u|/B\rceil)$ I/Os for the contraction and counting phase. Since $|T_2(u)| = \Theta(|C_u|)$, overall for a given $(C_u, T_2(u))$ pair the algorithm requires $2 + \Theta(\lceil|C_u|/B\rceil)$ I/Os. However, after $O(\log_2\frac{n}{M})$ levels in $MCD(T_1)$, any $(C_u, T_2(u))$ pair will fit in a cache of size M. All such pairs together incur $O(n/B)$ I/Os. By using a stack to store the contractions of T_2, the remaining pairs incur $O(\frac{n}{B}\log_2\frac{n}{M})$ I/Os. Overall, the algorithm requires $O(\frac{n}{B}\log_2\frac{n}{M})$ I/Os in the cache oblivious model.

2.4 The New Algorithm for General Trees

Unlike a binary tree, a general tree can have internal nodes with an arbitrary number of children. By anchoring the triplets of T_1 and T_2 in edges instead of nodes, we show that with only four colors we can count all the shared triplets

between the two trees. We start by describing a new $O(n^2)$ algorithm for general trees. We then show how we can use the same ideas presented in the previous section to extend the $O(n^2)$ algorithm and reduce the time to $O(n \log n)$.

2.4.1 Quadratic Algorithm

For a given tree T, let t be a triplet with leaves i, j, and k that is either a resolved triplet $ij|k$ or an unresolved triplet ijk, where i is to the left of j and for the triplet ijk, k is also to the right of j. Let w be the lowest common ancestor of i and j and (w, c) the edge from w to the child c whose subtree contains j. We anchor t in edge (w, c). Let $s'(w, c)$ be the set containing all triplets anchored in edge (w, c). For the number of shared triplets $S(T_1, T_2)$ we have:

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

For the efficient computation of $S(T_1, T_2)$ we use the following coloring procedure: Fix a node u in T_1 and a child c. Color the leaves of every child subtree of u to the left of c red, the leaves of the child subtree defined by c blue, the leaves of every child subtree to the right of c green and give the color black to every other leaf of T_1. We then transfer this coloring to the leaves of T_2. For the resolved triplet $ij|k$, i corresponds to the red color, j corresponds to the blue color and k corresponds to the black color. For the unresolved triplet ijk, i corresponds to the red color, j corresponds to the blue color and k corresponds to the green color.

Suppose that the node v in T_2 has k children. We are going to compute all shared triplets that are anchored in the k children edges of v in $O(k)$ time. This will give an $O(n^2)$ total running time, because for every edge in T_1 we spend $O(n)$ time in T_2 and there are $O(n)$ edges in T_1. In v we have the following counters:

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

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

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

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

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

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

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

2.4.2 Subquadratic Algorithm

Similarly to the case of binary trees in Section 2.3, there is a preprocessing step and a counting step. The counting step is divided into two phases, the contraction and counting phase of T_2.

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

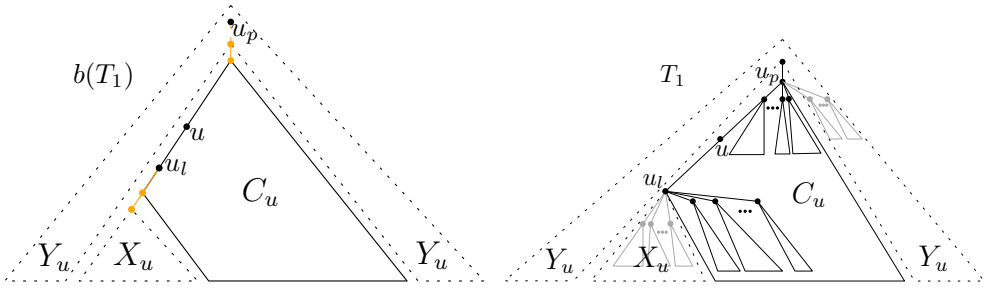


Figure 2.7: How a component in $b(T_1)$ translates to a component in T_1.

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

For the number of shared triplets we then have:

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

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

The reason behind transforming T_1 into the binary tree $b(T_1)$, is because now we can use exactly the same core ideas described in Section 2.3. The tree $b(T_1)$ is a binary tree, so we apply the same preprocessing step, except we do not need to make it left-heavy because by construction it already is. However, we change the labels of the leaves in $b(T_1)$ and T_2, so that the leaves in $b(T_1)$ are numbered 1 to n from left to right. The order in which we visit the nodes of $b(T_1)$ is determined by a depth first traversal of $MCD(b(T_1))$, where the children of every node u in $MCD(b(T_1))$ are visited from left to right.

In Figure 2.7 we see that a component C_u of $b(T_1)$ structurally looks like a component of T_1 in the binary algorithm of Section 2.3. However, the edges crossing the boundary can now be orange edges as well, which in T_1 translates to more than one consecutive subtrees.

Like in the case of binary input trees, while traversing $MCD(b(T_1))$ we process T_2 in two phases, the contraction phase and the counting phase. The

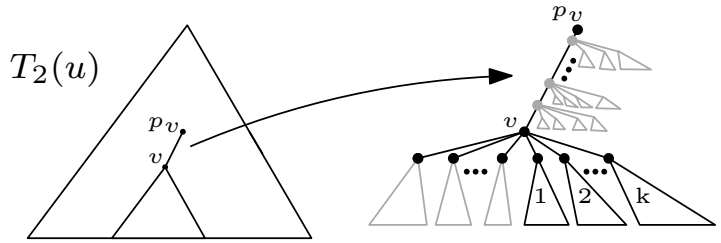


Figure 2.8: $T_2(u)$: Contracted children subtrees rooted at node v and contracted subtrees rooted at contracted nodes (gray color) on the edge (p_v, v).

only difference after this point between the algorithm for binary trees and the algorithm for general trees, is in the counters that we have to maintain in the contracted versions of T_2. Otherwise, the same analysis from Section 2.3 holds.

Contraction Phase of T_2. The contraction of T_2 with respect to a set of leaves $\Lambda \subseteq L(T_2)$, happens in the exact same way as described in Section 2.3, i.e., we start by pruning all leaves of T_2 that are not in Λ, then we prune all internal nodes of T_2 with no children, and finally, we contract the nodes that have exactly one child.

Let u be a node of $MCD(b(T_1))$ and C_u the corresponding component of $b(T_1)$. For every such node u we have a contracted version of T_2, denoted $T_2(u)$, where $L(T_2(u)) = L(C_u)$. Like in the binary algorithm of Section 2.3, the goal is to augment $T_2(u)$ with counters, so that we can find $\sum_{(v,c')\in T_2} |s''(u,c) \cap s'(v,c')|$ by scanning $T_2(u)$ instead of T_2.

Because of the location where the triplets are anchored, every leaf that was contracted when constructing $T_2(u)$ must have a color and be stored in some way. The color of each such leaf depends on the type of the corresponding component that we have in $b(T_1)$ and the splitting node that is used for that component. For example, in Figure 2.7 the contracted leaves from X_u will have the red color because $b(T_1)$ is left-heavy. The contracted leaves from the children subtrees of u_p in T_1 can either have the color green or black. If u in $b(T_1)$ happens to be orange and part of the orange binary tree that u_p is the root of, then the color must be green, otherwise black. Finally, every leaf that is not in the subtree defined by u_p, and thus is in Y_u, must have the color black. The way we store this information is described in the counting phase below.

Counting Phase of T_2. In Figure 2.8 we illustrate how a node v in $T_2(u)$ can look like. The contracted subtrees are illustrated with the dark gray color. Every such subtree contains some number of red, green, and black leaves. The counters that we maintain should be so that if v has k children in $T_2(u)$, then we can count all shared triplets that are anchored in every child edge (including those of the contracted children subtrees) of v in $O(k)$ time. At the same time, in $O(1)$ time we should be able to count all shared triplets that are anchored

in every child edge of every contracted node that lies on the edge (p_v, v). Then, the time required by the counting phase becomes $O(|T_2(u)|)$, giving the same time bounds as in the binary algorithm of Section 2.3. In v we have the following counters:

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

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

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

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

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

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

Every variable is updated in O(1) time in exactly the same manner like in the $O(n^2)$ algorithm of Section 2.4.1. The main difference is in the values of the variables before we begin scanning the children edges of v. Every variable is initialized as follows:

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

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

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

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

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

2.4.3 Scaling to External Memory

The analysis is the same as in Section 2.3, except for minor details. The proof of Lemma 3 can be trivially modified to apply to general trees as well. Finally, Lemma 4 is generalized to non-binary trees in the following Lemma 5. In Lemma 5, we consider the I/Os required to apply a depth first traversal on a non-binary tree T that is stored in memory following a local layout, i.e., the nodes of every subtree of T are stored consecutively in memory and every node has O(1) occurrences in memory. Similarly to the assumptions we made for Lemma 4, w.l.o.g. we assume that when an edge (u, v) of T is processed in a depth first traversal of T, both u and v are visited, i.e., both u and v are accessed in memory.

Lemma 5. *Let T be a non-binary tree with n leaves that is stored in an array following a local layout, i.e., the nodes of every subtree of T are stored consecutively in memory and every node has* O(1) *occurrences in memory. Any depth first traversal that starts from the root of T and in which for every internal node u in T, after the discovery of the first child of u the remaining children are discovered in order that they appear in memory from left to right, requires* $O(n/B)$ *I/Os in the cache oblivious model.*

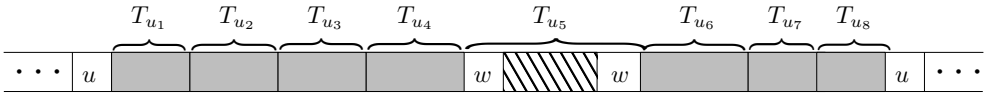


Figure 2.9: Position of a node u in memory with respect to the 8 subtrees defined by the children of u, with T_{u_5} being a largest subtree.

Proof. This proof can be thought of as an extension of the proof of Lemma 4. Following the proof of Lemma 4, for a node u in T, let T_u denote the set of nodes in the subtree defined by u. For $i \geq 2$, let $u_1, \ldots, u_i$ be the children of u and let $T_{u_1}, \ldots, T_{u_i}$ be the corresponding subtrees. We assume that these subtrees are ordered from left to right in order that they appear in memory. In the proof of Lemma 4, we implicitly assumed that the positions of the two children of u are stored together with u in memory. For general trees, together with u we need to store a list of arbitrary size $i \geq 2$ containing the positions in memory of every child of u. To avoid complicating the presentation of the proof, we assume that we can find the position in memory of every child of u without this list, i.e., this list is not stored together with u, thus finding the position of any child of u incurs no I/Os. An easy way to support this is to store in every node u in T, one pointer to the first child to be discovered and one pointer to the sibling appearing next in memory. For every node u in T, we allow a constant number of occurrences in memory. For any given placement of the copies of u in memory, we add two copies of u before the first child subtree and after the last child subtree. W.l.o.g. we assume that u

is only stored before the first child subtree and after the last child subtree (see Figure 2.9 for an example).

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

- S_1: Every B-light node
- S_2: Every B-heavy node with only B-light children
- S_3: Every B-heavy node with at least two B-heavy children and an arbitrary number of B-light children
- S_4: Every B-heavy node with exactly one B-heavy child and at least 1 B-light children.

For a B-light node u in T, let w be the first B-heavy node we reach in the path from u to the root of T. An I/O incurred by visiting the node u in T is charged to w. This node w can be either in S_2, S_3 or S_4. Let w' be the child of w such that $T_{w'}$ contains u. Since $2|T_{w'}| \leq B-2$, at most 1 I/O is sufficient to visit all nodes in $T_{w'}$. We say that $T_{w'}$ is a subtree that is B-light. In Figure 2.10(a) we have an example of a tree, where the gray subtrees denote B-light subtrees.

Similarly to the proof of Lemma 4, we have that $|S_2| = \mathrm{O}(n/B)$ and $|S_3| = \mathrm{O}(n/B)$. Since T is non-binary, we have to argue that the number of I/Os spent traversing the B-light subtrees that are rooted at every node in S_2 and S_3 is $\mathrm{O}(n/B)$. For a node u in T, let G_u be the size of all gray subtrees rooted at u. For every node u in S_2 we spend at most 1 I/O to traverse the first chosen child subtree and $1+|G_u|/B$ I/Os to traverse the remaining subtrees, thus $2+|G_u|/B$ I/Os in total. Since $|S_2| = \mathrm{O}(n/B)$ and the gray subtrees in T are disjoint, i.e., $\sum_{u \in T} |G_u| = \mathrm{O}(n)$, we spend $\mathrm{O}(n/B)$ I/Os traversing the B-light subtrees rooted at every node in S_2. For every node u in S_3, let $d'(u)$ denote the number of B-heavy children of u. For this node u, we spend at most 1 I/O to traverse the first chosen child subtree that could be B-light and $1+d'(u)+|G_u|/B$ I/Os to traverse the remaining gray subtrees rooted at u. Since $|S_3| = \mathrm{O}(n/B)$, we have $\sum_{u \in T'} d'(u) = \mathrm{O}(n/B)$. Together with the fact that $\sum_{u \in T} |G_u| = \mathrm{O}(n)$, we spend $\mathrm{O}(n/B)$ I/Os traversing the B-light subtrees rooted at every node in S_3.

We now argue that the total number of I/Os incurred by the nodes in S_4 is $\mathrm{O}(n/B)$, thus proving the statement. Let T' be defined as in the proof of Lemma 4, as well as $P_{(u,v)}$ and $C_{(u,v)}$ for an edge (u,v) in T'. By the local layout followed to store T in memory, the nodes in $C_{(u,v)}$ are stored in two segments of memory (e.g., see Figure 2.10(c)). Let w be a node in $P_{(u,v)}$ and G_w be the total size of the gray subtrees rooted at w. We say that w is G-light if $2G_w \leq B-2$, otherwise G-heavy. There can be $\mathrm{O}(n/B)$ G-heavy nodes in T, thus by the same argument as in the previous paragraph, scanning the gray subtrees for all G-heavy nodes together incurs $\mathrm{O}(n/B)$ I/Os. For the G-light nodes we follow a similar argument as in the proof of lemma 4. Let L be the left chunk and R the right chunk and w.l.o.g assume that every

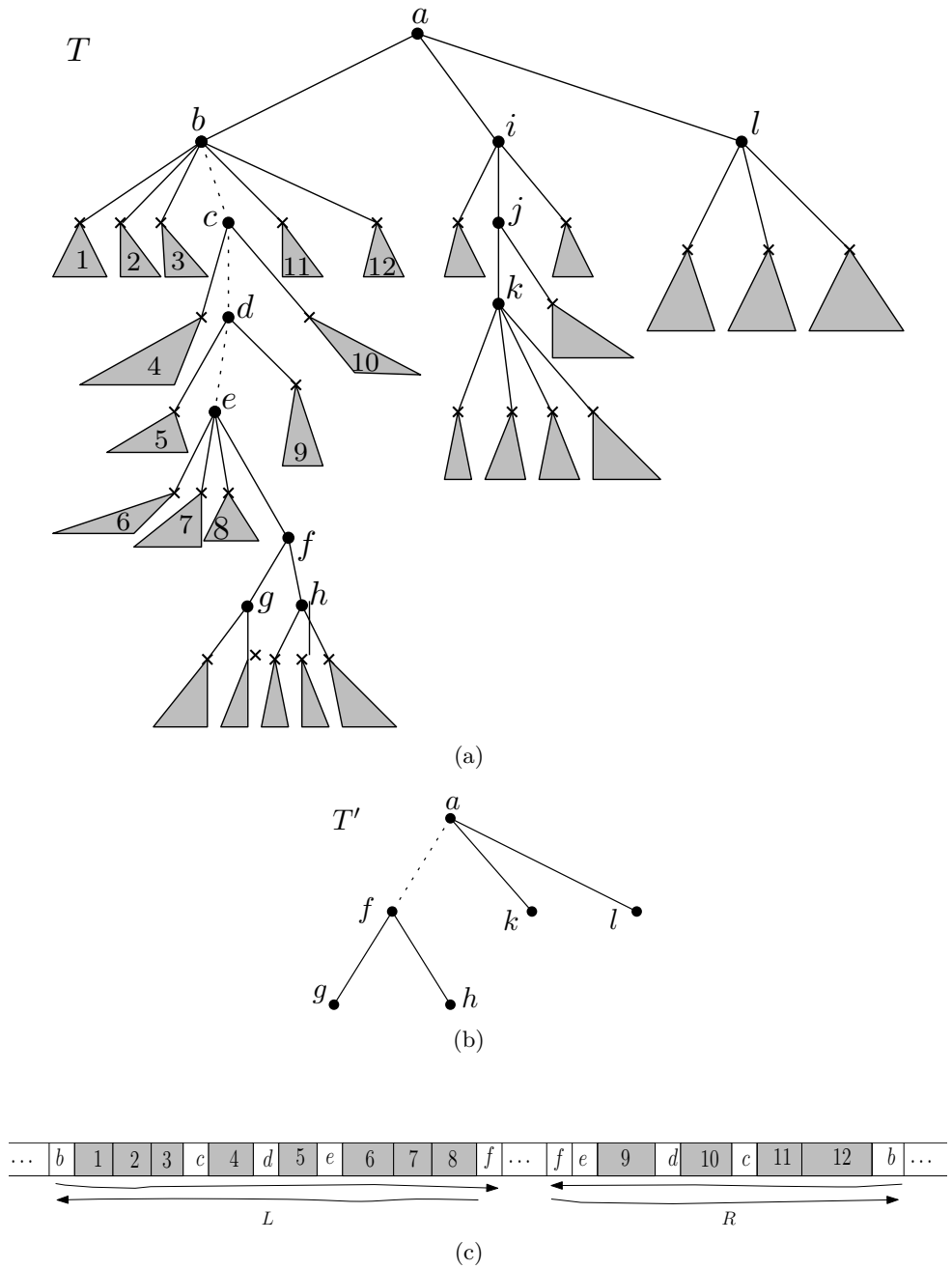


Figure 2.10: (a) A general tree T. The gray subtrees are B-light subtrees and every node not in a B-light subtree is a B-heavy node. (b) The corresponding tree T' according to the proof of Lemma 5. (c) How T is stored in memory and the two segments of memory that correspond to the edge (a, f) in T'.

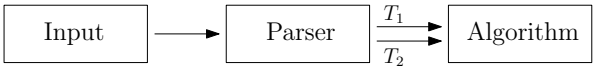


Figure 2.11: Implementation overview.

node in $P_{(u,v)}$ is G-light. During a depth first traversal of T, visiting all nodes in $P_{(u,v)}$ corresponds to visiting L from left to right and then from right to left, and visiting R from right to left and then from left to right. Let c be the child of w that is B-heavy. Since for every node w in $P_{(u,v)}$ we have $2G_w \leq B-2$, by accessing all two copies of w and c when c is visited in a depth first traversal of T, we guarantee that all the gray subtrees rooted at w are in cache i.e., they can be accessed in memory for free. Hence, $O(n/B)$ I/Os are sufficient to pay to traverse the gray subtrees of all G-light nodes. Overall, by having $M \geq 5B$, where two blocks are used to hold copies of a node w in T, two blocks are used to hold copies of a child of w and one block is used to scan gray subtrees, the statement follows. □

2.5 Implementation

The algorithms of Sections 2.3 and 2.4 have been implemented in the C++ programming language. A high level overview of each implementation is illustrated in Figure 2.11. The source code is publicly available and can be found at `https://github.com/kmampent/CacheTD`.

2.5.1 Input

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

2.5.2 Parser

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

2.5.3 Algorithm

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

Initialization. In the initialization step, the preprocessing parts of the algorithms are performed (see Sections 2.3.3 and 2.4.2), where the first component of T_1 is built, and the corresponding contracted version of T_2, from now on referred to as *corresponding component* of T_2, is built as well. After this step, the first component of T_1 is stored in an array (different than the one produced by the parser) following the preorder layout. Similarly, the corresponding component of T_2 is stored in an array following the postorder layout.

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

- Starting from the root of $\texttt{comp}(T_1)$ and according to Section 2.3.2, scan the leftmost path of $\texttt{comp}(T_1)$ to find the splitting node u.
- Scan $\texttt{comp}(T_2)$ to compute for the binary algorithm $\sum_{v \in T_2} |s(u) \cap s(v)|$ (see counting phase of T_2 in Section 2.3.3), or for the general algorithm $\sum_{(v,c') \in T_2} |s''(u,c) \cap s'(v,c')|$ (see counting phase of T_2 in Section 2.4.2).
- Using the splitting node u, generate the next three components of T_1. Let $\texttt{comp}(T_1(u_l))$, $\texttt{comp}(T_1(u_r))$, and $\texttt{comp}(T_1(u_p))$ be the components determined by the left child, right child, and parent of u respectively. Let $\texttt{comp}(T_2(u_l))$, $\texttt{comp}(T_2(u_r))$ and $\texttt{comp}(T_2(u_p))$ be the corresponding contracted versions of T_2 with all the necessary counters properly maintained (see contraction phase of T_2 in Section 2.3.3 for the binary case and in Section 2.4.2 for the general case).
- Scan and contract $\texttt{comp}(T_2)$ to generate $\texttt{comp}(T_2(u_l))$ and then recurse on the pair defined by $\texttt{comp}(T_1(u_l))$ and $\texttt{comp}(T_2(u_l))$.
- Scan and contract $\texttt{comp}(T_2)$ to generate $\texttt{comp}(T_2(u_r))$ and then recurse on the pair defined by $\texttt{comp}(T_1(u_r))$ and $\texttt{comp}(T_2(u_r))$.
- Scan and contract $\texttt{comp}(T_2)$ to generate $\texttt{comp}(T_2(u_p))$ and then recurse on the pair defined by $\texttt{comp}(T_1(u_p))$ and $\texttt{comp}(T_2(u_p))$.

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

Correctness. The correctness of our implementations was extensively tested by generating hundreds of thousands of random trees of varying size and varying degree and comparing the output of our implementations against the output of the implementations of the $O(n \log^3 n)$ algorithm in [45] and the $O(n \log n)$ algorithm in [68].

Changing the Leaf Labels. To get the right theory bounds, changing the leaf labels of T_1 and T_2 must be done with a cache oblivious sorting routine, e.g., merge sort. In the RAM model this approach takes $O(n \log n)$ time and in the cache oblivious model $O(\frac{n}{B} \log_2 \frac{n}{M})$ I/Os. A second approach is to exploit the fact that each label is between 1 and n and use an auxiliary array that stores the new labels of the leaves in T_1, which we then use to update the leaf labels of T_2. In the RAM model this second approach takes $O(n)$ time but in the cache oblivious model $O(n)$ I/Os. In practice, the problem with the first approach is that the number of instructions it incurs eliminates any advantage

that we expect to get due to its cache related efficiency for L_1, L_2, and L_3 cache. For the input sizes tested, the array of labels easily fits into RAM, so in our implementation of both algorithms we use the second approach.

2.6 Experiments

In this section we provide an extensive experimental evaluation of the practical performance of the algorithms described in Sections 2.3 and 2.4.

2.6.1 The Setup

The experiments were performed on a machine with 8GB RAM, Intel(R) Core(TM) i5-3470 CPU @ 3.20GHz, 32K L1 cache, 256K L2 cache and 6144K L3 cache. The operating system was Ubuntu 16.04.2 LTS. The compilers used were g++ 5.4 and g++ 4.7, together with cmake 3.5.1. The experiments were performed in text mode, i.e., by booting into the terminal of Ubuntu, to minimize the interference from other programs running at the same time.

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

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

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

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

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

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

Implementations Tested. Let p_1 and p_2 denote the contraction probability of T_1 and T_2 respectively. When $p_1 = p_2 = 0$, the trees T_1 and T_2 are binary trees, so in the experiments we use the algorithm from Section 2.3. In every other case, the algorithm from Section 2.4 is used. Note that the algorithm from Section 2.4 can handle binary trees just fine, however there is an extra overhead (factor 1.8 slower, see Figure A.1) compared to the algorithm from Section 2.3 that comes due to the additional counters that we maintain in the contractions of T_2.

We compared our implementation with the implementations that are provided in [45] and [12, 66], and are available at `http://sunflower.kuicr.kyoto-u.ac.jp/~jj/Software/Software.html` and `http://users-cs.au.dk/cstorm/software/tqdist/` respectively. The implementation of the algorithm in [45] has two versions, one that uses `unordered_map`[3], which we refer to as `CPDT`, and another that uses sparsehash[4], which we refer to as `CPDTg`. For binary input trees the hash maps are not used, thus `CPDT` and `CPDTg` are the same. The tqdist library [68], which we refer to as `tqDist`, has an implementation of the binary $O(n \log^2 n)$ algorithm from [66] and the general $O(n \log n)$ algorithm from [12]. If the two input trees are binary the $O(n \log^2 n)$ algorithm is used. We refer to our new algorithm as `CacheTD`.

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

2.6.2 Results

The experiments are divided into two parts. In the first part, we consider the performance of the algorithms when their memory requirements do not exceed the available main memory (8G RAM). In the second part, we consider the performance when the memory requirements exceed the available main memory (by limiting the available RAM to the operating system to be 1GB), thus forcing the operating system to start using the swap space, which in turn yields the very expensive disk I/Os. All figures can be found in Appendix A.

RAM experiments in the Random Model. In Figure A.2 we illustrate a time comparison of all implementations for trees of up to 2^{21} leaves ($\sim$ 2 million) with varying contraction probabilities. Every experiment is run 10 times, and each time on a different tree. All 10 data points are depicted together with a line that goes over their median. The compilers used were g++ 5.4 with cmake 3.5.1 for `tqDist` and g++ 5.4 for `CPDT`, `CPDTg`, and

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

`CacheTD`. In all cases, `CacheTD` achieves the best performance. We note that for the case where $p_1 = 0.95$ and $p_2 = 0.2$, `CPDT` behaves in a different way compared to the experiments in [45]. The same can be observed for the case where $p_1 = 0.8$ and $p_2 = 0.8$. The reason is because of the differences in the implementation of `unordered_map` that exist between the different versions of the g++ compilers. In Figure A.3 we compare the performance of `CPDT` when compiled with g++ 4.7 and g++ 5.4. When p_1 is large, i.e., $p_1 = 0.8$ and $p_1 = 0.95$, we observe that the older version of g++ achieves a better performance. For all other values of p_1, the version of the compiler has no effect on the performance. In Figure A.4 we have another time comparison of all implementations but now with `CPDT` compiled in g++ 4.7. The new algorithm achieves the best performance again, but now the behaviour of `CPDT` is more stable when p_1 is large. From now on, in every RAM experiment `CPDT` is compiled in g++ 4.7.

In Figure A.5 we show the space consumption of the algorithms. `CacheTD` is the only algorithm that uses $O(n)$ space for both binary and general trees. In theory we expect that the space consumption is better and this is also what we get in practice.

In Figures A.6 and A.7 we can see how the contraction parameter affects the running time and the space consumption of the algorithms respectively.

Finally, in Figures A.8, A.9 and A.10 we compare the cache performance of the algorithms, i.e., how many cache misses (L1, L2 and L3 respectively) the algorithms perform for increasing input sizes and varying contraction parameters. As expected, the new algorithm achieves a significant improvement over all previous algorithms.

RAM experiments in the Skewed Model. The main interesting experimental results are illustrated in Figure A.11, where we plot the alpha parameter against the execution time of the algorithms, when $n = 2^{21}$. The alpha parameter has the least effect on `CacheTD`, with the maximum running time in every graph of Figure A.11 being only a factor of 1.15 larger than the minimum. As mentioned in Section 2.2, `CPDT` and `CPDTg` use the heavy light decomposition for T_2. For binary trees, when α approaches 0 or 1, the number of heavy paths that have to be updated because of a leaf color change decreases, thus the total number of operations of the algorithm decreases as well. We can verify this in Figure A.12, where we have the plots of the alpha parameter against the instructions. The same cannot be said for all general trees, since the contraction parameters have an effect on the shape of the trees as well. In Figures A.13, A.14, and A.15 we have the same graphs but for L1, L2, and L3 cache misses respectively.

I/O experiments. In Figures A.16 and A.17 we illustrate the time, space, and I/O performance in the random and skewed model respectively. Every implementation was compiled with g++ 5.4. Every experiment is run 5 times, each on a different tree. Like in the RAM experiments, all 5 data points are displayed together with a line that goes over their median. To measure the

Table 2.2: Random model: Time performance when limiting the available RAM to be 1GB. For the left table we have $p_1 = p_2 = 0$ and for the right table $p_1 = p_2 = 0.5$.

n	CPDT	tqDist	CacheTD
2^{15}	0m:01s	0m:01s	0m:01s
2^{16}	0m:01s	0m:02s	0m:01s
2^{17}	0m:01s	0m:04s	0m:01s
2^{18}	0m:02s	1m:03s	0m:01s
2^{19}	0m:04s	1h:21m	0m:01s
2^{20}	0m:09s	**0%**	0m:01s
2^{21}	13m:12s	-	0m:03s
2^{22}	**0%**	-	0m:09s
2^{23}	-	-	3m:37s
2^{24}	-	-	10m:35s

n	CPDT	CPDTg	tqDist	CacheTD
2^{15}	0m:01s	0m:01s	0m:01s	0m:01s
2^{16}	0m:01s	0m:01s	0m:01s	0m:01s
2^{17}	0m:01s	0m:01s	0m:03s	0m:01s
2^{18}	0m:03s	0m:03s	0m:07s	0m:01s
2^{19}	0m:07s	0m:07s	5m:20s	0m:01s
2^{20}	3m:43s	1h:13m	**0%**	0m:02s
2^{21}	**15%**	**0%**	-	0m:20s
2^{22}	-	-	-	2m:02s
2^{23}	-	-	-	10m:42s
2^{24}	-	-	-	42m:06s

Table 2.3: Skewed model: Time performance when limiting the available RAM to be 1GB. For both tables we have $\alpha = 0.5$. For the left table we have $p_1 = p_2 = 0$ and for the right table $p_1 = p_2 = 0.5$.

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

n	CPDT	CPDTg	tqDist	CacheTD
2^{15}	0m:01s	0m:01s	0m:01s	0m:01s
2^{16}	0m:01s	0m:01s	0m:01s	0m:01s
2^{17}	0m:01s	0m:01s	0m:03s	0m:01s
2^{18}	0m:03s	0m:03s	0m:06s	0m:01s
2^{19}	0m:07s	0m:07s	3m:21s	0m:01s
2^{20}	6m:24s	2h:31m	**7h:51m**	0m:02s
2^{21}	**12%**	**0%**	-	0m:19s
2^{22}	-	-	-	1m:58s
2^{23}	-	-	-	9m:42s
2^{24}	-	-	-	38m:19s

execution time, we used the `time` function of Ubuntu and thus also took into account the time taken to parse the input trees. For the input trees of size 2^{23} and 2^{24} we used the 128 bit implementation of the new algorithms in order to avoid overflows.

Unlike `CacheTD`, the performance of `CPDT`, `CPDTg`, and `tqDist` deteriorates significantly from the moment they start performing disk I/Os. Only `CacheTD` managed to finish running in a reasonable amount of time for all input sizes. For every other algorithm, some data points are missing because the execution time required was too big. To get an idea of how big, in Tables 2.2 and 2.3 we again have the time performance of the algorithms in the random and skewed models respectively. This is the exact same time performance as depicted

in Figures A.16 and A.17, however we also include some information about how well the algorithms performed on the extra data point that is missing from the figures. We set a time limit of 10 hours, and only for one pair of input trees T_1 and T_2 we measured for how many nodes of T_1 the value of $\sum_{v \in T_2} |s(u) \cap s(v)|$ was found. Some algorithms managed to process only 0% of the total nodes in T_1, which means that they had to spend most of the time in the preprocessing step (e.g. building the HDT of T_2). The only algorithm that managed to produce a result was `tqDist`, requiring close to 8 hours for trees with 2^{20} leaves (see Table 2.3).

2.7 Conclusion

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

Future work and open problems involve the following:

- Could the new algorithms be improved so that in the analysis, the base of the logarithm becomes M/B, thus giving the sorting bound in the cache oblivious model? Would the resulting algorithm be even more efficient in practice?
- Is it possible to compute the triplet distance in $O(n)$ time?
- For the quartet distance computation, could we apply similar techniques to those described in Section 2.3 and 2.4 in order to get an algorithm with better time bounds in the RAM model that also scales to external memory?

Chapter 3

Computing the Rooted Triplet Distance between Phylogenetic Networks

[52] Jesper Jansson, Konstantinos Mampentzidis, Ramesh Rajaby, and Wing-Kin Sung. Computing the Rooted Triplet Distance Between Phylogenetic Networks. In *Combinatorial Algorithms*, pages 290–303. Springer International Publishing, 2019.

The *rooted triplet distance* measures the structural dissimilarity of two phylogenetic trees or networks by counting the number of rooted trees with exactly three leaf labels that occur as embedded subtrees in one, but not both of them. Suppose that $N_1 = (V_1, E_1)$ and $N_2 = (V_2, E_2)$ are rooted phylogenetic networks over a common leaf label set of size n, that N_i has level k_i and maximum in-degree d_i for $i \in \{1, 2\}$, and that the networks' out-degrees are unbounded. Denote $N = \max(|V_1|, |V_2|)$, $M = \max(|E_1|, |E_2|)$, $k = \max(k_1, k_2)$, and $d = \max(d_1, d_2)$. Previous work has shown how to compute the rooted triplet distance between N_1 and N_2 in $\mathrm{O}(n \log n)$ time in the special case $k \leq 1$. For $k > 1$, no efficient algorithms are known; a trivial approach leads to a running time of $\Omega(N^7 n^3)$ and the only existing non-trivial algorithm imposes restrictions on the networks' in- and out-degrees (in particular, it does not work when non-binary vertices are allowed). In this paper, we develop two new algorithms that have no such restrictions. Their running times are $\mathrm{O}(N^2 M + n^3)$ and $\mathrm{O}(M + k^3 d^3 n + n^3)$, respectively. We also provide implementations of our algorithms and evaluate their performance in practice. Our prototype implementations have been packaged into the first publicly available software for computing the rooted triplet distance between unrestricted networks of arbitrary levels.

3.1 Introduction

Background. Trees are commonly used in biology to represent evolutionary relationships, with the leaves corresponding to species that exist today and internal vertices to ancestor species that existed in the past. When studying the evolution of a fixed set of species, different available data and tree construction methods [27] can lead to trees that look structurally different. Quantifying this difference is essential to make better evolutionary inferences, which has led to the proposal of several tree distance measures in the literature. Examples of distance measures that are based on counting how many times certain features differ in the two trees are the Robinson-Foulds distance [64], the rooted triplet distance [25] for rooted trees, and the unrooted quartet distance [26] for unrooted trees. Other distance measures are the nearest-neighbor interchange distance, introduced independently in [60] and [63], the path-length-difference distance [62], the subtree prune-and-regraft distance [38], the maximum agreement subtree [28], and the tree edit distance [73].

A rooted phylogenetic network is an extension of a *rooted phylogenetic tree* (i.e., a rooted, unordered, distinctly leaf-labeled tree with no degree-1 vertices) that allows internal vertices to have more than just one parent. Such networks are designed to capture more complex evolutionary relationships when reticulation events such as horizontal gene transfer and hybridization are involved. Similarly to phylogenetic trees, it becomes useful to have distance measures for comparing phylogenetic networks. In this paper, we study a natural extension of the rooted triplet distance from the case of rooted phylogenetic trees to the case of rooted level-k phylogenetic networks, suggested by Gambette and Huber [33].

Problem Definitions. A *rooted phylogenetic network* $N' = (V, E)$ is a rooted, directed acyclic graph with one root (a vertex with in-degree 0), distinctly labeled leaves, and no vertices with both in-degree 1 and out-degree 1. Below, when referring to a "tree" we imply a "rooted phylogenetic tree" and when referring to a "network" we imply a "rooted phylogenetic network". For a vertex u in N', let $in(u)$ and $out(u)$ be the in-degree and out-degree of u. The network N' can have three types of vertices. A vertex u is an *internal vertex* if $out(u) \geq 1$, a *leaf vertex* if $in(u) = 1$ and $out(u) = 0$, and a *reticulation vertex* if $out(u) \geq 1$ and $in(u) \geq 2$. By definition, N' cannot have a vertex u with $in(u) > 1$ and $out(u) = 0$. Let $r(N')$ be the root of N' and $\mathcal{L}(N')$ the set of leaves in N'. A directed edge from a vertex u to a vertex v in N' is denoted by $u \to v$. A path from u to v in N' is denoted by $u \rightsquigarrow v$. Let the height $h(u)$ be the length (number of edges) of the longest path from u to a leaf in N'. By definition, if v is a parent of u in N', we have $h(v) > h(u)$.

Let $U(N')$ be the undirected graph created by replacing every directed edge in N' with an undirected edge. An undirected graph H is called *biconnected* if it has no vertex whose removal makes H disconnected. We call H' a *biconnected component of* $U(N')$ if H' is a maximal subgraph of $U(N')$ that is bicon-

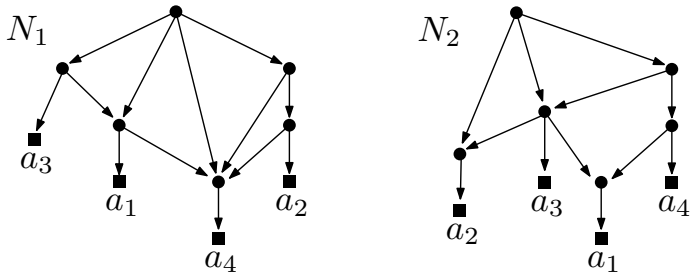


Figure 3.1: N_1 is a level-2 network and N_2 is a level-3 network with $\mathcal{L}(N_1) = \mathcal{L}(N_2) = \{a_1, a_2, a_3, a_4\}$. In this example, $D(N_1, N_2) = 6$. Some shared triplets are: $a_1|a_2|a_4$, $a_3a_4|a_2$, $a_1a_3|a_2$. Some triplets consistent with only one network are: $a_1|a_3|a_4$, $a_2a_3|a_1$.

nected. The biconnected components of $U(N')$ are edge-disjoint but not necessarily vertex-disjoint. We say that N' is a *level-k network*, or equivalently N' *has level k*, if every biconnected component of $U(N')$ contains at most k reticulation vertices. The level of a network was introduced by Choy *et al.* [20] as a parameter to measure the treelikeness of a network, with the special case of a level-0 network corresponding to a tree and a level-1 network a *galled tree* [36]. Fig. 3.1 shows a level-2 and a level-3 network.

A *rooted triplet* τ is a tree with three leaves. If it is binary we say that τ is a *rooted resolved triplet*, and if it is non-binary we say that τ is a *rooted fan triplet*. For a network N' we say that the rooted fan triplet $x|y|z$ is *consistent with* N', if and only if there exists an internal vertex u in N' such that there are three directed paths of non-zero length from u to x, from u to y, and from u to z that are vertex-disjoint except for in the vertex u. Similarly, we say that the rooted resolved triplet $xy|z$ is *consistent with* N', if and only if N' contains two internal vertices u and v ($u \neq v$), such that there are four directed paths of non-zero length from u to v, from v to x, from v to y, and from u to z that are vertex-disjoint except for in the vertices u and v, and furthermore, the path from u to z does not pass through v. See Fig. 3.1 for an example. From here on, by "disjoint paths" we imply "vertex-disjoint paths of non-zero length". Moreover, when referring to a "triplet" we imply a "rooted triplet".

Given two networks $N_1 = (V_1, E_1)$ and $N_2 = (V_2, E_2)$ built on the same leaf label set Λ of size n, the *rooted triplet distance* $D(N_1, N_2)$, or *triplet distance* for short, is the number of triplets over Λ that are consistent with exactly one of the two input networks [33] (see also [44, Section 3.2] for a discussion). Let $S(N_1, N_2)$ be the total number of triplets that are consistent with both N_1 and N_2, commonly referred to as *shared triplets*. We then have:

$$D(N_1, N_2) = S(N_1, N_1) + S(N_2, N_2) - 2S(N_1, N_2) \tag{3.1}$$

Note that a shared triplet contributes a $+1$ to $S(N_1, N_1)$, $S(N_2, N_2)$, and

Table 3.1: Previous and new results for computing $D(N_1, N_2)$, where N_1 and N_2 are two level-k networks built on the same leaf label set of size n.

Year	Reference	k	Degrees	Time
1980	Fortune *et al.* [29]	arbitrary	arbitrary	$\Omega(N^7n^3)$
2010	Byrka *et al.* [15]	arbitrary	binary	$\mathrm{O}(N^3 + n^3)$
2010	Byrka *et al.* [15]	arbitrary	binary	$\mathrm{O}(N + k^2N + n^3)$
2017	Brodal *et al.* [11, 12]	0	arbitrary	$\mathrm{O}(n \log n)$
2019	Jansson *et al.* [53]	1	arbitrary	$\mathrm{O}(n \log n)$
2019	new	arbitrary	arbitrary	$\mathrm{O}(N^2M + n^3)$
2019	new	arbitrary	arbitrary	$\mathrm{O}(M + k^3d^3n + n^3)$

$S(N_1, N_2)$, e.g., the triplet $a_1|a_2|a_4$ in Fig. 3.1. On the other hand, a triplet from either network that is not shared contributes a +1 to either $S(N_1, N_1)$ or $S(N_2, N_2)$, and a 0 to $S(N_1, N_2)$, e.g., $a_1|a_3|a_4$ from Fig. 3.1 contributes a +1 to $S(N_1, N_1)$ and a 0 to $S(N_2, N_2)$ and $S(N_1, N_2)$. Let $S_r(N_1, N_2)$ and $S_f(N_1, N_2)$ be the total number of resolved and fan triplets respectively that are consistent with both N_1 and N_2. We then have that $S(N_1, N_2) = S_r(N_1, N_2) + S_f(N_1, N_2)$.

We define the following notation that we use from here on. A network N_i is built on a leaf label set of size n and is defined by the vertex set V_i and the edge set E_i. Moreover, N_i has level k_i and the maximum in-degree of every vertex in N_i is d_i. Two given networks N_1 and N_2 are built on the same leaf label set Λ and $N = \max(|V_1|, |V_2|)$, $M = \max(|E_1|, |E_2|)$, $k = \max(k_1, k_2)$, and $d = \max(d_1, d_2)$.

Related Work. Table 3.1 lists the running times of different algorithms for computing $D(N_1, N_2)$. When $k = 0$, both N_1 and N_2 are trees. This case has been extensively studied [5, 11, 12, 23, 25, 45, 68, 72] in the literature, with the fastest algorithm in theory and practice by Brodal *et al.* [11, 12] running in $\mathrm{O}(n \log n)$ time. For $k = 1$, an $\mathrm{O}(n^{2.687})$-time algorithm based on counting 3-cycles in an auxiliary graph was given in [44], and a faster, $O(n \log n)$-time algorithm that transforms the input to a constant number of instances with $k = 0$ was given in [53]. All algorithms mentioned above allow vertices of arbitrary degree in the input networks. Moreover, software packages implementing the $\mathrm{O}(n \log n)$-time algorithms are available.

For $k > 1$, Byrka *et al.* [15] considered the special case of networks whose roots have out-degree 2 and whose other non-leaf vertices have in-degree 2 and out-degree 1 or in-degree 1 and out-degree 2. For such a network $N' = (V, E)$, they defined a data structure $\mathcal{D}$ that can be constructed in $O(|V|^3)$ time by dynamic programming and then used to determine in $O(1)$ time if any resolved triplet $xy|z$ is consistent with N'. This result was then strengthened by obtaining a new data structure $\mathcal{D}'$ that requires $\mathrm{O}(|V| + k^2|V|)$ construction time, where k is the level of N'. If N_1 and N_2 have arbitrary levels and follow the

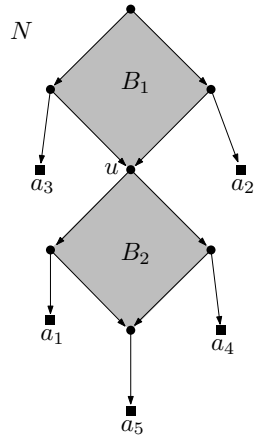


Figure 3.2: A network N' that follows the degree constraints of Byrka *et al.* [15] except the vertex u which has out-degree 2 instead of 1. The two biconnected components B_1 and B_2 share the vertex u, i.e., B_1 and B_2 are not vertex-disjoint.

degree constraints of N', $\mathcal{D}$ can be used to compute $D(N_1, N_2)$ in $\mathrm{O}(N^3 + n^3)$ time and $\mathcal{D}'$ can be used to compute $D(N_1, N_2)$ in $\mathrm{O}(N + k^2N + n^3)$ time.

Contribution. The data structures $\mathcal{D}$ and $\mathcal{D}'$ of Byrka *et al.* [15] can only support consistency queries for resolved triplets. However, a network with vertices of arbitrary degree may contain fan triplets. Moreover, $\mathcal{D}'$ exploits the fact that given the degree constraints in N', all biconnected components of $U(N')$ are vertex-disjoint. However, even a small change in these constraints, e.g., if we allow vertices with in-degree 2 to have an out-degree 2 instead of 1, could produce a network with biconnected components that are not vertex-disjoint (e.g., see Fig. 3.2), thus making the application of $\mathcal{D}'$ impossible.

Without any degree constraints in N_1 and N_2 and when k_1 and k_2 are arbitrary, an algorithm for computing $D(N_1, N_2)$ that iterates over all $4\binom{n}{3}$ triplets and for each triplet applies the pattern matching algorithm in [29] to determine its consistency with N_1 and N_2, has a $\Omega(N^7n^3)$ running time. In this paper we give two algorithms that improve significantly upon this approach. The running time of the first algorithm is $\mathrm{O}(N^2M + n^3)$ and the second algorithm $\mathrm{O}(M + k^3d^3n + n^3)$. For networks N_1 and N_2 that satisfy the degree constraint in Byrka *et al.* [15], we prove that our algorithms can compute $D(N_1, N_2)$ using the same time complexity as that of Byrka *et al.* [15]. To determine the efficiency of the two algorithms in practice, we provide an implementation as well as extensive experiments on both simulated and real datasets. We note that our prototype implementations have been packaged into the first publicly available software for computing the triplet distance between two unrestricted networks of arbitrary levels.

Organization of the Article. In Section 3.2 we present the first algorithm and in Section 3.3 the second algorithm. Section 3.4 presents an implementation of the two algorithms as well as experiments illustrating their practical performance. Finally, Section 3.5 presents our concluding remarks.

3.2 A First Approach

In this section we describe an algorithm that for two given networks N_1 and N_2 can compute $D(N_1, N_2)$ in $\mathrm{O}(N^2M + n^3)$ time.

Overview. The algorithm consists of a preprocessing step and a triplet distance computation step. In the preprocessing step, we extend a technique introduced by Shiloach and Perl [70] in 1978 that was used to solve the problem of finding two disjoint paths between two pairs of vertices, to construct suitably defined auxiliary graphs that compactly encode disjoint paths within N_1 and N_2. Two graphs, the *fan graph* and *resolved graph*, are created that enable us to check the consistency of any fan triplet and any resolved triplet, respectively, with N_1 and N_2 in $\mathrm{O}(1)$ time. In the triplet distance computation step, we compute $D(N_1, N_2)$ by iterating over all possible $4\binom{n}{3}$ triplets and using the fan and resolved graphs to check the consistency of each triplet with N_1 and N_2 efficiently.

3.2.1 Preprocessing

Let $G = (V, E)$ be a directed acyclic graph and s_1, t_1, s_2, and t_2 four vertices in G. Shiloach and Perl [70] gave an algorithm that can find two vertex-disjoint paths, one from s_1 to t_1 and one from s_2 to t_2, in $O(|V||E|)$ time or determine that no such pair of paths exists. They achieve this by creating a directed graph $G' = (V', E')$ in $\mathrm{O}(|V||E|)$ time, with the property that the existence of such a pair of vertex-disjoint paths in G is equivalent to the existence of a directed path from $\langle s_1, s_2\rangle$ to $\langle t_1, t_2\rangle$ in G', where $\langle s_1, s_2\rangle$ and $\langle t_1, t_2\rangle$ are vertices in G'. A fan triplet or resolved triplet involves more than two vertex-disjoint paths, and below we show exactly how to extend the technique by Shiloach and Perl [70] to determine if a given network has the necessary vertex-disjoint paths that would imply the consistency of a given triplet with the network.

Fan Graph. For any network N_i, let the *fan graph* $N_i^f = (V_i^f, E_i^f)$ be a graph such that $V_i^f = \{s\} \cup \{(u, v, w) \mid u, v, w \in V_i, u \neq v, u \neq w, v \neq w\}$ and E_i^f includes the following edges:

1. $\{(u_1, v_1, w_1) \to (u_2, v_1, w_1) \mid u_1 \to u_2 \in E_i, h(u_1) \geq \max(h(v_1), h(w_1))\}$
2. $\{(u_1, v_1, w_1) \to (u_1, v_2, w_1) \mid v_1 \to v_2 \in E_i, h(v_1) \geq \max(h(u_1), h(w_1))\}$
3. $\{(u_1, v_1, w_1) \to (u_1, v_1, w_2) \mid w_1 \to w_2 \in E_i, h(w_1) \geq \max(h(u_1), h(v_1))\}$
4. $\{s \to (u, v, w) \mid u \to v \in E_i, u \to w \in E_i\}$

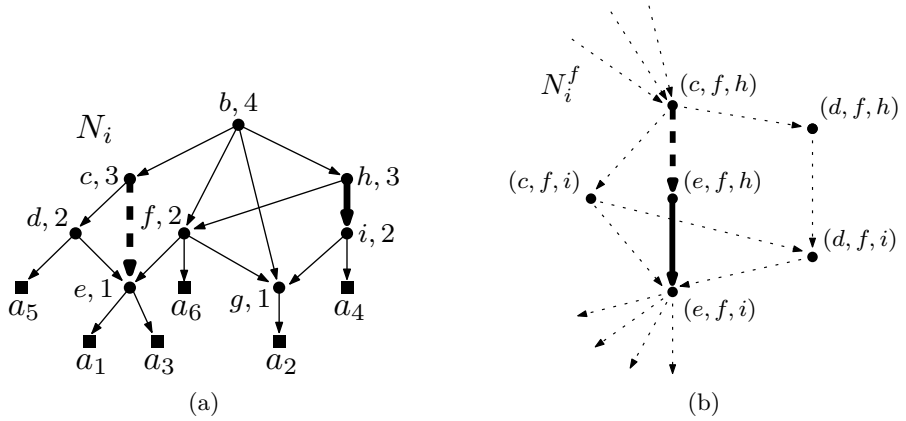


Figure 3.3: (a) An example network N_i. On top of every internal vertex we have a name and the height. (b) A small part of the fan graph N_i^f is drawn. For the triplet $a_3|a_6|a_4$ we draw two steps, $(c, f, h) \to (e, f, h)$ and $(e, f, h) \to (e, f, i)$ of the following path that exists in N_i^f: $s \to (b, f, h) \to (c, f, h) \to (e, f, h) \to (e, f, i) \to (e, a_6, i) \to (e, a_6, a_4) \to (a_3, a_6, a_4)$.

Fig. 3.3 has an example. Note that N_i^f contains $\mathrm{O}(|V_i|^3)$ vertices, $\mathrm{O}(|V_i|^2|E_i|)$ edges, and also has the property described in the following lemma:

Lemma 6. *Consider a network N_i and its fan graph $N_i^f = (V_i^f, E_i^f)$. For any three different leaves x, y, and z in N_i, vertex s can reach vertex (x, y, z) in N_i^f if and only if $x|y|z$ is a fan triplet in N_i.*

Proof. The below proof is a generalization of the proof of Theorem 3.1 in [70].

($\leftarrow$) Let $x|y|z$ be a fan triplet in N_i. This means that there exists an internal vertex q in N_i and three disjoint paths, except for q, one from q to x, one from q to y, and one from q to z. Let those three paths be $(q, x_0, x_1, \ldots, x_a)$, $(q, y_0, y_1, \ldots, y_b)$ and $(q, z_0, z_1, \ldots, z_c)$, where $x_a = x$, $y_b = y$, and $z_c = z$. We construct a path P in N_i^f from s to (x, y, z) as follows:

- $s \to (q, y_0, z_0)$ since $q \to y_0 \in E_i$ and $q \to z_0 \in E_i$.
- $(q, y_0, z_0) \to (x_0, y_0, z_0)$ since $q \to x_0 \in E_i$ and $h(q) > h(y_0), h(z_0)$.
- As $h(x_0) > h(x_1) > \cdots > h(x_a)$, $h(y_0) > h(y_1) > \cdots > h(y_b)$ as well as $h(z_0) > h(z_1) > \cdots > h(z_c)$, and $(x_0, \ldots, x_a)$, $(y_0, \ldots, y_b)$ and $(z_0, \ldots, z_c)$ are paths in N_i, there exists a path in N_i^f from (x_0, y_0, z_0) to (x_a, y_b, z_c).

By combining the above three paths, we have a path in N_i^f from s to (x, y, z). An example can be found in Fig. 3.3.

($\rightarrow$) Because s can reach (x, y, z) in N_i^f, there exists a path P in N_i^f for which we have that $P = (s, (x_1, y_1, z_1), (x_2, y_2, z_2), \ldots, (x_t, y_t, z_t))$, where $x_t = x$, $y_t = y$, and $z_t = z$. Let $P_1 = (x_1, \ldots, x_t)$, $P_2 = (y_1, \ldots, y_t)$, and $P_3 = (z_1, \ldots, z_t)$, where $x_t = x$, $y_t = y$, and $z_t = z$. By the definition of the

fan graph N_i^f, for every $j \in \{1, \ldots, t-1\}$ we have either of the following three cases: (1) $x_j \neq x_{j+1}$ only, (2) $y_j \neq y_{j+1}$ only, and (3) $z_j \neq z_{j+1}$ only. We show by induction that P_1, P_2, and P_3 are disjoint paths, meaning that $x|y|z$ is a fan triplet in N_i. When $i = t$, all three vertices x_t, y_t, and z_t are different by the definition of V_i^f. For $j \geq 1$, by the inductive hypothesis we have that $(x_{j+1}, \ldots, x_t)$, $(y_{j+1}, \ldots, y_t)$ and $(z_{j+1}, \ldots, z_t)$ are disjoint paths. Again by the definition of N_i^f, we have either one of the following three cases: (1) $x_j \neq x_{j+1}$ only, (2) $y_j \neq y_{j+1}$ only, and (3) $z_j \neq z_{j+1}$ only. For (1), note that $y_j = y_{j+1}$, $z_j = z_{j+1}$, which means that $(x_{j+1}, \ldots, x_t)$, $(y_j, \ldots, y_t)$ and $(z_j, \ldots, z_t)$ are disjoint paths. We now show that x_j does not appear in any of the three previous paths. We have $h(x_j) \geq \max(h(y_j), h(z_j))$, thus for $\mu \geq j+1$ and $y_\mu \neq y_j$ we have $h(x_j) > h(y_\mu)$. Similarly, for $\mu \geq j+1$ and $z_\mu \neq z_j$ we have $h(x_j) > h(z_\mu)$. Together with the fact that x_j, y_j, and z_j are different, we have that the paths $(x_j, \ldots, x_t)$, $(y_j, \ldots, y_t)$ and $(z_j, \ldots, z_t)$ are disjoint. The cases (2) and (3) can be argued similarly, thus P_1, P_2, and P_3 are disjoint paths, which means that $x|y|z$ is a fan triplet in N_i. □

Corollary 1. *Let N_i be a given network and r' a dummy leaf attached to $r(N_i)$. For any two different leaves x and y in N_i that are not r', there are two paths from $r(N_i)$ to x and y that are disjoint, except for in the vertex $r(N_i)$, if and only if s can reach (r', x, y) in N_i^f.*

Resolved Graph. For any network N_i, let the *resolved graph* $N_i^r = (V_i^r, E_i^r)$ be a graph such that $V_i^r = \{s\} \cup \{(u, v) \mid u, v \in V_i, u \neq v\} \cup \{(u, v, w) \mid u, v, w \in V_i, u \neq v, u \neq w, v \neq w\}$ and E_i^r includes the following edges:

1. $\{s \to (u, v) \mid u \to v \in E_i\}$
2. $\{(u_1, v_1) \to (u_2, v_1) \mid u_1 \to u_2 \in E_i, h(u_1) \geq h(v_1)\}$
3. $\{(u_1, v_1) \to (u_1, v_2) \mid v_1 \to v_2 \in E_i, h(v_1) \geq h(u_1)\}$
4. $\{(u, v) \to (u, v, w) \mid v \to w \in E_i, h(v) \geq h(u)\}$
5. $\{(u_1, v_1, w_1) \to (u_2, v_1, w_1) \mid u_1 \to u_2 \in E_i, h(u_1) \geq \max(h(v_1), h(w_1))\}$
6. $\{(u_1, v_1, w_1) \to (u_1, v_2, w_1) \mid v_1 \to v_2 \in E_i, h(v_1) \geq \max(h(u_1), h(w_1))\}$
7. $\{(u_1, v_1, w_1) \to (u_1, v_1, w_2) \mid w_1 \to w_2 \in E_i, h(w_1) \geq \max(h(u_1), h(v_1))\}$

Note that N_i^r contains $O(|V_i|^3)$ vertices, $O(|V_i|^2|E_i|)$ edges, and also has the property described in the following lemma:

Lemma 7. *Consider a network N_i and its resolved graph $N_i^r = (V_i^r, E_i^r)$. For any three different leaves x, y, and z in N_i, vertex s can reach vertex (x, y, z) in N_i^r if and only if $x|yz$ is a resolved triplet in N_i.*

Proof. ($\leftarrow$) If $x|yz$ is a resolved triplet in N_i, the following three paths exist in N_i: (1) $(x_0, x_1, \ldots, x_a)$, (2) $(x_0, y_1, \ldots, y_j, y_{j+1}, \ldots, y_b)$, and (3) $(y_j, z_1, \ldots, z_c)$ where $x_a = x$, $y_b = y$, and $z_c = z$, that are disjoint except for x_0 and y_j. Let x_μ be a vertex such that that $h(x_{\mu-1}) > h(y_j) \geq h(x_\mu)$. Then, we obtain the path: $s \to (x_0, y_1) \to \cdots \to (x_\mu, y_j) \to (x_\mu, y_j, z_1) \to \cdots \to (x_a, y_b, z_c)$.

($\rightarrow$) If there is a path from s to (x, y, z) in N_i^r, by definition that path has the following form: $s \rightarrow (x_1, y_1) \rightarrow \cdots \rightarrow (x_q, y_q) \rightarrow (x_{q+1}, y_{q+1}, z_{q+1}) \rightarrow \ldots (x_t, y_t, z_t)$ for which we have $x_t = x$, $y_t = y$, and $z_t = z$. By definition we have $x_1 \rightarrow y_1 \in E_i$, $x_q = x_{q+1}$, $y_q = y_{q+1}$, and $y_q \rightarrow z_{q+1} \in E_i$. Define the following three paths $P_1 = (x_1, \ldots, x_t)$, $P_2 = (y_1, \ldots, y_t)$, and $P_3 = (z_{q+1}, \ldots, z_t)$. We claim that these three paths are disjoint, meaning that $x|yz$ is a resolved triplet in N_i. We prove this claim, by showing that the following paths are disjoint:

(a) $(x_1, \ldots, x_q)$ and $(y_1, \ldots, y_q)$
(b) $(x_{q+1}, \ldots, x_t)$, $(y_{q+1}, \ldots, y_t)$, and $(z_{q+1}, \ldots, z_t)$
(c) $(x_1, \ldots, x_q)$ and $(y_{q+1}, \ldots, y_t)$
(d) $(x_1, \ldots, x_q)$ and $(z_{q+1}, \ldots, z_t)$
(e) $(y_1, \ldots, y_q)$ and $(z_{q+1}, \ldots, z_t)$
(f) $(y_1, \ldots, y_q)$ and $(x_{q+1}, \ldots, x_t)$

To show that the paths in (a) are disjoint we use induction. By the definition of N_i^r, we know that $x_q \neq y_q$. For the inductive hypothesis, assume that the paths $(x_{j+1}, \ldots, x_q)$ and $(y_{j+1}, \ldots, y_q)$ are disjoint. Again by definition we have either of the two following cases: (1) $x_j \neq x_{j+1}$ only and (2) $y_j \neq y_{j+1}$ only. For case (1), we have $y_j = y_{j+1}$ and $h(x_j) \geq h(y_j)$, thus for $\mu > j + 1$ and $y_\mu \neq y_j$ we have $h(x_j) > h(y_\mu)$. Together with the fact that $x_j \neq y_j$ we have that x_j does not appear in $(y_j, \ldots, y_q)$. Case (2) can be argued similarly, so the paths in (a) are disjoint. To show that the paths in (b) are disjoint, the same proof by induction as in Lemma 6 can be used.

To show that the paths in (c) are disjoint, let $j \in \{1, \ldots, q\}$ be the largest index such that $x_j \neq x_q$. We know from the paths in (b) that $x_q = x_{q+1}$ does not appear in $(y_{q+1}, \ldots, y_t)$, so we only need to prove that $(x_1, \ldots, x_j)$ is disjoint from $(y_{q+1}, \ldots, y_t)$. Because $x_j \neq x_q$, there exists some $\mu \in \{1, \ldots, q\}$ such that $(x_j, y_\mu) \rightarrow (x_q, y_\mu)$ is in the path from s to (x, y, z). By definition $x_j \neq y_\mu$ and $h(x_j) \geq h(y_\mu)$. We consider the following two cases: (1) $h(x_j) > h(y_\mu)$ and (2) $h(x_j) = h(y_\mu)$. For (1), because we have that $h(x_1), \ldots, h(x_j) > h(y_\mu), \ldots, h(y_t)$, the paths in (c) are disjoint. For (2), let $g \in \{1, \ldots, j\}$ be the maximum index such that $x_g \neq x_j$. Since we have that $h(x_g) > h(x_j) = h(y_\mu)$, using the same argument as in case (1) we obtain that $(x_1, \ldots, x_g)$ and $(y_\mu, \ldots, y_t)$ are disjoint. It only remains to show that x_j does not appear in $(y_\mu, \ldots, y_t)$. If we assume that x_j appears in $(y_\mu, \ldots, y_t)$, then because $y_\mu \neq x_j$ we have $h(y_\mu) > h(x_j)$, which leads to a contradiction.

For the paths in (d) similar arguments can be used as in (c), since we have $y_q \rightarrow z_{q+1} \in E_i$ and by definition $x_q = x_{q+1}$ and $x_{q+1} \neq z_{q+1}$. To show that the paths in (e) are disjoint as well, because $y_q \rightarrow z_{q+1} \in E_i$, we have that $h(y_1), \ldots, h(y_q) > h(z_{q+1}), \ldots, h(z_t)$, meaning that the paths in (e) are disjoint. Finally, to show that the paths in (f) are disjoint, by definition we have $x_q = x_{q+1}$ and $h(y_q) \geq h(x_q)$. So for every $\mu > q + 1$ and $x_\mu \neq x_q$ we have $h(y_q) > h(x_\mu)$. Since, we also have that $x_q \neq y_q$, the paths in (f) are disjoint. $\square$

Algorithm 2 Computing $D(N_1, N_2)$ using the data structures from Section 3.2.

procedure PREPROCESSING(N_1, N_2) ▷ Building the data structures
  for $i \in \{1, 2\}$ **do**
    build $N_i^f = (V_i^f, E_i^f)$ and $N_i^r = (V_i^r, E_i^r)$
    let A_i^f, A_i^r be $n \times n \times n$ arrays initialized with 0 entries
    for $x, y, z \in \Lambda$ **do**
      $A_i^f[x][y][z] = 1$ if s can reach (x, y, z) in N_i^f
      $A_i^r[x][y][z] = 1$ if s can reach (x, y, z) in N_i^r
  return $(A_1^r, A_1^f, A_2^r, A_2^f)$

procedure $S_f(A_1^f, A_2^f)$ ▷ Finding the shared fan triplets
  $sharedF = 0$
  for $x, y, z \in \Lambda$ **do**
    if $A_1^f[x][y][z] = A_2^f[x][y][z] = 1$ **then** $sharedF = sharedF + 1$
  return $sharedF$

procedure $S_r(A_1^r, A_2^r)$ ▷ Finding the shared resolved triplets
  $sharedR = 0$
  for $x, y, z \in \Lambda$ **do**
    if $A_1^r[x][y][z] = A_2^r[x][y][z] = 1$ **then** $sharedR = sharedR + 1$
    if $A_1^r[x][z][y] = A_2^r[x][z][y] = 1$ **then** $sharedR = sharedR + 1$
    if $A_1^r[y][z][x] = A_2^r[y][z][x] = 1$ **then** $sharedR = sharedR + 1$
  return $sharedR$

procedure $S(A_1^r, A_1^f, A_2^r, A_2^f)$ ▷ Finding the shared triplets
  return $S_f(A_1^f, A_2^f) + S_r(A_1^r, A_2^r)$

procedure $D(N_1 = (V_1, E_1), N_2 = (V_2, E_2))$ ▷ Computing $D(N_1, N_2)$
  $(A_1^r, A_1^f, A_2^r, A_2^f) =$ PREPROCESSING(N_1, N_2)
  return $S(A_1^r, A_1^f, A_1^r, A_1^f) + S(A_2^r, A_2^f, A_2^r, A_2^f) - 2S(A_1^r, A_1^f, A_2^r, A_2^f)$

Corollary 2. *Let N_i be a given network and r' a dummy leaf attached to $r(N_i)$. For any two different leaves x and y in N_i that are not r', there are two paths from some internal vertex $z \neq r(N_i)$ in N_i, to x and y that are disjoint, except for in the vertex z, if and only if s can reach (r', x, y) in N_i^r.*

Given N_i^f and N_i^r, we define the $n \times n \times n$ *fan table* A_i^f and the $n \times n \times n$ *resolved table* A_i^r as follows. For three different leaves x, y, and z, $A_i^f[x][y][z] = 1$ if the fan triplet $x|y|z$ is consistent with N_i and $A_i^f[x][y][z] = 0$ otherwise. Similarly, $A_i^r[x][y][z] = 1$ if the resolved triplet $x|yz$ is consistent with N_i and $A_i^r[x][y][z] = 0$ otherwise. Due to Lemmas 6 and 7, both A_i^f and A_i^r can be computed by a depth first traversal (starting from s) of N_i^f and N_i^r.

More precisely, $A_i^f[x][y][z] = 1$ if s can reach (x, y, z) in N_i^f and 0 otherwise. Finally, $A_i^r[x][y][z] = 1$ if s can reach (x, y, z) in N_i^r and 0 otherwise.

3.2.2 Triplet Distance Computation

Algorithm 2 summarizes all the procedures needed to compute the triplet distance between two given networks N_1 and N_2. For every $i \in \{1, 2\}$ the tables A_i^f and A_i^r are built in lines 2-7. These tables are then used in lines 11-12 and 16-19 to determine in O(1) time if a triplet is consistent with N_1 or N_2. Procedures $S_f()$ and $S_r()$ count the number of shared fan and resolved triplets. Both procedures enumerate over all possible triplets and use the tables A_i^f and A_i^r to determine their consistency with either network. The correctness is ensured by Lemmas 6 and 7. Procedure $S()$ reports the number of shared triplets, which is the sum of the number of shared fan triplets and shared resolved triplets. The main procedure is $D()$. It uses Equation 3.1 to determine $D(N_1, N_2)$.

To analyze the running time, after the preprocessing is finished, the procedures $S_f()$ and $S_r()$ require $O(n^3)$ time. For the total preprocessing time, by definition, building the data structures N_i^r and N_i^f for $i \in \{1, 2\}$ in line 3, requires $O(|V_1|^2|E_1| + |V_2|^2|E_2|)$ time. Building the auxiliary arrays A_i^r and A_i^f in lines 5-7 is performed by a depth first traversal of N_i^r and N_i^f, thus requiring $O(|V_1|^2|E_1|+|V_2|^2|E_2)$ time as well. Hence, the total time of the algorithm becomes $O(|V_1|^2|E_1| + |V_2|^2|E_2| + n^3)$. By the definition of N and M from Section 3.1, the running time becomes $O(N^2M + n^3)$. Hence, we obtain the following theorem:

Theorem 2. *There exists an algorithm that computes the triplet distance between two networks N_1 and N_2 in $O(N^2M + n^3)$ time.*

Let N_1 and N_2 follow the degree constraints of Byrka *et al.* [15]. We then have $N = \Theta(M)$ and the bound becomes $O(N^3 + n^3)$, thus matching the bound achieved by the first data structure of Byrka *et al.* [15].

3.3 A Second Approach

In this section we extend the algorithm from Section 3.2 in order to exploit the information about the level of the two input networks. More specifically, we describe an algorithm that for two given networks N_1 and N_2 can compute $D(N_1, N_2)$ in $O(M + k^3d^3n + n^3)$ time.

Overview. In the first approach, for a given network N_i we built the fan and resolved graph presented in Lemmas 6 and 7. In this second approach, for every biconnected component of $U(N_i)$ we define a network of approximately the same size as the biconnected component, which we call *contracted block network*. For this contracted block network we then build the corresponding

fan and resolved graph. By carefully contracting every biconnected component of $U(N_i)$ into one vertex we obtain a tree, which we call *block tree*. We finally show how to combine the block tree and all the fan and resolved graphs of the contracted block networks of N_i to count triplets efficiently.

3.3.1 Preprocessing

Let N_i be a given network. From here on, we call a biconnected component of $U(N_i)$ a *block*. For simplicity, when we refer to a block of N_i, we imply a block of $U(N_i)$. We say that for a block B of N_i, vertex $r(B)$ is the root of B, if $r(B)$ has the largest height in N_i among all vertices in B. Note that because N_i has one root that can reach every vertex of N_i and B corresponds to a maximal subgraph of $U(N_i)$ that is biconnected, B can only contain one root. If B contains only one edge $u \to v$ such that $v \in \mathcal{L}(N_i)$, then B is called a *leaf block*, otherwise B is called a *non-leaf block*. Lemma 8 presents a property of all blocks of N_i.

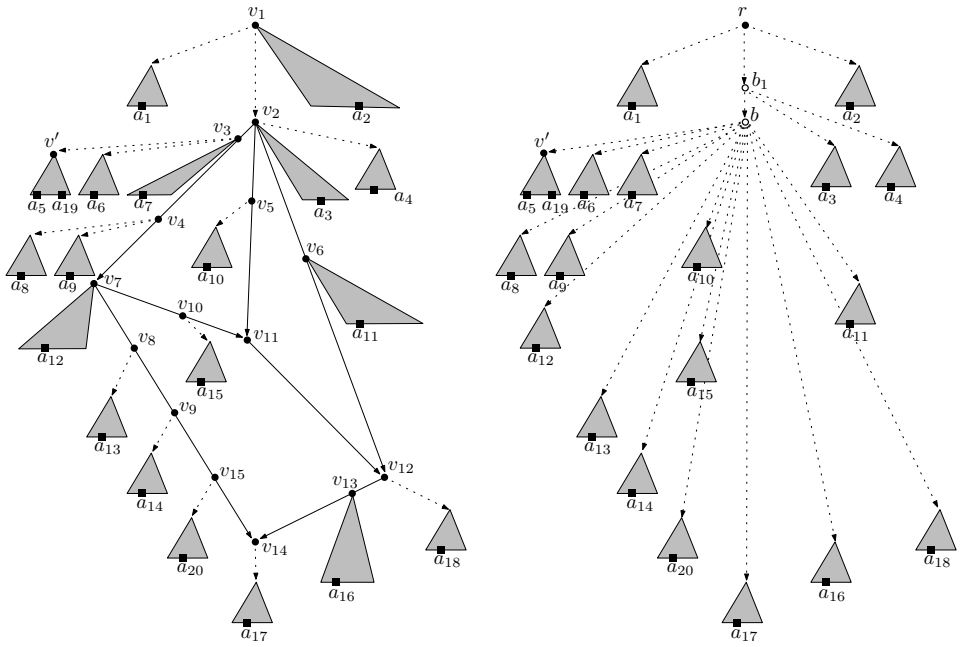
Lemma 8. *All blocks of a given network N_i are edge-disjoint.*

Proof. Assume for contradiction that N_i has two different blocks $B_1 = (V_1, E_1)$ and $B_2 = (V_2, E_2)$ that share an edge $u \to v$. Let $B = (V_1 \cup V_2, E_1 \cup E_2)$. By the definition of a block, $U(B_1)$ and $U(B_2)$ are connected graphs, where $U(B_1)$ and $U(B_2)$ are the undirected versions of the graphs defined by B_1 and B_2. Since B_1 and B_2 share an edge, then $U(B)$ is also connected. If any vertex w in $V_1 - \{u, v\}$ or $V_2 - \{u, v\}$ is removed from B, $U(B)$ will still be connected. If u is removed from B, $U(B)$ will remain connected because of the vertex v that it shared between B_1 and B_2. Finally, if v is removed from B, $U(B)$ will remain connected because of the vertex u that it shared between B_1 and B_2. We have thus shown that $U(B)$ is a block of N_i, thus B_1 and B_2 are not maximal subgraphs of N_i that are biconnected, meaning that B_1 and B_2 cannot be blocks of N_i. Hence, we reach a contradiction. □

Block Tree. From a high level perspective, we want to remove the cycles in $U(N_i)$ that are formed by the non-leaf blocks to obtain a directed tree on the leaf label set $\mathcal{L}(N_i)$. Let $T_i = (V', E')$ be a directed tree, from now on referred to as *block tree*, with the vertex set V' and edge set E' defined by the following steps:

- For every block B_j in N_i create a vertex b_j in T_i.
- Let B_1, B_2 be two blocks in N_i with $r(B_1) \neq r(B_2)$. If $r(B_2)$ is also a vertex in B_1 then create the edge $b_1 \to b_2$ in T_i.
- Create a root vertex ρ in T_i. For every block B_j that has $r(N_i)$ as a root, create the edge $\rho \to b_j$ in T_i.
- If B_j is a leaf block, rename b_j in T_i by the label of the leaf in B_j.

Figures 3.4 and 3.5 give example networks N_i and the corresponding block tree T_i. The set of all blocks of N_i and the vertex set $V' - r(T_i)$, i.e., the



(a) N_i (b) T_i

Figure 3.4: (a) N_i is a level-3 network, which contains one block B rooted at v_2 (the block is in thick edges). (b) The block tree T_i.

set of all vertices of T_i except the root, are bijective. An edge $b_1 \to b_2$ in T_i means that the corresponding blocks B_1 and B_2 in N_i do not have the same root and the root vertex $r(B_2)$ is a shared vertex between B_1 and B_2. Note that by the definition of a block, an edge connecting two vertices can define a block of its own (e.g., block B_9 in Fig. 3.5). The following lemma presents some properties of T_i:

Lemma 9. *Let $T_i = (V', E')$ be the block tree of a given network N_i. The block tree T_i is a directed tree that has n leaves, $|V'| = O(n)$, and $|E'| = O(n)$.*

Proof. We start by showing that T_i is a directed tree. Since every edge of T_i is directed, T_i is a directed graph. Let $U(T_i)$ be the undirected version of that graph. Since $U(N_i)$ is connected, then by the definition of E' we have that $U(T_i)$ is connected as well. We now claim that there is no cycle in $U(T_i)$, thus showing that T_i is a directed tree. A cycle in $U(T_i)$ would imply the existence of a vertex b in T_i such that $in(b) > 1$. If B is the corresponding block of b in N_i and by the definition of E', this in turn implies the existence of two different blocks B_1 and B_2 in N_i such that $r(B) \neq r(B_1)$, $r(B) \neq r(B_2)$, and $r(B)$ is a vertex in both B_1 and B_2. By the definition of N_i, the root $r(N_i)$ has a path to every vertex of N_i. This means that the two blocks B_1 and B_2

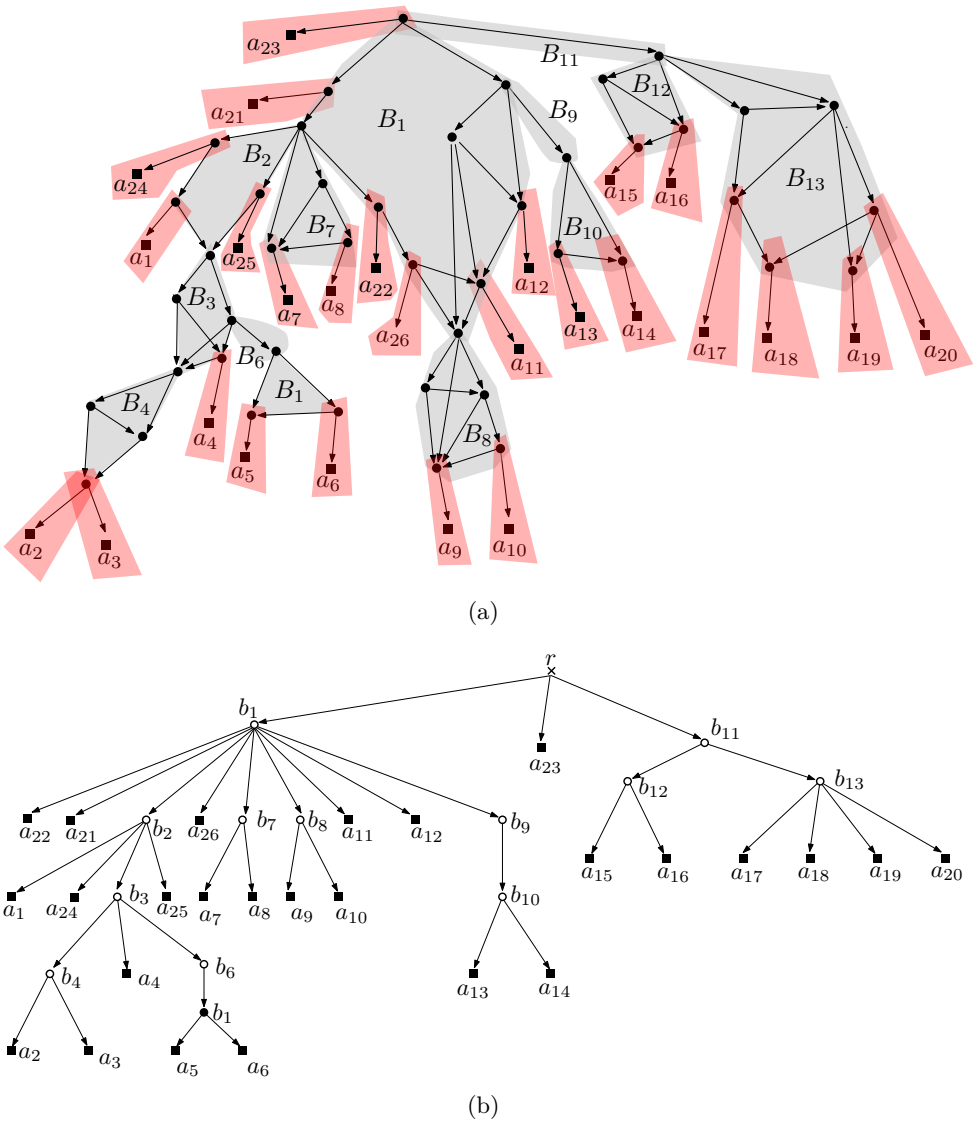


Figure 3.5: (a) An example network N_i. The blocks containing leaves are highlighted in red. Every other block is highlighted in gray. (b) The corresponding block tree T_i.

could be merged to create an even larger block that contains both, thus contradicting the assumption that B_1 and B_2 are blocks of N_i.

Let B be a block in N_i and b the corresponding vertex in T_i. By the definition of N_i, if b has an outgoing edge then B is a non-leaf block. Otherwise, a leaf exists in B, thus B is a leaf block. Hence, T_i has exactly n leaves when $|\mathcal{L}(N_i)| = n$. In T_i we allow vertices, from now on referred to as *extra vertices*, with in-degree 1 and out-degree 1. A vertex with this property, cor-

responds to a non-leaf block of N_i consisting of only 1 edge. If we contract the extra vertices, i.e., remove every such vertex u from T_i and add edges pointing from the parent of u to every child of u, we will obtain a tree $T_i'' = (V'', E'')$ with n leaves in which every internal vertex has in-degree 1 and out-degree at least 2. This means that $|V''| = \mathrm{O}(n)$ and $|E''| = \mathrm{O}(n)$. By the definition of N_i, (1) no vertex u exists in N_i such that $in(u) = out(u) = 1$. Hence, by the construction of T_i, (2) every extra vertex in T_i points to at least 1 non-extra vertex. Because T_i is a tree, (3) no two extra vertices point to the same non-extra vertex. From (1), (2), and (3) together, we have that there are $\mathrm{O}(n)$ extra vertices in T_i. This means that for the total number of vertices and edges in T_i we have $|V'| = \mathrm{O}(n)$ and $|E'| = \mathrm{O}(n)$. □

Since the set of all blocks of N_i and the set $V' - r(T_i)$ are bijective, we obtain:

Corollary 3. *A network N_i contains $\mathrm{O}(n)$ blocks.*

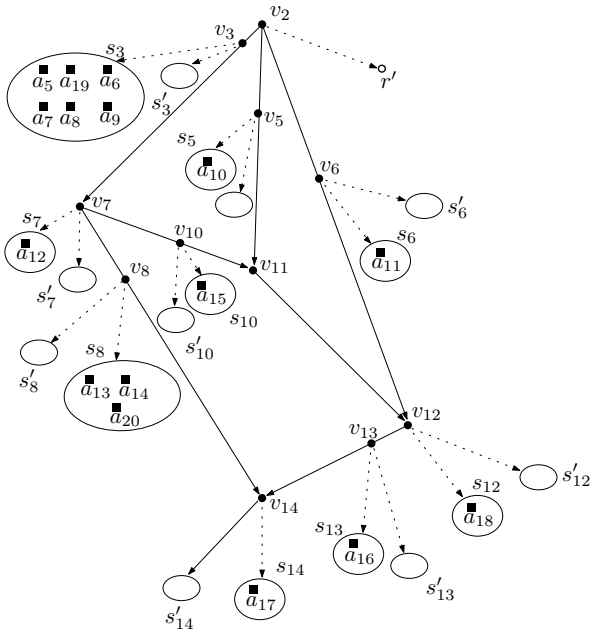
The following lemma presents an algorithm for constructing the block tree T_i:

Lemma 10. *Let $T_i = (V', E')$ be the block tree of a given network N_i. There exists an algorithm that builds T_i in $\mathrm{O}(|E_i|)$ time.*

Proof. Constructing T_i when the blocks of N_i are given is performed by scanning the vertices of N_i and the list of components that every vertex belongs to, while adding edges to T_i according to the definition of V' and E', thus requiring $\mathrm{O}(|V_i|)$ time. Finding the blocks can be performed in $\mathrm{O}(|E_i|)$ time, by applying the algorithm by Hopcroft and Tarjan in [41]. Since by definition $|E_i| \geq |V_i|$, i.e., $U(N_i)$ is connected, given N_i we can build T_i in $\mathrm{O}(|E_i|)$ time. □

Contracted Block Network. For a given network N_i, a block B in N_i, and a vertex u in B, define L_B^u to be the set of leaves that can be reached from u without using edges in B. For example, for the block in Fig. 3.4(a) we have $L_B^{v_3} = \{a_5, a_{19}, a_6, a_7\}$ and $L_B^{v_{10}} = \{a_{15}\}$. Let $C_B = (V', E')$ be a network, with the vertex set V' and edge set E' defined by the following steps:

- Let $C_B = N_i$. All operations from now on are applied on C_B.
- Remove every edge and vertex not in B.
- For every edge $u_1 \to u_2$ in B, if $in(u_1) = out(u_1) = in(u_2) = out(u_2) = 1$ contract the edge as follows: let $u_2 \to u_3$ be the other edge in B, then create the edge $u_1 \to u_3$, remove u_2 from B, and set $L_B^{u_1} = L_B^{u_1} \cup L_B^{u_2}$.
- For every vertex u_1 in C_B such that $L_B^{u_1} \neq \emptyset$, we add a child leaf with label s_1 representing all leaves in $L_B^{u_1}$. We also add another child leaf s_1' as a copy leaf that will help later on to count triplets.
- Include an artificial leaf r' which is attached to the root $r(C_B)$.



(a) C_B

Figure 3.6: The contracted block network C_B for the block B that we have in Fig. 3.4(a). Note that in C_B, the internal vertices v_3 and v_4 are merged, as well us v_8, v_9, and v_{15}. Moreover, $\{s_i, s'_i : i \in \{3, 5, 6, 7, 8, 10, 12, 13, 14\}\}$ is the set of leaves in C_B.

An example for constructing C_B from a given block B of a network N_i can be found in Fig. 3.6. Every vertex in C_B corresponds to a vertex in B and every edge between two internal vertices in C_B corresponds to a compressed path in B. For example, the edge $v_8 \to v_{14}$ in Fig. 3.6(a) corresponds to the path $v_8 \to v_9 \to v_{15} \to v_{14}$ in Fig. 3.4(a). We call C_B *the contracted block network of* N_i, corresponding to block B. The following lemma presents a property of C_B:

Lemma 11. *Let N_i be a network, B a block in N_i, and $C_B = (V', E')$ the contracted block network of N_i that corresponds to block B. We then have that $|\mathcal{L}(C_B)| = \mathrm{O}(k_i d_i + 1)$, $|V'| = \mathrm{O}(k_i d_i + 1)$, and $|E'| = \mathrm{O}(k_i d_i + 1)$.*

Proof. By the definition of a block, block B contains at most k_i reticulation vertices. Each such vertex has an arbitrary out-degree, however the in-degree is bounded by the parameter d.

If $k_i = 0$, then B consists of a single edge of N_i, meaning that by construction C_B is a binary tree on 3 leaves, thus $|E'| = |V'| = |\mathcal{L}(C_B)| = \mathrm{O}(1)$.

For any $k_i \geq 1$, we generalize the argument in the first paragraph of the proof of Lemma 3 in [15] as follows. We claim that $|V'| \leq 3(k_i + k_i d_i) + 1$, $|\mathcal{L}(C_B)| \leq 2(k_i + k_i d_i) + 1$, and $|E| \leq 2k_i(1+2d_i)+1$. If B contains one edge, then since C_B is a binary tree on 3 leaves and $d_i \geq 1$ the claim holds. Otherwise, by definition C_B contains at most k_i reticulation vertices. Every non-reticulation internal vertex can have an arbitrary out-degree, however every outgoing edge from a non-reticulation internal vertex generates a directed path that has to end in a reticulation vertex. Since there are at most k_i reticulation vertices and each reticulation vertex can end at most d_i such paths in C_B, the total number of non-reticulation internal vertices is at most $k_i d_i$. Two leaves can be attached under every reticulation vertex and non-reticulation internal vertex, plus we have the artificial leaf in C_B, thus $|\mathcal{L}(C_B)| \leq 2(k_i + k_i d_i) + 1$ and $|V'| \leq 3(k_i + k_i d_i) + 1$. For the edges, we have at most $k_i d_i$ edges pointing to reticulation vertices, $|\mathcal{L}(C_B)|$ edges pointing to leaves and at most $k_i d_i$ edges pointing to non-reticulation internal vertices. Adding all these edges together, we get $|E| \leq 2k_i(1 + 2d_i) + 1$.

Hence, for any $k_i \geq 0$ the statement follows. □

Constructing All Contracted Block Networks Efficiently. For a given network N_i and a block B in N_i, a leaf x in N_i is said to *associate with* B if there exists a vertex u in B such that $u \neq r(B)$ and $x \in L_B^u$. For example in Fig. 3.4(a), the leaf a_{16} associates with B while the leaves a_3 and a_2 do not associate with B. For any leaf x associated with some block B of N_i, let $q_B(x)$ be the vertex in B that has a path to x without using edges in B, i.e., $x \in L_B^{q_B(x)}$, $p_B(x)$ the leaf in C_B representing x and $p'_B(x)$ the copy leaf of $p_B(x)$. In the example in Fig. 3.4, $q_B(a_5) = v_3$, $p_B(a_5) = s_3$, $p'_B(a_5) = s'_3$, $q_B(a_8) = v_4$, and $p_B(a_8) = s_3$.

Lemma 8 implies an algorithm for constructing every block network C_B of N_i in $\mathrm{O}(|E_i|)$ time. As shown in the lemma below, by properly relabeling the leaves of N_i and with an additive $\mathrm{O}(n^2)$ time, it is possible to build every block network C_B so that we can afterwards compute for every leaf $l \in \mathcal{L}(N_i)$ the functions $q_B(l)$ and $p_B(l)$ in $\mathrm{O}(1)$ time.

Lemma 12. *For a network N_i, there exists an $\mathrm{O}(|E_i|+n^2)$-time algorithm that builds all the contracted block networks C_B of N_i, such that for all blocks of N_i and leaf $l \in \mathcal{L}(N_i)$ the functions $q_B(l)$ and $p_B(l)$ are computed in $\mathrm{O}(1)$ time.*

Proof. The algorithm consists of the following steps:

1. Find all the blocks of N_i. Let $B_1 \ldots, B_s$ be the blocks of N_i and let the corresponding vertex sets be $V(B_1), \ldots, V(B_s)$. Note that for every $j \in \{1, \ldots, s\}$ we have $V(B_j) \subseteq V_i$.
2. The leaves of N_i are relabeled as follows. A leaf receives the label i where $i \in \{1, 2, \ldots, n\}$, if it is the $i - th$ leaf in order that is discovered by a depth first traversal of N_i. This traversal starts from $r(N_i)$. Let u be a vertex in N_i and part of the blocks $B_1, \ldots, B_j$. Let B' be the

block from $B_1, \ldots, B_j$, such that $r(B')$ has the largest height among all roots of $B_1, \ldots, B_j$. During the traversal, every child u' of u that is not part of B' is visited first. This is to ensure that the labels in $L^u_{B'}$ are consecutive and defined by a range of numbers $[u_{left}, u_{right}]$.

3. For every $j \in \{1, \ldots, s\}$ the process of building $C_{B_j} = (V_j, E_j)$ is initialized as follows. Set $V_j = V(B_j)$. For every edge $u \to v$ in E_i, if both u and v are in V_j then add that edge to E_j. Finally, for any vertex u_1 in V_j, if $L^{u_1}_{B_j} \neq \emptyset$ create the leaf s_1 representing $L^{u_1}_{B_j}$, the copy leaf s'_1, add the edges $u_1 \to s_1$ and $u_1 \to s'_1$ to E_j, and set $Q_{B_j}[l] = u_1$ for every $l \in \{u_{left}, \ldots, u_{right}\}$.
4. For every $j \in \{1, \ldots, s\}$ the edges of C_{B_j} are contracted, following the definition of a contracted block network. While performing the contraction, for every $j \in \{1, \ldots, s\}$ we build the table $P_{B_j}[1, \ldots, n]$, defined so that for every $l \in \{1, \ldots, n\}$ we have $P_{B_j}[l] = p_{B_j}(l)$. The value of $P_{B_j}[l]$ is updated once the final set in which the leaf l will reside in is determined. After contracting all the edges we also add the artificial leaf r'_j.

To analyze the running time, step 1 is performed by using the algorithm from [41], thus requiring $O(|E_i|)$ time. Step 2 of the algorithm is performed by a depth first traversal of N_i, thus requiring $O(|E_i|)$ time as well. Since the blocks of N_i are edge disjoint (see Lemma 8), we have $\sum_{j=1}^{s} |E_j| \leq |E_i|$, thus the time spent on adding and contracting vertices and edges in steps 3 and 4 is $O(|E_i|)$ time. For every block network C_B, we spend $O(n)$ time to update the Q and P tables. From Corollary 3 there can be $O(n)$ such block networks, thus the time required to update every Q and P table is $O(n^2)$. Hence, the total running time of the algorithm is $O(|E_i| + n^2)$. □

For the block network C_B, we denote C^f_B the fan graph of C_B and C^r_B the resolved graph of C_B. Moreover, we denote A^f_B the fan table of C_B and A^r_B the resolved table of C_B (see Section 3.2.1 for the definition of a fan graph & table and resolved graph & table). The following lemma shows the time required to build C^f_B, C^r_B, A^f_B, and A^r_B for every block B of a given network N_i.

Lemma 13. *For a network N_i, building C^f_B, C^r_B, A^f_B, and A^r_B for every block B of N_i requires $O(n(k_i^3 d_i^3 + 1))$ time.*

Proof. From Lemma 11 and when $k_i > 0$, for every block network $C_B = (V', E')$ we have $|V| = O(k_i d_i)$ and $|E| = O(k_i d_i)$. By the definition of a fan and resolved graph in Section 3.2, C^f_B and C^r_B contain $O(k_i^3 d_i^3)$ vertices and edges. Since $|\mathcal{L}(C_B)| = O(k_i d_i)$ the size of A^f_B and A^r_B is $O(k_i^3 d_i^3)$. When $k_i = 0$ we have $|V| = |E| = O(1)$, thus C^f_B and C^r_B contain $O(1)$ vertices and edges and the size of A^f_B and A^r_B is $O(1)$. From Corollary 3, N_i has $O(n)$ blocks. Hence the total time required to build C^f_B, C^r_B, A^f_B, and A^r_B for every block B of N_i is $O(n(k_i^3 d_i^3 + 1))$ for any $k_i \geq 0$. □

3.3.2 Triplet Distance Computation

Let B be a block of a network N_i. We say that $x|y|z$ is a *fan triplet consistent with* B, if and only if there exists a vertex u in B such that there are three directed paths in N_i from u to x, from u to y, and from u to z that are disjoint except for in the vertex u. We also say that $x|y|z$ *is rooted at* u in B. Since u is also in N_i, this means that $x|y|z$ is rooted at u in N_i as well. Similarly, we say that $xy|z$ is a *resolved triplet consistent with* B, if and only if there exist two vertices u and v ($u \neq v$) in B, such that there are four directed paths in N_i from u to v, from v to x, from v to y, and from u to z that are disjoint except for in the vertices u and v, and furthermore, the path from u to z does not pass through v. Moreover, we say that $xy|z$ is *rooted at* u *and* v in B or N_i (similarly to the fan triplet). Note that if $x|y|z$ is a fan triplet consistent with B, then it will also be consistent with N_i. Similarly, if $xy|z$ is a resolved triplet consistent with B, it will also be consistent with N_i.

Given the data structures from the preprocessing step, Lemmas 14, 15, 16, and 17 together show how to determine the consistency of a fan and resolved triplet with N_i in O(1) time. From a high level perspective to achieve this, for three different leaves x, y, and z, we consider all the possible cases for the location of the lowest common ancestor of every pair (x, y), (x, z), and (y, z) in T_i. Since every vertex in T_i except $r(T_i)$ corresponds to a block in N_i, we can then use the available data structures to determine efficiently if N_i has the necessary disjoint paths that would imply the consistency of the fan triplet $x|y|z$ or resolved triplet $xy|z$ with N_i. We start by showing in Lemmas 14 and 15 how to determine the consistency of a fan and resolved triplet with a block B of N_i. Afterwards, we show in Lemma 16 how to use Lemma 14 to determine the consistency of a fan triplet with N_i. Similarly, we show in Lemma 17 how to use Lemma 15 to determine the consistency of a resolved triplet with N_i.

Lemma 14. *Let N_i be a network, T_i its block tree, and suppose that the preprocessing of Lemma 13 has been applied on N_i. Consider any $x, y, z \in \Lambda$ with the lowest common ancestor of every pair (x, y), (x, z), and (y, z) being a node w in T_i. If $w \neq r(T_i)$, Algorithm 7 determines whether or not the fan triplet $x|y|z$ is consistent with the block B in N_i corresponding to w in* O(1) *time.*

Proof. For every $l \in \{x, y, z\}$ we let $p_l = p_B(l)$, $p'_l = p'_B(l)$, $q_l = q_B(l)$, and h_l be the height of q_l in N_i. By construction (see Lemmas 10 and 12), we know that p_x, p_y, and p_z are not the root of C_B. The algorithm uses the tables Q and P to check all the possible cases for the values of p_x, p_y, p_z, q_x, q_y, and q_z, and return a true or false value, indicating a positive and a negative answer respectively. We have the following cases:

1. $p_x = p_y = p_z$:
   a) $h_x = h_y = h_z$: We have $q_x = q_y = q_z$ and $x|y|z$ is rooted at q_x. Hence, $x|y|z$ is consistent with B (e.g., $a_5|a_6|a_7$ in Fig. 3.4).

   b) $((h_x = h_y) \wedge (h_x > h_z)) \vee ((h_x = h_z) \wedge (h_x > h_y)) \vee ((h_y = h_z) \wedge (h_y > h_x))$. W.l.o.g. assume truth for $(h_x = h_y \wedge h_x > h_z)$: Then, $(q_x = q_y) \wedge (q_x \neq q_z)$ and $x|y|z$ is rooted at q_x. Hence, $x|y|z$ is consistent with B (e.g., $a_5|a_6|a_8$ in Fig. 3.4).
   c) $h_x \neq h_y \neq h_z$: Then $q_x \neq q_y \neq q_z$, thus $x|y|z$ is not consistent with B (e.g., $a_{13}|a_{14}|a_{20}$ in Fig. 3.4).
2. $((p_x = p_y) \wedge (p_x \neq p_z)) \vee ((p_x = p_z) \wedge (p_x \neq p_y)) \vee ((p_y = p_z) \wedge (p_y \neq p_x))$. W.l.o.g. assume truth for $(p_x = p_y \wedge p_x \neq p_z)$:
   a) $h_x = h_y$: We have $q_x = q_y$. If $p'_x|p_x|p_z$ is a fan triplet in C_B, then $x|y|z$ is rooted at q_x, thus $x|y|z$ is consistent with B (e.g., $a_8|a_9|a_{15}$ in Fig. 3.4). If $p'_x|p_x|p_z$ is not a fan triplet in C_B, $x|y|z$ is not rooted at any vertex in B, thus $x|y|z$ is not consistent with B (e.g., $a_8|a_9|a_{11}$ in Fig. 3.4).
   b) $h_x \neq h_y$: Then $q_x \neq q_y$ and either q_x or q_y was contracted when creating C_B. Moreover, both x and y are now in the set of leaves defined by p_x. Since we also have that $p_z \neq p_x$, the triplet $x|y|z$ is not consistent with B (e.g., $a_7|a_8|a_{15}$ in Fig. 3.4).
3. $p_x \neq p_y \neq p_z$: If $p_x|p_y|p_z$ is consistent with C_B, then there exists a vertex u in B such that $x|y|z$ is rooted at u. Hence, $x|y|z$ is consistent with B (e.g., $a_8|a_{11}|a_{16}$ in Fig. 3.4). If $p_x|p_y|p_z$ is not consistent with C_B, $x|y|z$ is not rooted at any vertex in B, thus $x|y|z$ is not consistent with B (e.g., $a_{14}|a_{16}|a_{17}$ in Fig. 3.4).

Note that in every case above, testing if a fan triplet is consistent with C_B translates to finding a path that starts from s in C_B^f and ends in a vertex of C_B^f defined by the leaves of the fan triplet. Hence, every case can be handled in O(1) time. In Algorithm 7 in Appendix B we have the above cases summarized in a procedure. □

Lemma 15. *Let N_i be a network, T_i its block tree, and suppose that the preprocessing of Lemma 13 has been applied on N_i. Consider any $x, y, z \in \Lambda$ with the lowest common ancestor of every pair (x, y), (x, z), and (y, z) being a node w in T_i. If $w \neq r(T_i)$, Algorithm 8 determines whether or not the resolved triplet $xy|z$ is consistent with the block B in N_i corresponding to w in* O(1) *time.*

Proof. Similarly to the case of fan triplets in Lemma 14, for every $l \in \{x, y, z\}$ we let $p_l = p_B(l)$, $p'_l = p'_B(l)$, $q_l = q_B(l)$, and h_l be the height of q_l in N_i. By construction (see Lemmas 10 and 12), we know that p_x, p_y, and p_z are not the root of C_B. The algorithm uses the tables Q and P to check all the possible cases for the values of p_x, p_y, p_z, q_x, q_y, and q_z, and return a true or false value, indicating a positive and a negative answer respectively. We have the following cases:

1. $p_x = p_y = p_z$:
   a) $(h_z > h_x) \wedge (h_z > h_y)$. W.l.o.g. let $h_x \geq h_y$: Then, $xy|z$ is rooted at q_z and q_x, thus $xy|z$ is a resolved triplet in B (e.g., $a_8a_9|a_6$ in

Fig. 3.4).

b) $(h_z \leq h_x) \vee (h_z \leq h_y)$: Because $p_x = p_y = p_z$, $xy|z$ is not rooted at any pair of vertices in B, thus $xy|z$ is not consistent with B (e.g., $a_8a_6|a_9$ in Fig. 3.4).

2. $(p_x = p_y) \wedge (p_x \neq p_z)$ and w.l.o.g. assume $h_x \geq h_y$: If $p'_xp_x|p_z$ is consistent with C_B, there exists $u \neq q_x$ in B such that $xy|z$ is rooted at u and q_x in B. Hence, $xy|z$ is consistent with B (e.g., $a_5a_8|a_{17}$ in Fig. 3.4). If $p'_xp_x|p_z$ is not consistent with C_B, $xy|z$ is not rooted at any pair of vertices in B, thus $xy|z$ is not consistent with B (e.g., $a_5a_8|a_{15}$ in Fig. 3.4).
3. $((p_x = p_z) \wedge (p_x \neq p_y)) \vee ((p_y = p_z) \wedge (p_y \neq p_x))$. W.l.o.g. assume that $(p_x = p_z) \wedge (p_x \neq p_y)$:
   a) $h_z > h_x$: If $p'_x|p_x|p_y$ is a fan triplet in C_B, then $xy|z$ is rooted at q_z and q_x, thus $xy|z$ is consistent with B (e.g., $a_{14}a_{17}|a_{13}$ in Fig. 3.4). If $p'_x|p_x|p_y$ is not consistent with C_B, $xy|z$ is not rooted at any pair of vertices in B, thus $xy|z$ is not consistent with B (e.g., $a_{14}a_{16}|a_{13}$ in Fig. 3.4.).
   b) $h_z \leq h_x$: Since $p_x = p_z$, the resolved triplet $xy|z$ cannot be consistent with B (e.g., $a_{14}a_{17}|a_{20}$ in Fig. 3.4).
4. $p_x \neq p_y \neq p_z$: If $p_xp_y|p_z$ is consistent with C_B, then there exist two different vertices u, v in B such that $xy|z$ is rooted at u and v, thus $xy|z$ is consistent with B (e.g., $a_{12}a_{13}|a_{18}$ in Fig. 3.4). If $p_xp_y|p_z$ is not consistent with C_B, $xy|z$ is not rooted at any pair of vertices in B, thus $xy|z$ is not consistent with B (e.g., $a_{12}a_{18}|a_{13}$ in Fig. 3.4).

Similarly to fan triplets, testing if a resolved triplet is consistent with C_B translates to finding a path that starts from s in C_B^r and ends in a vertex of C_B^r defined by the leaves of the resolved triplet. Hence, every case can be handled in O(1) time. In Algorithm 8 in Appendix B we have the above cases summarized in a procedure. □

Lemma 16. *Let N_i be a given network and T_i its block tree, and suppose that the preprocessing from Lemma 13 has been performed on N_i. For any $x, y, z \in \Lambda$, Algorithm 9 determines whether or not the fan triplet $x|y|z$ is consistent with N_i in* O(1) *time.*

Proof. For a block B of N_i and a vertex u in B that can reach a leaf x of N_i, define $h_B(x)$ to be the height of $q_B(x)$ in N_i. In Algorithm 9 in Appendix B we have the procedure for testing the consistency of the fan triplet $x|y|z$. We consider the following cases:

1. $x|y|z$ is consistent with T_i: Let w be the lowest common ancestor of x, y, and z in T_i.
   a) $w = r(T_i)$: $x|y|z$ is rooted at $r(N_i)$, thus $x|y|z$ is consistent with N_i (e.g., $a_{23}|a_9|a_{20}$ in Fig. 3.5).
   b) $w \neq r(T_i)$: w corresponds to a block B in N_i, thus we use Lemma 14 to determine if $x|y|z$ is consistent with B. If $x|y|z$ is consistent

with B, then it is also consistent with N_i. If $x|y|z$ is not consistent with B, then it is not consistent with N_i (e.g., $a_3|a_9|a_{12}$ in Fig. 3.5).

2. $xy|z \vee xz|y \vee yz|x$ is consistent with T_i. Assume w.l.o.g. that $xy|z$ is consistent with T_i. Let $w = lca(x, y)$ in T_i and $\mu = lca(x, z)$ in T_i, and let B be the block in N_i corresponding to w and F the block in N_i corresponding to μ:
   a) μ is not the parent of w in T_i: then $x|y|z$ is not rooted at any vertex in N_i, thus $x|y|z$ is not consistent with N_i (e.g., $a_2|a_4|a_{13}$ in Fig. 3.5).
   b) μ is the parent of w in T_i. By the definition of T_i, B is rooted at a vertex u of F that is not $r(F)$:
      i. $p_B(x) = p_B(y)$: then $x|y|z$ is not rooted at any vertex in N_i, thus $x|y|z$ is not consistent with N_i (e.g., $a_2|a_3|a_4$ in Fig. 3.5).
      ii. $(p_B(x) \neq p_B(y)) \wedge (\mu = r(T_i))$: If $r'|p_B(x)|p_B(y)$ is consistent with C_B, where r' is the dummy leaf in C_B (see Corollary 1), then $x|y|z$ is rooted at $r(N_i)$, thus $x|y|z$ is consistent with N_i (e.g., $a_1|a_{11}|a_{15}$ in Fig. 3.5). Otherwise, $x|y|z$ is not rooted at any vertex in N_i, thus $x|y|z$ is not consistent with N_i (e.g., $a_{11}|a_{13}|a_{15}$ in Fig. 3.5).
      iii. $(p_B(x) \neq p_B(y)) \wedge (\mu \neq r(T_i))$:
         A. $(p_F(x) = p_F(z)) \wedge (h_F(z) \leq h_F(x))$: Since B is rooted at a vertex of F, we have $q_F(x) = q_F(y)$, thus $h_F(x) = h_F(y)$. Using Corollary 1, if $r'|p_B(x)|p_B(y)$ is a fan triplet in C_B, where r' is the dummy leaf in C_B, then $x|y|z$ is rooted at $q_F(x)$, thus $x|y|z$ is a fan triplet in N_i (e.g., $a_1|a_4|a_8$ in Fig. 3.5). Otherwise, $x|y|z$ is not rooted at any vertex in N_i, thus $x|y|z$ is not consistent with N_i (e.g., $a_1|a_{24}|a_8$ in Fig. 3.5).
         B. $(p_F(x) = p_F(z)) \wedge (h_F(z) > h_F(x))$: Since B is rooted at a vertex of F, we have $q_F(x) = q_F(y)$ and $h_F(x) = h_F(y)$. Hence, $x|y|z$ is not consistent with N_i (e.g., $a_1|a_4|a_{21}$ in Fig. 3.5).
         C. $p_F(x) \neq p_F(z)$: Using Corollary 1, if $r'|p_B(x)|p_B(y)$ is a fan triplet in C_B, where r' is the dummy leaf in C_B, and $p_F(x)|p'_F(x)|p_F(z)$ is a fan triplet in C_F, then $x|y|z$ is rooted at $q_F(x)$. Hence, $x|y|z$ is consistent with N_i (e.g., $a_1|a_4|a_9$ in Fig. 3.5). Otherwise, $x|y|z$ is not rooted at any vertex of N_i, thus $x|y|z$ is not consistent with N_i (e.g., $a_1|a_4|a_{12}$ in Fig. 3.5).

□

Lemma 17. *Let N_i be a given network and T_i its block tree, and suppose that the preprocessing from Lemma 13 has been performed on N_i. For any $x, y, z \in \Lambda$, Algorithm 10 determines whether or not the resolved triplet $xy|z$*

is consistent with N_i in $O(1)$ time.

Proof. For a block B of N_i and a vertex u in B that can reach a leaf x of N_i, define $h_B(x)$ to be the height of $q_B(x)$ in N_i. In Algorithm 10 in Appendix B we have the procedure for testing the consistency of the resolved triplet $xy|z$. We consider the following cases which are very similar to the cases for the fan triplet in Lemma 16:

1. $x|y|z$ is consistent with T_i: Let w be the lowest common ancestor of x, y, and z in T_i.
   a) $w = r(T_i)$: $xy|z$ is not rooted at any pair of vertices in N_i, thus $xy|z$ is not consistent with N_i (e.g., $a_{23}a_9|a_{20}$ in Fig. 3.5).
   b) $w \neq r(T_i)$: w corresponds to a block B in N_i, thus we use Lemma 15 to determine if $xy|z$ is consistent with B. If $xy|z$ is consistent with B, then it is also consistent with N_i. If $xy|z$ is not consistent with B, then it is not consistent with N_i (e.g., $a_1a_9|a_{12}$ in Fig. 3.5).
2. $xy|z \vee xz|y \vee yz|x$ is consistent with T_i. Assume w.l.o.g. that $xy|z$ is consistent with T_i. Let $w = lca(x, y)$ in T_i and $\mu = lca(x, z)$ in T_i, and let B be the block in N_i corresponding to w and F the block in N_i corresponding to μ:
   a) μ is not the parent of w in T_i: then there exists a vertex u in B and a vertex v in F such that $xy|z$ is rooted at v and u, thus $xy|z$ is consistent with N_i (e.g., $a_2a_4|a_{13}$ in Fig. 3.5).
   b) μ is the parent of w in T_i. By the definition of T_i, B is rooted at a vertex u of F that is not $r(F)$. We consider the following cases:
      i. $p_B(x) = p_B(y)$: w.l.o.g. assume $h_B(x) > h_B(y)$. Then, $xy|z$ is rooted at either $r(B)$ and $q_B(x)$, or $q_F(z)$ and $q_B(x)$, or $r(F)$ and $q_B(x)$. Hence, $xy|z$ is consistent with N_i (e.g., $a_2a_3|a_4$ in Fig. 3.5).
      ii. $(p_B(x) \neq p_B(y)) \wedge (\mu = r(T_i))$: Using Corollary 2, if we have that $p_B(x)p_B(y)|r'$ is consistent with C_B, where r' is the dummy leaf in C_B, then there exists a vertex u in B such that $xy|z$ is rooted at $r(N_i)$ and u. Hence, $xy|z$ is consistent with N_i (e.g., $a_{11}a_{13}|a_{15}$ in Fig. 3.5). Otherwise, $xy|z$ is not rooted at any pair of vertices in N_i, thus $xy|z$ is not consistent with N_i (e.g., $a_1a_{11}|a_{15}$ in Fig. 3.5).
      iii. $(p_B(x) \neq p_B(y)) \wedge (\mu \neq r(T_i))$:
         A. $(p_F(x) = p_F(z)) \wedge (h_F(z) \leq h_F(x))$: Since B is rooted at a vertex of F, we have $q_F(x) = q_F(y)$, thus $h_F(x) = h_F(y)$. Using Corollary 2, if $p_B(x)p_B(y)|r'$ is consistent with C_B, where r' is the dummy leaf in C_B, then there exists a vertex u in B such that $xy|z$ is rooted at $q_F(x)$ and u. Hence, $xy|z$ is consistent with N_i (e.g., $a_1a_4|a_8$ in Fig. 3.5). Otherwise, $xy|z$ is not rooted at any pair of vertices in N_i, thus $xy|z$ is not consistent with N_i (e.g., $a_1a_{25}|a_{22}$ in

Fig. 3.5).

B. $(p_F(x) = p_F(z)) \wedge (h_F(z) > h_F(x))$: Since B is rooted at a vertex of F, we have $q_F(x) = q_F(y)$ and $h_F(x) = h_F(y)$. Then, there exists a vertex u in block B such that $xy|z$ is rooted at $q_F(z)$ and u, thus $xy|z$ is consistent with N_i (e.g., $a_1a_4|a_{21}$ in Fig. 3.5).

C. $p_F(x) \neq p_F(z)$: Using Corollary 2, if $p_B(x)p_B(y)|r'$ is consistent with C_B, where r' is the dummy leaf in C_B, then there exists a vertex u in B such that $xy|z$ is rooted at either $r(B)$ and u, or $q_F(z)$ and u, or $r(F)$ and u. If $p_F(x)p'_F(x)|p_F(z)$ is consistent with C_F, then w.l.o.g. if $h_F(x) > h_F(y)$ we have that $xy|z$ is rooted at some vertex u of F and $q_F(x)$. In both cases, $xy|z$ is consistent with N_i (e.g., $a_1a_4|a_{12}$ in Fig. 3.5). If both cases are false, $xy|z$ is not rooted at any pair of vertices in N_i, thus it is not consistent with N_i (e.g., $a_1a_{25}|a_{26}$ in Fig. 3.5).

□

Algorithm 11 in Appendix B includes all the procedures needed to compute the triplet distance between two given networks N_1 and N_2. It is similar to Algorithm 2, the main difference is in the preprocessing step. In this step (lines 3-9), for every $i \in \{1, 2\}$ we start by building the block tree T_i. Then, we build a $n \times n$ table for T_i in order to be able later to answer lowest common ancestor queries between pairs of leaves in T_i in O(1) time. Afterwards, we build all the contracted block networks of N_i. Finally, for every block B in N_i and the corresponding contracted block network C_B, we build the fan graph C_B^f and the resolved graph C_B^r, as well as the corresponding A_B^f and A_B^r tables.

From Lemma 10, building T_i for every $i \in \{1, 2\}$ takes $O(|E_1| + |E_2|)$ time. Building the two tables for answering lowest common ancestor queries requires $O(n^3)$ time. From Lemma 11, building all the contracted block networks requires $O(|E_1| + |E_2| + n^2)$ time. From Lemma 13, the time required to build C_B^f, C_B^r, A_B^f, and A_B^r for every block B of N_1 and N_2 is $O(n(k_1^3d_1^3 + k_2^3d_2^3 + 2))$. Then, the total preprocessing time becomes $O(|E_1|+|E_2|+n(k_1^3d_1^3+k_2^3d_2^3)+n^3)$.

Using the results from Lemmas 16 and 17, after the preprocessing step we can determine the consistency of a triplet with N_1 or N_2 in O(1) time. Since the number of triplets that need to be checked is exactly $4\binom{n}{3}$, the total running time of the algorithm remains $O(|E_1| + |E_2| + n(k_1^3d_1^3 + k_2^3d_2^3) + n^3)$. By the definition of N, M, k, and d from Section 3.1, the running time becomes $O(M + k^3d^3n + n^3)$. Hence, we obtain the following theorem:

Theorem 3. *There exists an algorithm that computes the triplet distance between between two networks N_1 and N_2 in $O(M + k^3d^3n + n^3)$ time.*

Let N_i be a network that follows the degree constraints of Byrka *et al.* [15]. If for a block $B_s = (V_s, E_s)$ of N_i we define k_s to be the number of reticulation

vertices in B_s, where $k_s \leq k_i$, using the same arguments as those used in the proof of Lemma 11, we get for $C_{B_s} = (V', E')$ that $|V'| = |E'| = \mathrm{O}(k_s + 1)$. The time to build C_B^f, C_B^r, A_B^f, and A_B^r for every block B of N_i then becomes $\sum_s \mathrm{O}(k_s^3 + 1)$. Note that Lemma 13 would give a $\mathrm{O}(nk_i^3 + 1)$ time instead, because it uses n to upper bound (from Corollary 3) the number of blocks we can have in N_i. Since $\sum_s k_s = \mathrm{O}(|V_i|)$, the preprocessing time required by our algorithm for N_i would be $\mathrm{O}(|V_i| + k^2|V_i|)$. Then, the time to compute $D(N_1, N_2)$ becomes $\mathrm{O}(N + k^2N + n^3)$, thus matching the time bound required by using the second data structure of Byrka *et al.* [15].

3.4 Implementation and Experiments

This section provides an implementation of the algorithms described in Sections 3.2 and 3.3, as well as experiments illustrating their practical performance. From here on, the algorithm from Section 3.2 is referred to as `NTDfirst` and the algorithm from Section 3.3 as `NTDsecond`.

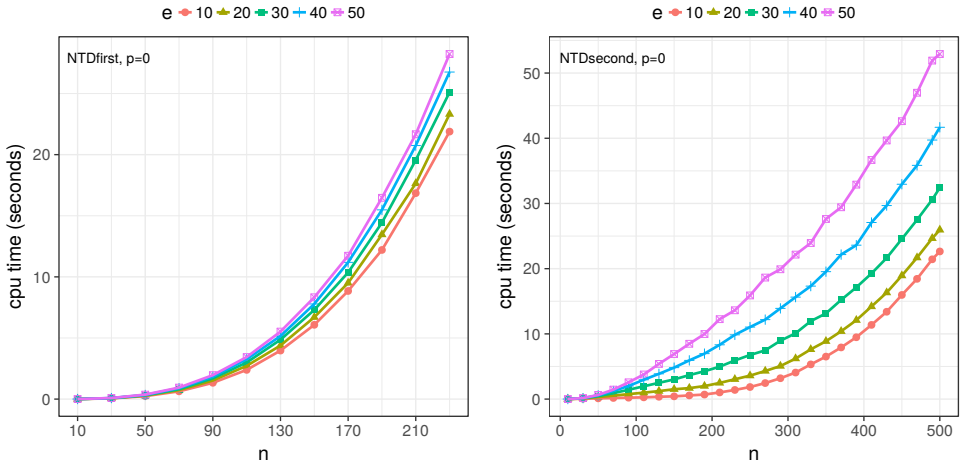
The Setup. We implemented the two algorithms in C++ and the source code is publicly available at `https://github.com/kmampent/ntd`. The experiments were performed on a machine with 16GB RAM and Intel(R) Core(TM) i5-3470 CPU @ 3.20GHz. The operating system was Ubuntu 16.04.2 LTS. The compiler used was g++ 5.4 with cmake 3.11.0.

Correctness. Since no other implementation is available for computing the rooted triplet distance between two networks of arbitrary level, correctness was ensured by comparing the output of `NTDfirst`, which is simple to implement, and the output of `NTDsecond`, on a large number of pairs of input networks under varying parameters.

The Input. We consider both simulated and real datasets. For the simulated datasets, we create tree-based networks [30] by inserting edges at random locations in a tree. From here on, *random* implies *uniformly at random*. More precisely, an input network N' is built as follows:

1. Build a random rooted binary tree T on n leaves in the uniform model [59].
2. Let $N' = T$. Given a parameter $0 \leq p \leq 1$, contract every internal vertex of N' except $r(N')$ with probability p. For a vertex w in N', let $d(w)$ be the total number of edges on the path from $r(N')$ to w in N'.
3. Given a parameter $e \geq 0$, add e random edges in N' as follows. An edge $u \to v$ is created in N' if $d(u) < d(v)$. If the total number of edges that can be added happens to be y, where $y < e$, then we only add those y edges.

For the real datasets, we consider networks that have been published in the literature, which are not necessarily tree-based. More precisely, we consider the 6 trees and the corresponding networks in [58, Table S4]. For the sixth network in [58, Table S4], we use the non-tree based version given in [54,



(a) (b)

Figure 3.7: Simulated Datasets: Running time of the two algorithms `NTDfirst` and `NTDsecond` for different values of e and when $p = 0$.

s	$S(T_s, T_s)$	$S(N_s, N_s)$	$S(T_s, N_s)$	$D(T_s, N_s)$
A	560	716	443	390
B	1140	1870	840	1330
C	1330	2185	965	1585
D	1330	2205	964	1607
E	1540	1996	983	1570
F	19600	43710	16553	30204

	N_A	N_B	N_C	N_D	N_E
N_A	0	20	19	20	10
N_B	20	0	1	0	10
N_C	19	1	0	1	9
N_D	20	0	1	0	10
N_E	10	10	9	10	0

Table 3.2: Experiments on the real datasets. The networks $N_A, \ldots, N_E$ have identical leaf label sets. The leaf label set of N_F is different, which is why N_F is not included in the table on the right. Note that $D(N_B, N_D) = 0$.

Fig. 19]. The trees, named T_A, T_B, T_C, T_D, T_E, and T_F below, are given in the standard Newick format, and the networks, named N_A, N_B, N_C, N_D, N_E, and N_F below, in the extended Newick format [17]. Note that we use the graph-theoretic standard *adjacency list* to store the input networks, making it easy to support different input formats at the same time.

Experiments. For the simulated datasets, every data point in the graphs corresponds to the average of 30 different runs under the same set of parameters. To make the experiments more realistic, because reticulation events are considered to be rare [10], we tried relatively small values for e, i.e., $e \leq 50$ when $n \leq 500$.

In Fig. 3.7 we have the CPU time in seconds required by the two algorithms when $p = 0$ and $e \in \{10, 20, 30, 40, 50\}$. For `NTDfirst` we have $10 \leq n \leq 230$ and for `NTDsecond` we have $10 \leq n \leq 500$. Space is the reason behind the

different restrictions on n. More precisely, as seen in Fig. B.1, for $n = 230$ the memory usage of `NTDfirst` approaches the limits of the available 16GB RAM. When $n \geq 240$, the memory requirements exceed these limits, so the operating system initiates the highly time consuming communication with the disk. In Fig. 3.7, we observe the effect of e on the running time of the two algorithms. The main purpose of extending the algorithm from Section 3.2 in Section 3.3, was to avoid building the highly time and space consuming fan and resolved graph on the entire input network and instead, build several such graphs on the smaller blocks of the network. As shown in Fig. B.3, larger values for e imply both fewer non-leaf blocks in N' and a larger value for k, which in turn implies more time spent by `NTDsecond` building the fan and resolved graphs. A bad scenario is when e is so large that N' has a very small number of non-leaf blocks, making them approximately as large as N' itself. Then, given that the preprocessing of `NTDsecond` is more complicated than that of `NTDfirst`, the performance of `NTDsecond` will be worse than `NTDfirst`. An example of such a scenario can be found in Fig. B.2(a), where for $p = 0$, $n = 110$, and $e = 50$, the time performance of `NTDfirst` is better than that of `NTDsecond`. When p is large, e.g., $p = 0.8$ in Fig. B.2(b), the effect of e on the running time is decreased because the number of internal vertices in the networks becomes small, thus the number of possible edges we can add (defined by e) becomes small as well. More information on the effect of p on the running times of the two algorithms can be found in Fig. B.4.

For the real datasets and for every $s \in \{A, B, C, D, E, F\}$, we denote by T_s the tree and N_s the corresponding network, where s is a scenario in [58, Table S4], with F corresponding to scenario "`E, CHAM and MELVIO resolved`". For the network N_F, we use its non-tree based version from [54, Fig. 19]. N_B is shown in Fig. 3.8(a) and Fig. 3.9(a), and N_D in Fig. 3.8(b) and Fig. 3.9(b). For the other four networks' branching structures, see [58]. From Equation 3.1 we have the following: $D(T_s, N_s) = S(T_s, T_s) + S(N_s, N_s) - 2S(T_s, N_s)$. When computing $D(T_s, N_s)$ and to have $\mathcal{L}(T_s) = \mathcal{L}(N_s)$, if a leaf x in N_s appears as several leaves $x.1, \ldots, x.i$ in T_s, we replace x in N_s with the leaves $x.1, \ldots, x.i$ that we attach under the parent of x. For the size of the leaf label sets, in T_A, T_B, T_C, T_D, and T_E we have 16, 20, 21, 21, 22, and 50 leaves, in every network N_s where $s \in \{A, B, C, D, E\}$ we have 8 leaves and in N_F we have 16 leaves.

In Table 3.2 we include the experimental results. On the left table we compute for every $s \in \{A, B, C, D, E, F\}$, the values of $S(T_s, T_s)$, $S(N_s, N_s)$, $S(T_s, N_s)$, and $D(T_s, N_s)$. On the right table we compute for every $s, s' \in \{A, B, C, D, E\}$ the triplet distance $D(N_s, N_{s'})$. The maximum time spent was when computing $D(T_F, N_F)$, with `NTDfirst` requiring only 0.18 seconds to run and `NTDsecond` 0.05 seconds. Interestingly, while the two networks N_B and N_D look structurally different, their triplet distance is 0. This suggests that alternative definitions may be useful in practice, as discussed in the concluding remarks below.

3.5 Concluding Remarks

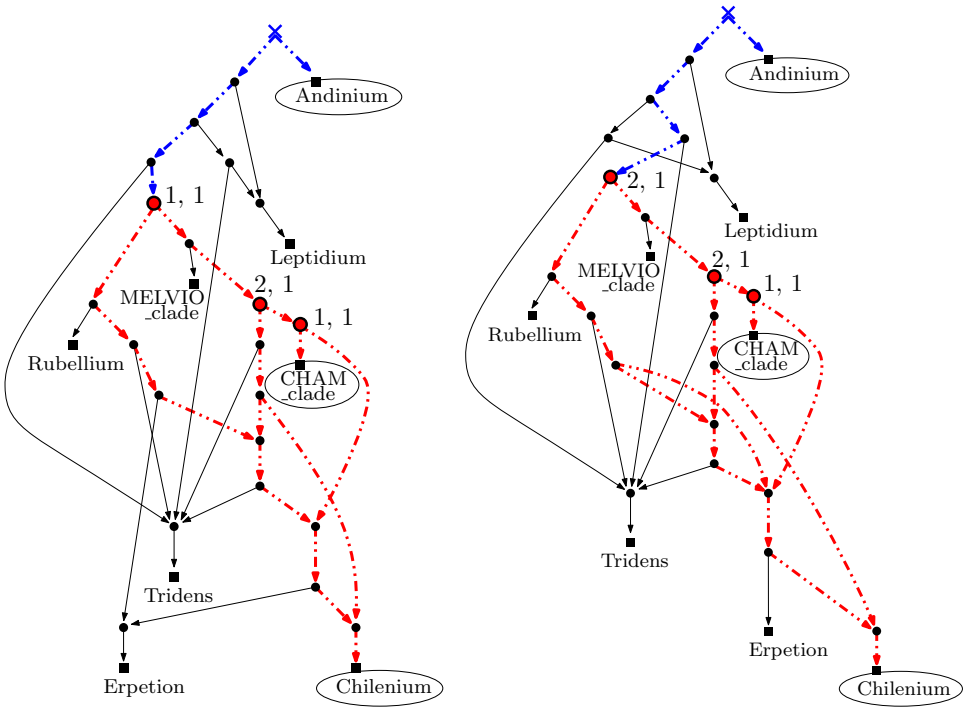
In this paper, we presented two new algorithms for computing the rooted triplet distance between two phylogenetic networks that are built on the same leaf label set. We provided an implementation of the algorithms, as well as extensive experiments illustrating their performance on both simulated and real datasets.

Future work involves creating new algorithms that are more efficient than the algorithms presented in this paper, as well as research variants of the problem studied in this paper that might provide more biologically meaningful ways for comparing networks. An example of such a variant is motivated by the experiments on the real dataset, where the two networks N_B and N_D are structurally different but their triplet distance is 0. Unlike in the case of trees, a triplet can appear several times in a network. For two networks N_1 and N_2 that we wish to compare, if a triplet appears 1000 times in N_1 and only once in N_2, it means that the structures of the two networks are different, which is not captured by the current definition of $D(N_1, N_2)$. Should the definition of the triplet distance for networks be extended to capture the information about the frequency in which triplets appear in the networks, and if so, could the algorithms presented in this paper be extended to support this new definition? For the number of occurrences of a triplet in a network, it is crucial to find a biologically meaningful definition, as different definitions can lead to different outcomes. For example, consider the following two definitions of multiplicity for a resolved triplet $xy|z$, where u and v are the vertices used in the definition of the consistency of a resolved triplet with a network in Section 3.1.

A. total number of quadruples of paths of the form $((u \rightsquigarrow v), (v \rightsquigarrow x), (v \rightsquigarrow y), (u \rightsquigarrow z))$ that are disjoint except for in u and v, and furthermore, the path from u to z does not pass through v.
B. total number of pairs of vertices (u, v) such that there exist paths of the form $((u \rightsquigarrow v), (v \rightsquigarrow x), (v \rightsquigarrow y), (u \rightsquigarrow z))$ that are disjoint except for in u and v, and furthermore, the path from u to z does not pass through v.

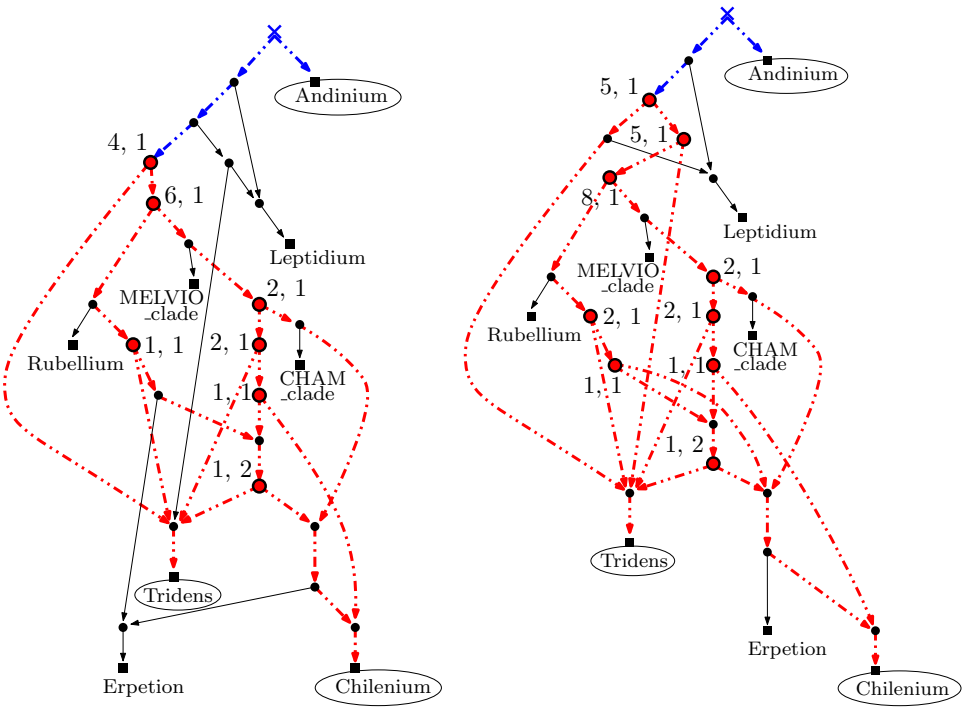
The definitions for the case of fan triplets are analogous. In Fig. 3.8 we draw the two networks N_B and N_D. If we follow definition A of multiplicity, the resolved triplet `Chilenium CHAM_clade | Andinium` appears 4 times in N_B and 5 times in N_D. On the other hand, if we follow definition B, this resolved triplet appears three times in both networks. In Fig. 3.9 we have the same two networks but now consider the resolved triplet `Tridens Chilenium | Andinium`. With definition A of multiplicity, this triplet appears 18 times in N_B and 28 times in N_D, and with definition B it appears 7 times in N_B and 9 times in N_D.

Finally, Cardona *et al.* [18] gave an alternative generalization of the rooted triplet distance from trees to networks. While the extension proposed by Gambette and Huber [33] is closer to the definition of the extensively studied tree triplet distance, efficient algorithms for this extension could be useful to create as well.



(a) The network N_B. (b) The network N_D.

Figure 3.8: The two networks N_B and N_D from the real datasets with $D(N_B, N_D) = 0$. On top of every vertex illustrated with a circle we have the number of different pairs of disjoint paths from the vertex to the leaves with labels `Chilenium` and `CHAM_clade`, and the number of different disjoint paths to the vertex from the root of the corresponding network. With definition A of multiplicity for resolved triplets described in Section 2.7, the resolved triplet `Chilenium CHAM_clade` | `Andinium` appears $(1+2+1)\cdot 1 = 4$ times in N_B and $(2+2+1)\cdot 1 = 5$ times in N_D. With definition B, this triplet appears three times in both networks.



(a) The network N_B. (b) The network N_D.

Figure 3.9: The two networks N_B and N_D from the real datasets with $D(N_B, N_D) = 0$. On top of every vertex illustrated with a circle we have the number of different pairs of disjoint paths from the vertex to the leaves with labels `Tridens` and `Chilenium`, and the number of different disjoint paths to the vertex from the root of the corresponding network. With definition A of multiplicity for resolved triplets described in Section 2.7, the resolved triplet `Tridens Chilenium` | `Andinium` appears $(4{+}6{+}2{+}1{+}2{+}1){\cdot}1{+}1{\cdot}2 = 18$ times in N_B and $(5+5+8+2+2+2+1+1)\cdot 1+1\cdot 2 = 28$ times in N_D. With definition B, this triplet appears 7 times in N_B and 9 times in N_D.

Chapter 4

Building a Small and Informative Phylogenetic Supertree

[51] Jesper Jansson, Konstantinos Mampentzidis, and Sandhya Thekkumpadan Puthiyaveedu. Building a Small and Informative Phylogenetic Supertree. In *19th International Workshop on Algorithms in Bioinformatics (WABI 2019)*, volume 143 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. 2019.

We combine two fundamental, previously studied optimization problems related to the construction of phylogenetic trees called *maximum rooted triplets consistency* (MAXRTC) and *minimally resolved supertree* (MINRS) into a new problem, which we call *q-maximum rooted triplets consistency* (q-MAXRTC). The input to our new problem is a set $\mathcal{R}$ of resolved triplets (rooted, binary phylogenetic trees with three leaves each) and the objective is to find a phylogenetic tree with exactly q internal nodes that contains the largest possible number of triplets from $\mathcal{R}$. We first prove that q-MAXRTC is NP-hard even to approximate within a constant ratio for every fixed $q \geq 2$, and then develop various polynomial-time approximation algorithms for different values of q. Next, we show experimentally that representing a phylogenetic tree by one having much fewer nodes typically does not destroy too much triplet branching information. As an extreme example, we show that allowing only nine internal nodes is still sufficient to capture on average 80% of the rooted triplets from some recently published trees, each having between 760 and 3081 internal nodes. Finally, to demonstrate the algorithmic advantage of using trees with few internal nodes, we propose a new algorithm for computing the *rooted triplet distance* between two phylogenetic trees over a leaf label set of size n that runs in $\mathrm{O}(qn)$ time, where q is the number of internal nodes in the smaller tree, and is therefore faster than the currently best algorithms for the problem (with $\mathrm{O}(n \log n)$ time complexity [SODA 2013, ESA 2017]) whenever $q = o(\log n)$.

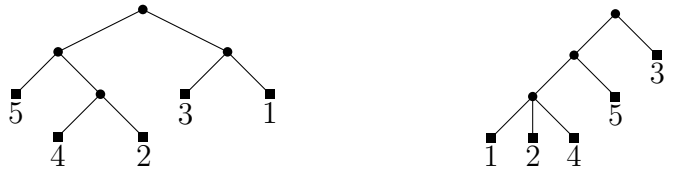


Figure 4.1: Let $L = \{1, 2, 3, 4, 5\}$ and $\mathcal{R} = \{45|3, 25|3, 13|5, 24|5, 23|1\}$. In this example no tree T such that $|\mathcal{R} \cap rt(T)| = 5$ exists. Left figure: optimal solution for MAXRTC with value 4. Right figure: optimal solution for 3-MAXRTC with value 3.

4.1 Introduction

Background. Phylogenetic trees are used in biology to represent evolutionary relationships [27]. The leaves in such a tree correspond to species that exist today and internal nodes to ancestor species that existed in the past. An important problem when studying the evolution of species is, given some data describing the species, to construct a phylogenetic tree that supports the input data as much as possible. The supertree approach [9, 19, 39] deals with the challenging problem of constructing a reliable phylogenetic tree for a large set of species by combining several accurate trees for small, overlapping subsets of the species into one final tree. Depending on the type of data that is available and the type of trees that we want to construct, we obtain several variants of the same problem. Phylogenetic trees are also used in linguistics to deduce and visualize evolutionary relationships between natural languages [61].

Problem Definition. A *rooted phylogenetic tree* is a rooted unordered tree in which every leaf has a distinct label and every internal node has at least two children. In this article, for simplicity we use the word "tree" to refer to a "rooted phylogenetic tree". A *resolved triplet* is a binary tree with three leaves. The resolved triplet with leaf labels x, y, and z where z is closest to the root is denoted by $xy|z$. From now on when referring to a "triplet" we mean a "resolved triplet". Let T be a tree on a leaf label set L of size n. For a node $u \in T$, $deg(u)$ is the number of children of u and $T(u)$ is the subtree induced by u and all the proper descendants of u. For two nodes u and v in T, $lca(u, v)$ is the lowest common ancestor node of u and v in T. We say that the triplet $xy|z$ is *induced by* T if $lca(x, z) = lca(y, z)$ and $lca(x, y) \neq lca(x, z)$. Let $rt(T)$ be the set of all triplets induced by T. Given a set $\mathcal{R}$ of triplets, we say that $\mathcal{R}$ *is consistent with* T, or equivalently T *is consistent with* $\mathcal{R}$, if $\mathcal{R} \subseteq rt(T)$.

Given a set $\mathcal{R}$ of triplets on a leaf label set L of size n, in the *q-maximum rooted triplets consistency problem*, denoted q-MAXRTC, the goal is to find a tree T with exactly q internal nodes such that $|\mathcal{R} \cap rt(T)|$ is maximized, i.e., the total number of triplets induced by T that are also in $\mathcal{R}$ is as large as possible. An example can be seen in Figure 4.1.

Let A be an algorithm for any maximization problem. Given an input instance I, let $opt(I)$ be the value of an optimal solution and $A(I)$ the value of the solution returned by A. Let $0 \leq r \leq 1$. We say that A is an r-approximation algorithm with *relative ratio* r, if $A(I) \geq r \cdot opt(I)$ for any I. Similarly, A is an r-approximation algorithm with *absolute ratio* r, if $A(I) \geq r \cdot |I|$ for any I. In particular, for q-MAXRTC we have that $A(I) \geq r \cdot |\mathcal{R}|$. From here on and unless otherwise stated, when we refer to any ratio r, we imply an absolute ratio.

Previous Work. Aho *et al.* [2] proposed a polynomial-time algorithm, called BUILD, that can determine if there exists a tree inducing all triplets from an input $\mathcal{R}$, and if such a tree exists, output it. As observed by Bryant [13], the BUILD algorithm does not always produce a tree with the minimal number of internal nodes. In fact, BUILD might return a tree with $\Omega(n)$ more internal nodes than needed [48], which is undesirable because unnecessary internal nodes may suggest false groupings of the leaves, also known as *spurious novel clades* [9]. Moreover, scientists typically look for the simplest possible explanation for some given observations and would prefer a tree that makes as few additional statements as possible about evolutionary relationships that are not supported by the input data. This motivates the *minimally resolved phylogenetic supertree* (MINRS) problem, where the output is a tree (if one exists) inducing all triplets from $\mathcal{R}$ while having the minimum number of internal nodes. The decision version of MINRS is NP-complete for $q \geq 4$ and polynomial-time solvable for $q \leq 3$ [48], where q is the total number of internal nodes in the output tree. An exact exponential-time algorithm for MINRS and experimental results for the non-optimality of BUILD for MINRS were given in [50]. For the special case of *caterpillar trees* (trees in which every internal node has at most one non-leaf child), MINRS is polynomial-time solvable [48].

The above problems only consider finding trees that induce all triplets from $\mathcal{R}$. However, in situations where such a tree cannot be constructed, e.g., due to a single error in the input triplets, it is still useful to build a tree that induces as many of the triplets from $\mathcal{R}$ as possible. This has been formalized as the *maximum rooted triplets consistency problem* (MAXRTC). Bryant [13] showed that MAXRTC is NP-hard and Gąsieniec *et al.* [34] proposed a polynomial-time top-down $\frac{1}{3}$-approximation algorithm that always returns a caterpillar tree. Byrka *et al.* [16] showed that a bottom-up algorithm by Wu [77] can be modified to also obtain a polynomial-time $\frac{1}{3}$-approximation algorithm. In [15], Byrka *et al.* gave a polynomial-time $\frac{1}{3}$-approximation algorithm by derandomizing a randomized algorithm. In Section 3.4 below, we refer to the algorithm in [34] as *One-Leaf-Split* (`OLS`) and the algorithm in [16] as *the modified Wu algorithm* (`Mod-Wu`). For more results related to the computational complexity of MAXRTC, see [16].

Motivation. The existing approximation algorithms for MAXRTC typically produce trees with an arbitrary number of internal nodes. For example, the algorithms in [15, 16] always produce a tree with $n - 1$ internal nodes

Table 4.1: Previous and new results for computing q-MAXRTC. The abbreviations "abs." and "rel." correspond to "absolute" and "relative" respectively.

Year	Reference	Deterministic	q	Approximation	Type
1999	Gąsieniec *et al.* [34]	yes	unbounded	1/3	abs.
2010	Byrka *et al.* [15, 16]	yes	$n-1$	1/3	abs.
2019	new [Section 4.3.1]	no	2	1/2	rel.
2019	new [Section 4.3.1]	yes	2	1/4	rel.
2019	new [Theorem 5]	yes	2	4/27	abs.
2019	new [Theorem 7]	yes	$q \geq 3$	$\frac{1}{3} - \frac{4}{3(q+q \bmod 2)^2}$	abs.

and the algorithm in [34], $n-1$ for certain $\mathcal{R}$. However, due to the issue of spurious novel clades [9] mentioned above, biologists may prefer to build a supertree with few internal nodes that is still consistent with a large number of input triplets, which leads to the new problem q-MAXRTC introduced in this paper. More precisely, q-MAXRTC can be regarded as a combination of MINRS and MAXRTC that models how well the triplet branching information contained in the set of input triplets can be preserved while forcing the size of the output tree to be bounded by a user-specified parameter q. Note that by the problem definition, some input branching structure typically has to be discarded and the objective is to do it in a least destructive way.

On a high level, q-MAXRTC is comparable to the problem of compressing a large data file into a small data file. As an analogy, consider the widely used JPEG compression method for images. Both JPEG and q-MAXRTC are examples of *lossy compression* where the user controls a parameter yielding a trade-off between the size of the compressed data (the number of bits for JPEG and the number of internal nodes for q-MAXRTC) and the amount of preserved information (the image quality for JPEG and the number of induced triplets in $\mathcal{R}$ for q-MAXRTC).

Finally, in the design of phylogenetic tree comparison algorithms, trees with fewer internal nodes sometimes admit faster running times. For example, given two trees built on the same leaf label set of size n, the fastest known algorithms for computing the so-called *rooted triplet distance* between the two trees takes $O(n \log n)$ time [11, 12], but if at least one of the input trees has $O(1)$ internal nodes then the time complexity can be reduced to $O(n)$; see Section 4.5. As the available published trees get larger and larger (the total number of species on Earth was recently estimated to be 1 trillion [57]), to make their comparison practical, it may become necessary to approximate them using trees with fewer internal nodes while keeping enough triplet branching structure to represent them accurately.

New Results and Outline of the Article. In Section 4.2 we show that q-MAXRTC is NP-hard for every fixed $q \geq 2$ and give some inapproximability results. Section 4.3 describes our new approximation algorithms. In Section

4.4 we provide implementations, and present some experimental results showing that representing a tree by one having much fewer nodes typically does not destroy too much triplet branching information. As an extreme example, we show that allowing only nine internal nodes is still sufficient to capture on average 80% of the rooted triplets from some recently published trees, each having between 760 and 3081 internal nodes. Section 4.5 presents our new O(qn)-time algorithm, as well as provides an implementation of it, for computing the rooted triplet distance between two trees, where n is the size of the leaf label set in the two trees and q the number of internal nodes in the smaller tree. Finally, Section 4.6 contains some concluding remarks and open problems. For a summary of previous and new results related to q-MAXRTC, refer to Table 4.1.

4.2 Computational Complexity of q-MAXRTC

In this section, we study the computational complexity of q-MAXRTC. We first establish the NP-hardness of q-MAXRTC, and then present some inapproximability results.

Theorem 4. *q-MAXRTC is NP-hard for every fixed $q \geq 2$.*

Proof. We consider the known NP-hard problem MAX q-CUT [55], in which the input is an undirected graph $G = (V, E)$ and the goal is to find a partition $(A_1, A_2, \ldots, A_q)$ of V such that the total number of edges connecting two nodes residing in different sets, i.e., *the size of the cut*, is maximized. We prove that q-MAXRTC is NP-hard by reducing MAX q-CUT to q-MAXRTC as follows: let $L = V \cup \{z\}$ and $\mathcal{R} = \{xz|y, yz|x : \{x, y\} \in E\}$. We claim that there exists a cut $(A_1, A_2, \ldots, A_q)$ of size k in G if and only if there exists a solution to q-MAXRTC that is consistent with k triplets from $\mathcal{R}$. We now prove the claim.

First, assume that there exists a cut $(A_1, A_2, \ldots, A_q)$ of size k in G. We construct a tree T that is rooted at the vertex a_1 with additional internal nodes $a_2, \ldots, a_q$ such that a_{i+1} is a child of a_i for $1 \leq i \leq q-1$. For $i \in \{1, 2, \ldots, q\}$, we attach $|A_i|$ leaves bijectively labeled by A_i as children of a_i, and the vertex z is added as a child of a_q. Consider any edge $\{x, y\}$ in the cut. By the definition of a cut, $x \in A_i$ and $y \in A_j$ for two different $i, j \in \{1, 2, \ldots, q\}$. If $i < j$, then $yz|x$ will be consistent with T, since $lca(y, z) = a_j$ is a proper descendant of $lca(x, y) = lca(x, z) = a_i$. Similarly, if $i > j$, then $xz|y$ will be consistent with T. For every edge in the cut, exactly one triplet will be consistent with T, so T will be consistent with exactly k triplets from $\mathcal{R}$.

Conversely, assume that there exists a tree T that has q internal nodes $a_1, a_2, \ldots, a_q$ and is consistent with k triplets from $\mathcal{R}$. Define $A_i = \{x : x \text{ is a child of } a_i\} \setminus \{z, a_1, a_2, \ldots, a_q\}$ for $1 \leq i \leq q$. Define $S = \mathcal{R} \cap rt(T)$. For each $xz|y \in S$, clearly x and y belong to different sets A_i and A_j for some

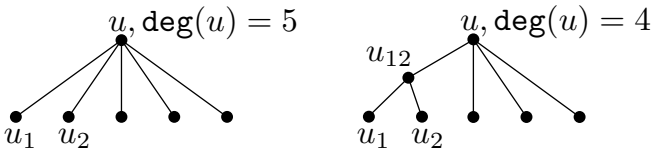


Figure 4.2: Increasing the number of internal nodes by one without destroying any triplets.

$i, j \in \{1, 2, \ldots, q\}$, and thus the corresponding edge $\{x, y\}$ contributes one to the size of the cut, making the size of the cut $|S| = k$. □

From the inapproximability of MAXCUT [37], we obtain the following:

Corollary 4. *Unless P=NP, 2-MAXRTC cannot be approximated in polynomial time within a relative ratio of* $16/17 + \varepsilon$*, for any constant* $\varepsilon > 0$.

From the inapproximability of MAX q-CUT [55], we obtain the following:

Corollary 5. *Unless P=NP, for any* $q \geq 3$*, it holds that q-MAXRTC cannot be approximated in polynomial time within a relative ratio of* $1 - 1/(34q) + \varepsilon$*, for any constant* $\varepsilon > 0$.

Remark 1. *Recall from Section 3.1 that MINRS is polynomial-time solvable if restricted to caterpillar trees [48]. In contrast, the proof of Theorem 4 shows that q-MAXRTC remains NP-hard even in this special case.*

4.3 Approximability of q-MAXRTC

Intuitively, a tree with a larger number of internal nodes should be able to induce more triplets from a given $\mathcal{R}$. The next lemma shows that this is indeed so, and upper bounds the total number of triplets that can be induced. Define $opt(q)$ to be the maximum number of triplets that can be consistent with a tree T with q internal nodes.

Lemma 18. *Let* $2 \leq q' \leq q \leq n - 1$. *We then have* $opt(q') \leq opt(q) \leq \lceil \frac{q-1}{q'-1} \rceil opt(q')$.

Proof. We start by showing that $opt(q') \leq opt(q)$. Let T' be the tree with q' internal nodes that induces $opt(q')$ triplets from $\mathcal{R}$. We can create a tree T with q internal nodes that induces at least as many triplets from $\mathcal{R}$ as follows. Let $T = T'$. While T does not have q internal nodes, let $u \in T$ such that $deg(u) > 2$ and u_1, u_2 be two children of u. Create an internal node u_{12}, make u_1 and u_2 the children of u_{12} and u_{12} the child of u. An example can be found in Figure 4.2.

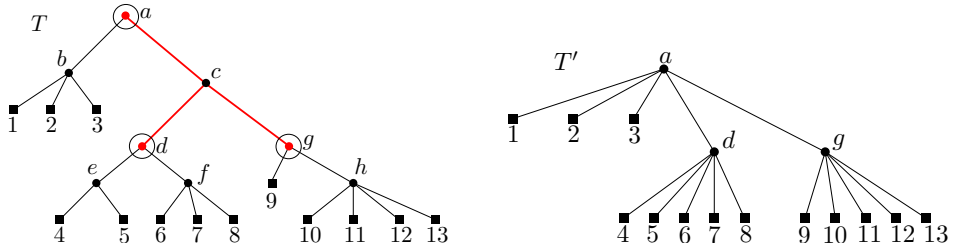


Figure 4.3: An example. Let T be the tree on the left with 9 internal nodes. The tree T' on the right with 3 internal nodes is created by deleting all internal nodes in T except $W = \{a, d, g\}$.

To show the second half of the inequality, proceed as follows. Define the *delete operation* on any non-root node u in a tree as the operation of making the children of u become children of the parent of u, and then removing u and all edges incident to u. Let T be the tree that induces $opt(q)$ triplets from $\mathcal{R}$. Let $t = ab|c$ be a triplet induced in T that is also in $\mathcal{R}$. Anchor t in $lca(a, b)$. Let $W = \{u_1, u_2, \ldots, u_{q'}\}$ be any set of q' internal nodes in T such that the root of T is included in W. Create a tree T' with q' internal nodes by letting T' be a copy of T and applying the delete operation to every internal node of T' not in W. Note that for a node u in T such that $u \in W$, every triplet anchored in u will also be induced by T'. An example can be found in Figure 4.3.

Let $T'_1, T'_2, \ldots, T'_\lambda$ be trees that are built like T', but in a way such that every internal node $u \in T$ except $r(T)$, corresponds to an internal node of exactly one such tree. Observe that $\lambda = \lceil \frac{q-1}{q'-1} \rceil$.We can create these trees with the following procedure:

- Store all internal nodes of T except $r(T)$ in the ordered set S, in any order from left to right and set $j = 1$.
- If $|S| \geq q' - 1$, pick and remove from S the first $q' - 1$ internal nodes to define W, and construct T'_j. Otherwise, pick the remaining nodes in S to define W and create T'_j just like T' but with $|S| = |W|$ nodes instead of $q' - 1$. Set $j = j + 1$.
- if $|S| = 0$ stop. Otherwise go to the previous step.

We then have: $|rt(T) \cap \mathcal{R}| = opt(q) \leq \sum_{i=1}^{\lambda} |rt(T'_i) \cap \mathcal{R}| \leq \lambda opt(q') = \lceil \frac{q-1}{q'-1} \rceil opt(q')$. □

4.3.1 Polynomial-time Approximation Algorithms Based on MAX 3-CSP

MAX 3-AND is a Boolean satisfiability problem in which we are given as input a logical formula consisting of a set of clauses, each being a conjunction (AND) of three literals formed from a set of Boolean variables, and the goal is to assign

each Boolean variable a *True*/*False*-value so that the total number of satisfied clauses is maximized. Both MAX 3-AND and the well-known MAX 3-SAT problem are special cases of the MAX 3-CSP problem [78], where a clause can be an arbitrary function over three literals. The following lemma shows that 2-MAXRTC can be reduced to MAX 3-AND in polynomial time while preserving the approximation ratio.

Lemma 19. *If MAX* 3*-AND can be approximated within a factor of* r*, then* 2*-MAXRTC can also be approximated within a factor of* r*.*

Proof. We present a reduction from 2-MAXRTC to MAX 3-AND. Given a set of triplets $\mathcal{R}$ built on the leaf label set L, we construct an instance of MAX 3-AND as follows: let $V = L$ be the set of variables and $C = \{x \wedge y \wedge \bar{z} : xy|z \in \mathcal{R}\}$ the set of clauses. We claim that there exists a solution to MAX 3-AND that satisfies k clauses from C if and only if there exists a solution to 2-MAXRTC that is consistent with k triplets from $\mathcal{R}$. We now prove this claim.

First, suppose that there exists an assignment ϕ on V that satisfies a set S of clauses from C, where $|S| = k$. Let $A = \{z : x \wedge y \wedge \bar{z} \in S\}$ and $B = \{x, y : x \wedge y \wedge \bar{z} \in S\}$. Observe that $A \cap B = \emptyset$, which can be argued as follows. Let $c \in A \cap B$. Then $c \in A$ implies that $c =$ *False* in ϕ and $c \in B$ implies that $c =$ *True* in ϕ, leading to a contradiction. Next, we construct a tree T with root a and an internal node b that is a child of a that is consistent with k triplets from $\mathcal{R}$. The elements in A and B are added as the children of a and b respectively. The set of all k triplets of the form $\{xy|z : x \wedge y \wedge \bar{z} \in S\}$ is consistent with T, since $lca(x, y) = b$ and $lca(x, z) = lca(y, z) = a$.

Conversely, assume that T is a tree that is consistent with the set S' of k triplets from $\mathcal{R}$. Let a be the root of T and b the other internal node of T that is the child of a. Let $B = \{x : x \text{ is a child of } b\}$ and $A = \{x : x \text{ is a child of } a\} \setminus \{b\}$. We set $x =$ *True*, for every $x \in B$ and $x =$ *False*, for every $x \in A$. For every $x, y \in A$ and $z \in B$, the corresponding clause $x \wedge y \wedge \bar{z}$ is satisfied. Hence, all clauses corresponding to the triplets in S' are satisfied. □

Lemma 19 allows every approximation algorithm for MAX 3-AND to be used to approximate 2-MAXRTC. For MAX 3-AND, Zwick [78] presented a randomized polynomial-time $\frac{1}{2}$-approximation algorithm with relative ratio based on semi definite programming. Trevisan [74] presented a deterministic polynomial-time $\frac{1}{4}$-approximation algorithm with relative ratio based on linear programming. A deterministic polynomial-time algorithm based on local search by Alimonti [3], would satisfy $\geq \frac{1}{8}|C|$ number of clauses, giving a $\frac{1}{8}$-approximation ratio for 2-MAXRTC. Since this ratio is absolute, from Lemma 3 this algorithm also gives a $\frac{1}{8}$-approximation ratio for q-MAXRTC, where $q \geq 3$.

4.3.2 Polynomial-time Approximation Algorithms Based on Derandomization

Reducing 2-MAXRTC to MAX 3-AND can produce a polynomial-time deterministic $\frac{1}{8}$-approximation algorithm for q-MAXRTC, however from Lemma 3, we should be able to capture more triplets by allowing more internal nodes. The algorithms based on MAX 3-AND cannot be directly extended to support Lemma 3. We propose a new deterministic polynomial-time algorithm for q-MAXRTC that achieves a $\frac{4}{27}$-approximation ratio, based on a randomized algorithm for 2-MAXRTC, and then show how to extend it to get better approximation ratios for higher values of q. Note that the only available related algorithm based on derandomization by Byrka *et al.* [15], always constructs a binary tree on n leaves, i.e., the case $q < n - 1$ is not considered. Moreover, as we will show below, our derandomization procedure is highly optimized for trees instead of the more complex *phylogenetic networks* (for a definition, see Section 2 of [15]).

Lemma 20. *There exists a randomized polynomial-time $\frac{4}{27}$-approximation algorithm for q-MAXRTC.*

Proof. Let $\mathcal{R} = \{r_1, \ldots, r_{|\mathcal{R}|}\}$ be the set of triplets and $L = \{x_1, \ldots, x_n\}$ the leaf label set. Build a tree T with two internal nodes, with a being the root and b the child of the root. Make every leaf $x_i \in L$ with probability $\frac{2}{3}$ a child of b and probability $\frac{1}{3}$ a child of a. Let Y_j be a random variable that is 1 if $r_j \in rt(T)$ and 0 otherwise. Let $W = \sum_{j=1}^{|\mathcal{R}|} Y_j$. For the expected number of triplets consistent with T we have $E[W] = \sum_{j=1}^{|\mathcal{R}|} E[Y_j] = \sum_{j=1}^{|\mathcal{R}|} \frac{4}{27} = \frac{4}{27}|\mathcal{R}|$. □

Theorem 5. *There exists a deterministic polynomial-time $\frac{4}{27}$-approximation algorithm for q-MAXRTC that runs in $\mathrm{O}(|\mathcal{R}|)$ time.*

Proof. We derandomize the algorithm in Lemma 20 with the method of conditional expectations [76] in a way that differs from Byrka *et al.* [15], where the main focus is the general case of phylogenetic networks. In our method, the leaves $x_1, \ldots, x_n$ are scanned from left to right, and each leaf is deterministically assigned to either be the child of b, denoted $x_i \leftarrow b$, or the child of a, denoted $x_i \leftarrow a$. The leaves are assigned in a way, such that after every assignment the expected value of the solution is preserved. From probability theory we have $E[W] = \frac{1}{3}E[W|x_1 \leftarrow a] + \frac{2}{3}E[W|x_1 \leftarrow b]$. We then choose $n_1 = a$ or $n_1 = b$ such that $E[W|x_1 \leftarrow n_1] = \max(E[W|x_1 \leftarrow a], E[W|x_1 \leftarrow b])$. This gives $E[W|x_1 \leftarrow n_1] \geq E[W] = \frac{4}{27}|R|$. Suppose that the first i leaves have been assigned to $n_1, \ldots, n_i$. Let N_i contain those assignments, i.e., we have that $N_i = \{x_1 \leftarrow n_1, \ldots, x_i \leftarrow n_i\}$. To find the assignment for x_{i+1} we follow the same approach as that for x_1 as follows. We have $E[W|N_i] = \frac{1}{3}E[W|N_i, x_{i+1} \leftarrow a] + \frac{2}{3}E[W|N_i, x_{i+1} \leftarrow b]$ and then n_{i+1} is chosen so that

Algorithm 3 O($n|\mathcal{R}|$)-time $\frac{4}{27}$-approximation algorithm for q-MAXRTC based on 2-MAXRTC (Theorem 5)

```
1: procedure PR2(xy|z)                          ▷ Computing Pr[xy|z ∈ rt(T)|N_i]
2:     if x ← a or y ← a or z ← b then return 0
3:     p = 4/27
4:     if x ≠ ∅ and  x ← b then p = 3p/2 ▷ x ≠ ∅, thus x has been assigned
5:     if y ≠ ∅ and  y ← b then p = 3p/2
6:     if z ≠ ∅ and  z ← a then p = 3p
7:     return p
8: procedure 2-MAXRTC-SLOW(R)                   ▷ The main procedure
9:     for i = 1 to n do
10:        aValue = 0                           ▷ To compute E[W|N_{i-1}, x_i ← a]
11:        bValue = 0                           ▷ To compute E[W|N_{i-1}, x_i ← b]
12:        for j = 1 to |R| do
13:            x_i ← a
14:            aValue = aValue + PR2(R[j])
15:            x_i ← b
16:            bValue = bValue + PR2(R[j])
17:        x_i ← b
18:        if aValue > bValue then x_i ← a
```

$E[W|N_{i+1}] = \max(E[W|N_i, x_{i+1} \leftarrow a], E[W|N_i, x_{i+1} \leftarrow b])$. By induction, we then get that $E[W|N_{i+1}] \geq E[W|N_i] \geq \cdots \geq \frac{4}{27}|\mathcal{R}|$.

To compute $E[W|N_i]$, we use the fact that $E[W|N_i] = \sum_{j=1}^{|\mathcal{R}|} Pr[r_j \in rt(T)|N_i]$, where $Pr[r_j \in rt(T)|N_i]$ can be computed in O(1) time (see the procedure PR2() of Algorithm 3). A trivial implementation that scans all the leaves and for every possible assignment of a leaf x_i, computes the expected value $E[W|N_i]$ by scanning the entire set $\mathcal{R}$ (see the procedure 2-MAXRTC-SLOW() of Algorithm 3) would require O($n|\mathcal{R}|$) time.

We can achieve a more efficient implementation (see the procedure 2-MAXRTC-FAST() of Algorithm 4) that would require O($|\mathcal{R}|$) time, by maintaining for every leaf $x_i \in L$, a list of all the triplets that x_i is part of, denoted $\mathcal{R}[x_i]$. At the beginning of the i-th iteration of the first for loop in Algorithm 4, the value of the variable *prev* is $E[W|N_{i-1}]$. To determine the assignment for leaf x_i, we need to compute $E[W|N_{i-1}, x_i \leftarrow a]$ and $E[W|N_{i-1}, x_i \leftarrow b]$, and for this we use the second for loop. At the end of the execution of the second for loop, we have that the value of $E[W|N_{i-1}, x_i \leftarrow a]$ will be stored in the variable *aValue* and the value of $E[W|N_{i-1}, x_i \leftarrow b]$ in the variable *bValue*. To compute *aValue* (resp. *bValue*), we initialize it to the value of *prev*, and then for every triplet in the list $\mathcal{R}[x_i]$, we subtract the contribution of that triplet to the value of *prev* when $x_i \leftarrow \emptyset$, and add its new

Algorithm 4 O($|\mathcal{R}|$) $\frac{4}{27}$-approximation algorithm for q-MAXRTC based on 2-MAXRTC

```
1: procedure PR2(xy|z)                          ▷ Computing Pr[xy|z ∈ rt(T)|N_i]
2:     if x ← a or y ← a or z ← b then return 0
3:     p = 4/27
4:     if x ≠ ∅ and x ← b then p = 3p/2 ▷ x ≠ ∅ meaning that x has been assigned
5:     if y ≠ ∅ and y ← b then p = 3p/2
6:     if z ≠ ∅ and z ← a then p = 3p
7:     return p

8: procedure 2-MAXRTC-FAST(R)                   ▷ The main procedure
9:     prev = 4|R|/27                           ▷ Storing E[W|N_0], where N_0 = ∅
10:    for i = 1 to n do
11:        aValue = prev                        ▷ To compute E[W|N_{i-1}, x_i ← a]
12:        bValue = prev                        ▷ To compute E[W|N_{i-1}, x_i ← b]
13:        for j = 1 to |R[x_i]| do
14:            x_i ← ∅
15:            aValue = aValue − PR2(R[x_i][j])
16:            bValue = bValue − PR2(R[x_i][j])
17:            x_i ← a
18:            aValue = aValue + PR2(R[x_i][j])
19:            x_i ← b
20:            bValue = bValue + PR2(R[x_i][j])
21:        x_i ← b
22:        prev = bValue
23:        if aValue > bValue then
24:            x_i ← a
25:            prev = aValue
```

contribution by having $x_i \leftarrow a$ (resp. $x_i \leftarrow b$). Since every triplet in $\mathcal{R}$ will be part of 3 lists, every triplet will induce O(1) calls to the procedure PR2() of Algorithm 4, giving the O($|\mathcal{R}|$) final bound of the algorithm. □

In the following theorem, we prove that the best possible absolute approximation ratio for 2-MAXRTC is $\frac{4}{27}$, making the approximation algorithm in Theorem 5 optimal when considering algorithms with absolute approximation ratios.

Theorem 6. *For any $\varepsilon > 0$, there exists some n and set $\mathcal{R}$ of triplets on a leaf label set of size n, such that the approximation ratio $\geq \frac{4}{27} + \varepsilon$ for 2-MAXRTC is impossible.*

Proof. For any n, let $L_n = \{1, 2, \ldots, n\}$ and $\mathcal{R}_n = \{ab|c, ac|b, bc|a : a, b, c \in L, |\{a, b, c\}| = 3\}$. Since $|L_n| = n$, we have $|\mathcal{R}_n| = 3\binom{n}{3}$. Next, we construct a tree T with two internal nodes, which is rooted at the vertex a with an internal node b (b is a child of a). Let $A = \{x : x \text{ is a child of } a\} \setminus \{b\}$ and $B = \{x : x \text{ is a child of } b\}$. Assume that $m = |A|$. Then $|B| = n - m$ and $|rt(T) \cap \mathcal{R}_n| = m(\frac{m-1}{2})(n-m)$. By taking derivatives, we obtain that T is consistent with the largest number of triplets when $m = \frac{n+1+\sqrt{n^2-n+1}}{3}$. For that given m, we then have $|rt(T) \cap \mathcal{R}_n| = \big(\frac{n+1+\sqrt{n^2-n+1}}{3}\big)\big(\frac{n-2+\sqrt{n^2-n+1}}{6}\big)\big(\frac{2n-1-\sqrt{n^2-n+1}}{3}\big)$ and $\lim\limits_{n\to\infty} \frac{|rt(T)\cap\mathcal{R}_n|}{|\mathcal{R}_n|} = \frac{4}{27}$. □

To obtain an algorithm (see Algorithm 12 in Appendix C) with a better approximation ratio for $q \geq 3$, we allow the output tree T to have q internal nodes $\{u_1, \ldots, u_q\}$. Every internal node $u_j \in T$ has a probability $p(u_j)$, which is the probability of a fixed leaf being assigned to that node. Given that $\sum_{j=1}^{q} p(u_j) = 1$, we can obtain a randomized algorithm, the analysis of which follows from Lemma 20. Let $E[W]$ be the expected value of that randomized algorithm. Like in Theorem 5, we can derandomize the algorithm to obtain a $\frac{E[W]}{|\mathcal{R}|}$-approximation ratio. The only difference in the proof is that the total number of possible assignments is q instead of 2, i.e., given N_i, we choose n_{i+1} for x_{i+1} such that $E[W|N_i, x_{i+1} \leftarrow n_{i+1}] = \max(E[W|N_i, x_{i+1} \leftarrow u_1, \ldots, E[W|N_i, x_{i+1} \leftarrow u_q)$. The problem is thus reduced to finding a tree with q internal nodes and a choice of probabilities $p(u_1), \ldots, p(u_q)$ such that $E[W] > \frac{4}{27}|\mathcal{R}|$.

Theorem 7. *Given $q \geq 3$, there exists a randomized polynomial-time algorithm for q-MAXRTC that achieves a $(\frac{1}{3} - \frac{4}{3(q+q \bmod 2)^2})$-approximation. The algorithm can be derandomized while preserving the approximation ratio. The running time of the deterministic algorithm is $\mathrm{O}(q|\mathcal{R}|)$.*

Proof. Let $\mathcal{R} = \{r_1, \ldots, r_{|\mathcal{R}|}\}$ and $L = \{x_1, \ldots, x_n\}$. If $q = 2k + 1$ for some $k \in \mathbb{N}$, build a binary tree T with q nodes in total. By construction, T has $k+1$ leaves and k internal nodes. Assign the probability $1/(k+1)$ to every leaf and 0 to every internal node. For the expected value of the randomized algorithm we then obtain:

$$E[W] = \sum_{j=1}^{|\mathcal{R}|} Pr[r_j \in rt(T)] = \sum_{j=1}^{|\mathcal{R}|} \frac{k(k+1) + 2\binom{k+1}{3}}{(k+1)^3} = \Big(\frac{1}{3} - \frac{1}{3(k+1)^2}\Big)|\mathcal{R}| \,.$$

By setting $k = \frac{q-1}{2}$, we get $E[W] = (\frac{1}{3} - \frac{4}{3(q+1)^2})|\mathcal{R}|$. If $q = 2k$, we follow the same approach but by building a binary tree T with $2k - 1$ nodes instead. The tree T has k leaves and $k - 1$ internal nodes. Every leaf has the probability $1/k$ and every internal node the probability 0. After finding the assignment of the leaves according to the chosen probabilities, we pick one

internal node u for which $deg(u) \geq 3$ to add a new internal node as shown in Figure 4.2. This would give $E[W] \geq (\frac{1}{3} - \frac{4}{3q^2})|\mathcal{R}|$. Hence, for any $q \geq 3$ we have $E[W] \geq (\frac{1}{3} - \frac{4}{3(q+q \bmod 2)^2})|\mathcal{R}|$.

To obtain a deterministic algorithm that preserves the approximation ratios, we use the method of conditional expectations. We now describe how to achieve the $O(q|\mathcal{R}|)$ time bound, given any binary tree T with q internal nodes and a probability assignment as described above. For any internal node u in T, let $d(u)$ be the number of edges on the path from the root of T to u. Define the following counters:

- $u_{\downarrow}$ = total number of leaves in $T(u)$.
- $u_{\uparrow s}$ = total number of pairs of leaves x, y such that $x \neq y$, x and y are not in $T(u)$, and $d(lca(u,x)) = d(lca(u,y))$.
- $u_{\uparrow d}$ = total number of pairs of leaves x, y such that $x \neq y$, x and y are not in $T(u)$, and $d(lca(u,x)) < d(lca(u,y))$.

After building T, all these counters can be trivially computed in $O(q)$ time by applying two depth first search traversals on T. The procedure PRQ() of Algorithm 12 shows how to compute $Pr[xy|z \in rt(T)|N_i]$. The most expensive operation is computing the lowest common ancestor of two nodes. However, data structures exist [7], that can answer such queries in $O(1)$ time, while only requiring $O(q)$ preprocessing time. Given such a data structure, the time complexity of PRQ() becomes $O(1)$. The main procedure Q-MAXRTC() of Algorithm 12 is a direct extension of the main procedure 2-MAXRTC-FAST() of Algorithm 4, with the only difference being that we have q possible assignments for a leaf instead of 2. Hence, every triplet in $\mathcal{R}$ induces $O(q)$ calls to PRQ() of Algorithm 12, making the entire complexity of the algorithm $O(q) + O(q|\mathcal{R}|) = O(q|\mathcal{R}|)$. □

4.4 Implementation and Experiments

We used the C++ programming language to implement the algorithm from Theorem 5 for 2-MAXRTC, and the deterministic algorithm from Theorem 7 for q-MAXRTC when $q > 2$. The implementation is publicly available at `https://github.com/kmampent/qMAXRTC`. For simplicity in our implementation, given two nodes u and v, a trivial $O(q)$ time algorithm is used to compute $lca(u,v)$. To evaluate our algorithms experimentally, we also implemented *One-Leaf-Split* (`OLS`) [34] and *the modified Wu algorithm* (`Mod-Wu`) [16]. Below, we describe some experiments on both simulated and real datasets and the results.

Simulated Dataset. The input to q-MAXRTC is a set of triplets $\mathcal{R}$ and a parameter q. We define the following types of sets for $\mathcal{R}$:

- *dense consistent* (abbreviated `dc`): if $|\mathcal{R}| = \binom{n}{3}$ and $\mathcal{R}$ is consistent with a tree T containing $n-1$ internal nodes. The tree T is created using the uniform model [59].

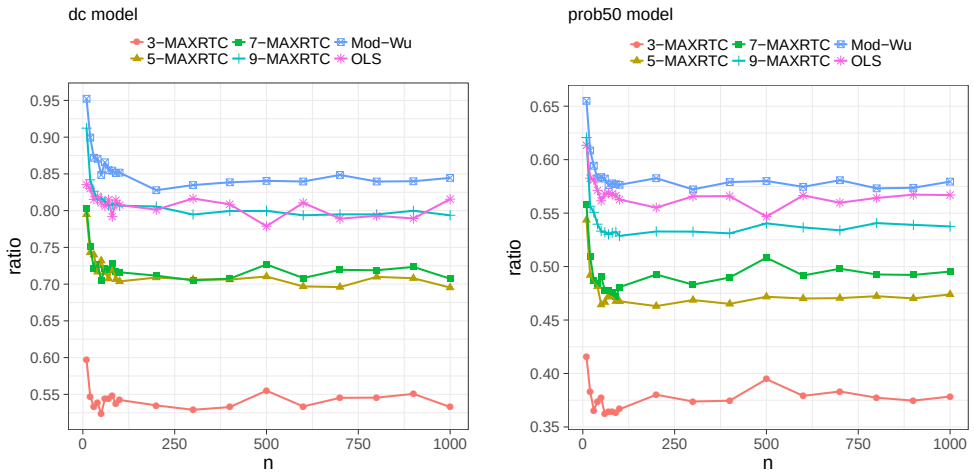


Figure 4.4: Performance of $\{3, 5, 7, 9\}$-MAXRTC compared to `OLS` and `Mod-Wu` in the `dc` and `prob50` models. Every data point corresponds to the mean of 100 runs. Observe that the performance of 9-MAXRTC is very close to that of `OLS` & `Mod-Wu`, even though 9-MAXRTC uses only 9 internal nodes, while `OLS` uses at most $n - 1$ internal nodes and `Mod-Wu` exactly $n - 1$.

- *probabilistic*: if $|\mathcal{R}| = n^2$ and $\mathcal{R}$ is a set of triplets on n leaf labels created as follows. After building a binary tree T on n leaves following the uniform model, start extracting triplets from T to add into $\mathcal{R}$. For every extracted triplet $xy|z$, permute the leaves uniformly at random with probability p. Depending on whether $p = 0.25$, $p = 0.50$ or $p = 0.75$ the abbreviations we use are `prob25`, `prob50`, and `prob75` respectively.
- *noisy*: if $|\mathcal{R}| = n^2$ and $\mathcal{R}$ is a set of triplets on n leaf labels that is created at random.

In the experiments of this dataset, the *performance* of an algorithm for any fixed q, n, and dataset model is defined as its mean approximation ratio, taken over 100 randomly generated instances of size n. Figure 4.4 compares the performance of q-MAXRTC, `OLS`, and `Mod-Wu` in the `dc` and `prob50` models, for small values of q and n at most 1000. In both models, the larger the value of q, the better the performance of q-MAXRTC. Moreover, the rate of improvement in performance decreases as the value of q increases, which is expected. For the `dc` dataset, which contains no conflicting triplets, the performance is much better. Significantly, a tree with just nine internal nodes (obtained by setting $q = 9$) can induce close to 80% of the triplets even if the input tree contains as many as 1000 leaves. When compared against `OLS` & `Mod-Wu`, we can see that while `OLS` & `Mod-Wu` perform better, the difference in performance is small compared to the difference in the number of internal nodes used by the algorithms.

In Figure C.1 we have the performance of q-MAXRTC for $q \in \{3, 5, 7, 9\}$,

q	poS1(761)	poS2(761)	poS4(841)	nmS4(1869)	nmS2(3082)	Average
2	0.27	0.36	0.43	0.41	0.29	0.35
3	0.67	0.54	0.48	0.41	0.46	0.51
5	0.77	0.81	0.67	0.66	0.72	0.73
7	0.82	0.75	0.76	0.62	0.73	0.74
9	0.86	0.71	0.87	0.80	0.79	0.81
11	0.91	0.89	0.87	0.79	0.87	0.87

Table 4.2: Accuracy of q-MAXRTC on real datasets. Every entry corresponds to the best accuracy over 100 runs. The size of each leaf label set is written inside the parenthesis.

`OLS` and `Mod-Wu` on the noisy, `prob25` and `prob75` models. In Figure C.2 we have the performance of q-MAXRTC for $q \in \{2, 5, 7, 9\}$ on every simulation model. In every experiment, the larger the value of q, the better the performance. As expected, the performance deteriorates when the number of conflicting triplets in $\mathcal{R}$ is increased.

Real Dataset We considered five trees from recently published papers ([42] and [56]). From [42] we used the trees from the supplementary datasets 2 and 4, denoted `nmS2` and `nmS4` respectively. From [56] we used the trees from the supplementary datasets 1, 2, and 4, denoted `poS1`, `poS2`, and `poS4` respectively. All trees are binary except `nmS2` and `nmS4`. However, we removed the leaf that is a child of `nmS2`'s root to make `nmS2` binary. Similarly, we removed the two leaves that are children of `nmS4`'s root to make `nmS4` binary as well. The total number of leaves in `nmS2`, `nmS4`, `poS1`, `poS2`, and `poS4` is 1869, 3082, 761, 761, and 841. Since the trees are binary, the total number of internal nodes is 1868, 3081, 760, 760, and 840.

For a tree $T \in \{\texttt{nmS2}, \texttt{nmS4}, \texttt{poS1}, \texttt{poS2}, \texttt{poS4}\}$ with n leaf labels, let T_q be the tree produced by the new algorithm from Theorem 7. Let $D(T, T_q)$ be the rooted triplet distance between T and T_q (for a definition see Section 4.5 below). The *accuracy* of q-MAXRTC in the experiments of this dataset is then defined by the ratio $S(T, T_q)/\binom{n}{3}$, where $S(T, T_q) = \binom{n}{3} - D(T, T_q)$. To compute this ratio efficiently, we used the rooted triplet distance implementation in [11]. We measured the accuracy of q-MAXRTC for $q \in \{2, 3, 5, 7, 9, 11\}$. Every experiment consisted of 100 runs, and in each run n^2 triplets were picked at random from the corresponding tree to define the set $\mathcal{R}$. We made sure that each leaf from a given tree appeared in $\mathcal{R}$ so that the size of the leaf label set was as big as the leaf label set of the tree.

Table 4.2 shows the best ratios achieved, and the corresponding trees in Newick format can be found at `https://github.com/kmampent/qMAXRTC`. As can be seen from the results, a larger number of internal nodes tends to improve the accuracy. Significantly, with only 9 nodes we can induce between

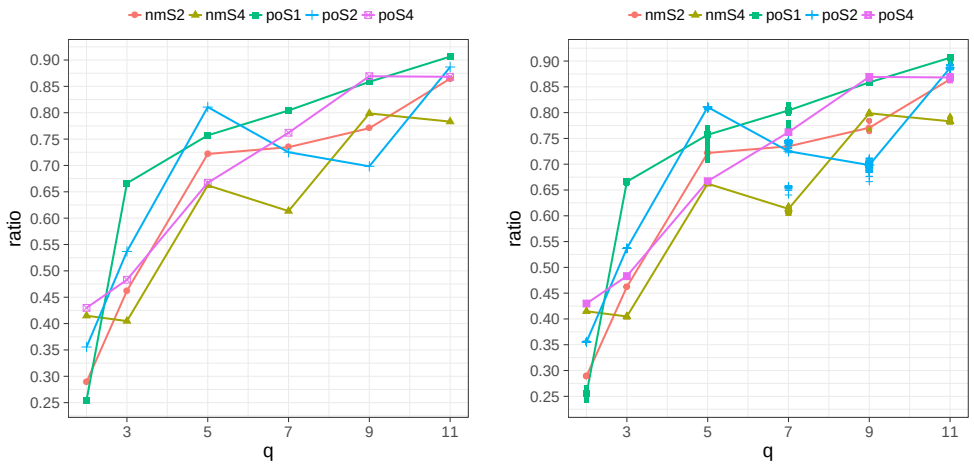


Figure 4.5: Graphical representation of Table 3.2. Performance of q-MAXRTC on real datasets. Left figure: Every data point corresponds to the mean of 100 runs. Right figure: All 100 data points appear in the figure.

q	poS1(761)	poS2(761)	poS4(841)	nmS4(1869)	nmS2(3082)
2	0.06	0.06	0.07	0.40	1.16
3	0.32	0.32	0.39	1.96	5.38
5	0.47	0.47	0.58	2.92	7.85
7	0.63	0.62	0.76	3.83	10.53
9	0.78	0.79	0.96	4.85	13.15
11	0.94	0.94	1.14	5.71	15.56

Table 4.3: The execution time in seconds, to generate every tree produced by q-MAXRTC on the real datasets. Every entry corresponds to the mean of 100 runs. The experiments were performed on a machine with 8GB RAM, Intel(R) Core(TM) i5-3470 CPU @ 3.20GHz, and having Ubuntu 16.04.2 LTS as an operating system.

71% and 86% of the triplets in each case, and with 11 nodes between 79% and 91%. When $q > 11$, we did not observe a significant improvement in accuracy. Figure 4.5 has graphical representations of Table 4.2, with all points plotted on the right graph. Because of the large sample size, i.e., n^2 triplets were picked at random to define $\mathcal{R}$, the quality of the trees does not differ significantly between the different runs. Finally, Table 4.3 gives the average number of seconds needed for q-MAXRTC to generate each tree from Table 4.2. The practicality of the algorithm can be seen from the fact that the largest amount of time spent for a tree is 15.56 seconds, and that is for the `nmS2` dataset which contains 3082 leaves and 3081 internal nodes.

4.5 Motivation for q-MAXRTC: Faster Rooted Triplet Distance

Finally, we give an example of the algorithmic advantage of using phylogenetic trees with few internal nodes. More precisely, we develop an algorithm for computing the rooted triplet distance between two phylogenetic trees in $O(qn)$ time, where q is the number of internal nodes in the smaller tree and n is the number of leaf labels. Finally, we provide an implementation of our new algorithm, as well as experimental results comparing the practical performance of our algorithm against previous state-of-the-art algorithms.

4.5.1 Problem Definition

The rooted triplet distance between two trees T_1 and T_2 built on the same leaf label set, is the total number of trees with three leaves that appear as embedded subtrees in T_1 but not in T_2. Intuitively, two trees with very similar branching structure will share many embedded subtrees, so the rooted triplet distance between them will be small.

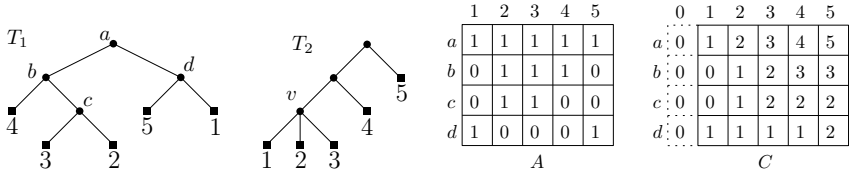
Formally, let T_1 and T_2 be two trees built on the same leaf label set of size n. We need to distinguish between two types of triplets. The first type is the *resolved triplet*, previously defined in Section 4.1. In addition, since T_1 and T_2 can be non-binary, we also need to define the *fan triplet*. We call $t = x|y|z$ a *fan triplet*, if t is a tree with the three leaves x, y, and z, and one internal node that is the root of t. The definition of when a resolved triplet is consistent with a tree T follows from Section 4.1. Similarly to a resolved triplet, we say that the fan triplet $x|y|z$ is consistent with a tree T, where x, y, and z are leaves in T, if $lca(x, y) = lca(x, z) = lca(y, z)$. In this section only, we use the word *triplet* to refer to both fan and resolved triplets. Moreover, when we refer to a fan triplet $x|y|z$ or a resolved triplet $xy|z$ induced by a tree T, there exists a left to right ordering of x, y, and z in T.

Let $D(T_1, T_2)$ be the rooted triplet distance between T_1 and T_2. Define $S(T_1, T_2)$ to be the total number of triplets that are consistent with both T_1 and T_2, commonly referred to as *shared triplets*. For the rooted triplet distance we then have that $D(T_1, T_2) = \binom{n}{3} - S(T_1, T_2)$.

4.5.2 The Algorithm

It is known how to compute $D(T_1, T_2)$ in $O(n \log n)$ time [11, 12]. Below, we show how to compute $D(T_1, T_2)$ in $O(qn)$ time, which is faster than [11, 12] when $q = o(\log n)$. There is a preprocessing step and a counting step.

Preprocessing. The leaves in T_2 are relabeled according to their discovery time by a depth first traversal of T_2, in which the children of a node are discovered from left to right. Notice that for a node v in T_2, the labels of the leaves in $T_2(v)$ will correspond to a continuous range of numbers. Afterwards,



	1	2	3	4	5
a	1	1	1	1	1
b	0	1	1	1	0
c	0	1	1	0	0
d	1	0	0	0	1

A

	0	1	2	3	4	5
a	0	1	2	3	4	5
b	0	0	1	2	3	3
c	0	0	1	2	2	2
d	0	1	1	1	1	2

C

Figure 4.6: Changing the leaf labels of the trees from Figure 4.1 according to the preprocessing step of the rooted triplet distance algorithm in Section 4.5. The leaves in $T_2(v)$ are defined by the range of leaf labels $[1, 3]$. The number of leaves appearing in both $T_2(v)$ and $T_1(b)$ is $C[b][3] - C[b][0] = 2$.

we transfer the new labels of the leaves in T_2 to the leaves in T_1. For T_1, we define the $q \times n$ table A such that for an internal node u in T_1 we have $A[u][\ell] = 1$ if ℓ is a leaf in $T_1(u)$, and $A[u][\ell] = 0$ otherwise. We construct another table C to answer one dimensional range queries as follows. For $1 \leq i \leq n$ we have $C[u][i] = \sum_{j=1}^{i} A[u][j]$ and $C[u][0] = 0$. The C table will be used to answer queries asking for the total number of leaves in $T_2(v)$ that are also in $T_1(u)$ in $O(1)$ as follows. Let $[l, \ldots, r]$ be the continuous range of leaf labels in $T_2(v)$. The answer to the query will be exactly $C[u][r] - C[u][l-1]$ (see Figure 4.6 for an example).

Counting. We extend the technique introduced in [11]. Let $t = xy|z$ or $t = x|y|z$ be a triplet induced by a tree T, which in our problem can be either T_1 or T_2. We anchor t in the edge $\{v, c\}$, where $v = lca(x, y)$ and c is the child of v such that $T(v)$ contains y. The following lemma shows that every triplet induced by T is anchored in exactly one edge of T.

Lemma 21. *Let T be a tree in which every triplet t with the three leaves x, y, and z is anchored in the edge $\{u, c\}$, such that $u = lca(x, y)$ and $T(c)$ contains y. Every triplet induced by T is anchored in exactly one edge of T.*

Proof. Assume by contradiction that t is anchored in i edges where $i \geq 2$. Let those edges be $\{u_1, v_1\}, \ldots, \{u_i, v_i\}$. By definition, $u_1 = lca(x, y)$. For the same reason, we have $u_2 = lca(x, y), \ldots, u_i = lca(x, y)$. Because T is a tree, it must hold that $u_1 = u_2 = \ldots = u_i$. Again by definition, $v_1, \ldots, v_i$ must all be the children of u_1 such that every subtree in $S = \{T(v_1), \ldots, T(v_i)\}$ contains leaf y. Because T is a tree, there can only be one such child of u_1, meaning that $|S| = 1$, which leads to a contradiction. □

Suppose that a node v in T_2 has the children $v_1, \ldots, v_j, \ldots, v_i$, where $1 < j \leq i$. To detect all triplets anchored in edge $\{v, v_j\}$ of T_2, we color the leaves of T_2 as follows. Let every leaf in $T_2(v_1), \ldots, T_2(v_{j-1})$ have the color red, every leaf in $T_2(v_j)$ have the color blue, every leaf in $T_2(v_{j+1}), \ldots, T_2(v_i)$ have the color green and every other leaf in T_2 have the color white. The red, blue, and green colors will be used to detect fan triplets and the red, blue, and white colors, resolved triplets. An illustrative example can be found

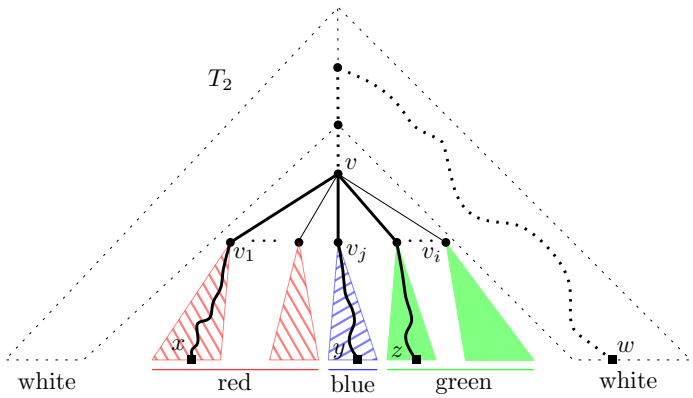


Figure 4.7: Using colors to detect triplets in T_2. Three leaves x, y, and z that have the color red, blue, and green respectively, define the fan triplet $x|y|z$. Three leaves x, y, and z that have the color red, blue, and white respectively, define the resolved triplet $xy|z$.

in Figure 4.7. By the relabeling scheme of the leaves, we have that the red, blue, and green colors correspond to exactly one continuous range of leaf labels each. Let those ranges be $R = [a_{\text{red}}, \ldots, a'_{\text{red}}]$, $B = [a_{\text{blue}}, \ldots, a'_{\text{blue}}]$, and $G = [a_{\text{green}}, \ldots, a'_{\text{green}}]$, for the colors red, blue, and green respectively. Note that we have $a_{\text{blue}} = a'_{\text{red}} + 1$ and if G is non-empty, $a_{\text{green}} = a'_{\text{blue}} + 1$. Finally, note that a leaf has the color white if and only if it does not have any other color.

We are now going to describe how to compute the total number of triplets anchored in some edge $\{v, v'\}$ in T_2, where v is the parent of v', that are also consistent with T_1, denoted $S^{\{v,v'\}}(T_1, T_2)$. Let $S_r^{\{v,v'\}}(T_1, T_2)$ denote the shared resolved triplets anchored in $\{v, v'\}$ and similarly let $S_f^{\{v,v'\}}(T_1, T_2)$ denote the shared fan triplets. For the value of $S^{\{v,v'\}}(T_1, T_2)$ we then have that $S^{\{v,v'\}}(T_1, T_2) = S_r^{\{v,v'\}}(T_1, T_2) + S_f^{\{v,v'\}}(T_1, T_2)$. The following lemma gives an algorithm for computing $S^{\{v,v'\}}(T_1, T_2)$ efficiently.

Lemma 22. *Given the ranges R, B, and G that define a coloring of the leaves in T_2 according to an edge $\{v, v'\}$ of T_2, there exists a $O(q)$-time algorithm for computing $S^{\{v,v'\}}(T_1, T_2)$.*

Proof. Since both T_1 and T_2 are built on the same leaf label set, a coloring of the leaves of T_2 defines a coloring of the leaves of T_1. Suppose that a node u in T_1 has the m children $u_1, \ldots, u_m$, where $m \geq 2$. Some children could be leaves and others, internal nodes. Let I denote the set containing the children that are internal nodes and L the children that are leaves. Let $T(I) = \{T(u) : u \in I\}$. Define the following counters:

1. u_{white}: total number of leaves with the white color in T_1 but not in $T_1(u)$.

Algorithm 5 Computing $S_f^{\{v,v'\}}(T_1, T_2)$ and $S_r^{\{v,v'\}}(T_1, T_2)$ in $O(q)$ time.

1: **procedure** $S_f^{\{v,v'\}}(T_1, T_2)$
2: fans $= 0$
3: **for** every internal node u in T_1 **do**
4: fans $=$ fans $+ u_{\text{red,blue,green}}$
5: fans $=$ fans $+ u_{\text{red,blue}} \cdot u_{\text{green}L} + u_{\text{red,green}} \cdot u_{\text{blue}L} + u_{\text{blue,green}} \cdot u_{\text{red}L}$
6: fans $=$ fans $+ u_{\text{red}I} \cdot u_{\text{blue}L} \cdot u_{\text{green}L} + u_{\text{blue}I} \cdot u_{\text{red}L} \cdot u_{\text{green}L} + u_{\text{green}I} \cdot u_{\text{red}L} \cdot u_{\text{blue}L}$
7: fans $=$ fans $+ u_{\text{red}L} \cdot u_{\text{blue}L} \cdot u_{\text{green}L}$
8: **return** fans
9: **procedure** $S_r^{\{v,v'\}}(T_1, T_2)$
10: resolved $= 0$
11: **for** every internal node u in T_1 **do**
12: resolved $=$ resolved $+ u_{\text{red,blue}} \cdot u_{\text{white}}$
13: resolved $=$ resolved $+ u_{\text{red}I} \cdot u_{\text{blue}L} \cdot u_{\text{white}} + u_{\text{blue}I} \cdot u_{\text{red}L} \cdot u_{\text{white}}$
14: resolved $=$ resolved $+ u_{\text{red}L} \cdot u_{\text{blue}L} \cdot u_{\text{white}}$
15: **return** resolved

2. u_i, for $i \in \{\text{red}, \text{blue}, \text{green}\}$: total number of leaves with color i in $T_1(u)$.
3. u_{iI}, for $i \in \{\text{red}, \text{blue}, \text{green}\}$: total number of leaves with color i in $T(I)$.
4. u_{iL}, for $i \in \{\text{red}, \text{blue}, \text{green}\}$: total number of leaves with color i in L.
5. $u_{i,j}$, for $(i,j) \in \{(red, blue), (red, green), (blue, green)\}$: total number of pairs of leaves in $T(I)$, such that one has color i, the other has color j and both come from different subtrees attached to u.
6. $u_{\text{red, blue, green}}$: total number of leaf triples in $T(I)$, such that one leaf has the color red, another the color blue, another the color green and they all come from different subtrees attached to u.

Algorithm 13 in Appendix C shows how to compute these counters for every internal node of T_1 efficiently. A depth first traversal is applied on T_1 while making sure that we only visit internal nodes. For every node u in I we apply a dynamic programming procedure (lines 17-37) to compute the counters. Since in every recursive call we spend $O(|I|)$ time, the total time of the algorithm is $O(q)$.

After computing all counters in T_1 by applying Algorithm 13, Algorithm 5 shows how to compute $S_f^{\{v,v'\}}(T_1, T_2)$ and $S_r^{\{v,v'\}}(T_1, T_2)$ in $O(q)$ time as well. It counts shared triplets by considering for every internal node u in T_1, all possible cases for the location of the leaves of a shared triplet anchored in any edge $\{u, u'\}$ in T_1, where u is the parent of u'. More precisely, for the leaves of a fan triplet anchored in any edge $\{u, u'\}$ in T_1, we have the following cases:

1. all three leaves come from $T(I)$ (line 4, e.g., 8|3|7 in Figure 4.8).
2. two leaves come from $T(I)$ and one from L (line 5, e.g., 7|3|2 in Fig-

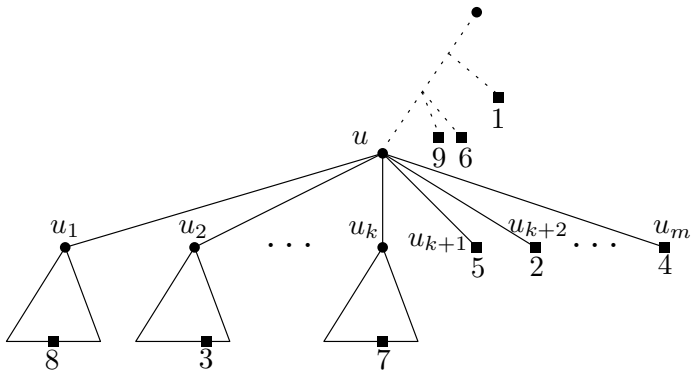


Figure 4.8: Rooted triplet distance computation: an internal node u in T_1, having m children k of which are internal nodes. In Algorithm 5 we have $I = \{u_1, \ldots, u_k\}$ and $L = \{u_{k+1}, \ldots, u_m\}$.

ure 4.8).

3. one leaf comes from $T(I)$ and two from L (line 6, e.g., 3|5|4 in Figure 4.8).
4. all three leaves come from L (line 7, e.g., 4|2|5 in Figure 4.8).

Similarly, for the leaves of a resolved triplet we have the following cases:

1. two leaves come from $T(I)$ and one not from $T_1(u)$ (line 11, e.g., 38|6 in Figure 4.8).
2. one leaf comes from $T(I)$, one from L, and one not from $T_1(u)$ (line 12, e.g., 35|1 in Figure 4.8).
3. two leaves come from L and one not from $T_1(u)$ (line 13, e.g., 52|9 in Figure 4.8).

Since we have that $S^{\{v,v'\}}(T_1, T_2) = S_r^{\{v,v'\}}(T_1, T_2) + S_f^{\{v,v'\}}(T_1, T_2)$, the statement follows. □

In Algorithm 6 we show how to compute $D(T_1, T_2)$. From the preprocessing step, line 2 requires $O(qn)$ time. Line 3 is performed by a depth first traversal of T_1, thus requiring $O(n)$ time. From Lemma 22, lines 7-9 require $O(q)$ time. Since we also have that $\sum_{v \in T_2} deg(v) = O(n)$, the total time required to compute $D(T_1, T_2)$ is $O(qn)$. The correctness is ensured by Lemma 21, thus we obtain the following theorem:

Theorem 8. *The rooted triplet distance between two rooted phylogenetic trees T_1 and T_2 built on the same leaf label set of size n, can be computed in $O(qn)$ time, where q is the total number of internal nodes in T_1.*

4.5.3 Implementation and Experiments

We have implemented the $O(qn)$-time triplet distance algorithm in the C++ programming language. The source code is available at `https://github.com/kmampent/qtd`. The experiments were performed on a machine with

Algorithm 6 O(qn)-time algorithm for computing $D(T_1, T_2)$

```
1: procedure D(T_1, T_2)
2:     Compute the q × n table C.
3:     For every u in T_1 compute the parameter u_l, which is the number of
   leaves in T(u).
4:     shared = 0
5:     for every internal node v in T_2 do
6:         for every child v' of v do
7:             Let R, B, and G be the color ranges defined by edge {v, v'}
8:             COUNTERS(T_1, C, R, B, G)                    ▷ From Lemma 22
9:             shared = shared + S_f^{{v,v'}}(T_1, T_2)+S_r^{{v,v'}}(T_1, T_2)
10:    return (n choose 3) − shared
```

8GB RAM, Intel(R) Core(TM) i5-3470 CPU @ 3.20GHz, and having Ubuntu 16.04.2 LTS as an operating system. For the space usage we considered the *Maximum resident set size* parameter returned by /usr/bin/time -v. We run the experiments using the following two different models for generating input trees:

- *Model A*: Generate a binary tree with n leaves for T_2 following the uniform model [59]. For T_1, first create a tree with q internal nodes as follows: start with a tree containing one internal node, iteratively pick an internal node u uniformly at random and add a new internal node w to be a child of u. To assign the leaves, start by assigning one leaf to every internal node that has only one child and assign every other leaf by picking internal nodes uniformly at random.
- *Model B*: Generate a binary tree with n leaves for T_2 following the uniform model [59]. For T_1, start by generating a binary tree with n leaves following the uniform model. Let S be a set containing the root and $q-1$ internal nodes picked uniformly at random. Contract every other internal node u that is not in S by having the children of u become the children of u's parent.

We compared our algorithm against the O($n \log n$)-time algorithm in [11] and the O($n \log^3 n$)-time algorithm in [46]. We call our algorithm `qtd` and the algorithm in [11] `cachetd`. In [46] two different implementations are provided, one that uses unordered_map[1], which we refer to as `cpdt`, and one that uses sparsehash[2], which we refer to as `cpdtg`.

Every data point in the figures corresponds to the mean of 50 different runs, each run on a different pair of input trees. In Figure 4.9 we have the execution time in seconds as well as the space usage of the algorithms in model A and when $n = 10^5$. In Figure 4.10 we have the same experiments except

[1] `http://en.cppreference.com/w/cpp/container/unordered_map`
[2] `https://github.com/sparsehash/sparsehash`

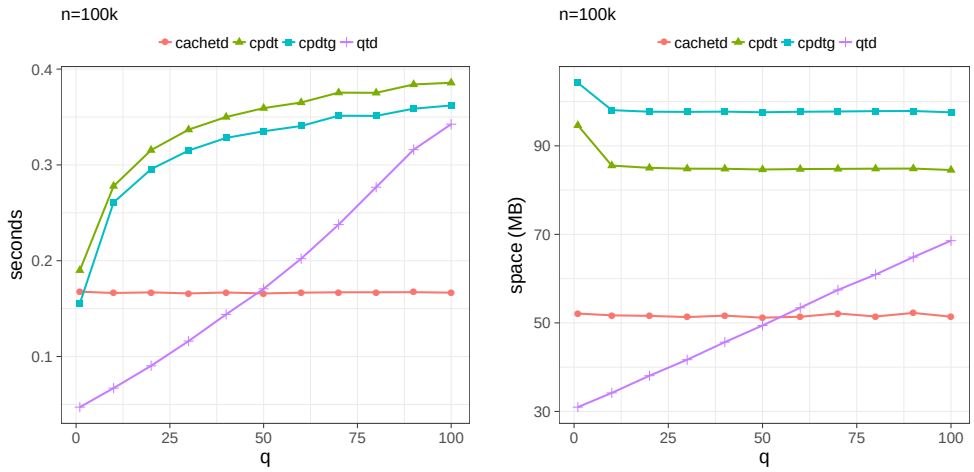


Figure 4.9: Model A: Running time and space usage of the different triplet distance algorithms when $n = 10^5$.

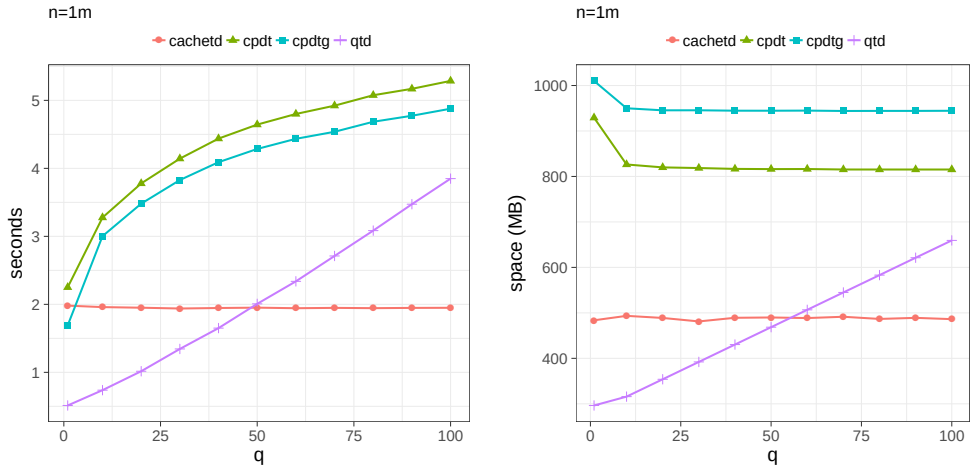


Figure 4.10: Model A: Running time and space usage of the different triplet distance algorithms when $n = 10^6$.

now $n = 10^6$. Figures C.3 and C.4 have the previous experiments but in model B. The results indicate that our implementation uses less space and is faster than the previous algorithms when $q \leq 50$.

4.6 Concluding Remarks

In this paper, we introduced the problem of building a phylogenetic tree with q internal nodes that induces the largest number of triplets from an input triplet

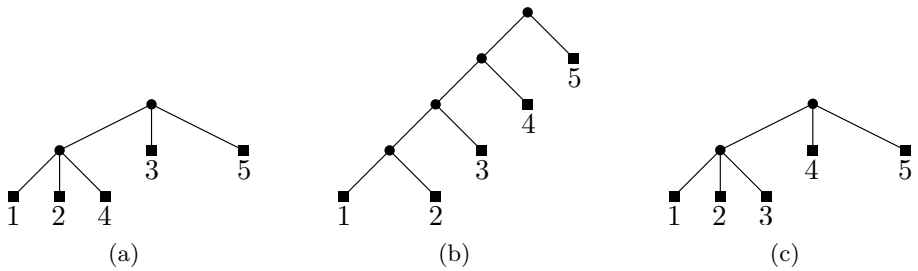


Figure 4.11: Let $\mathcal{R} = \{12|3, 13|4, 24|5\}$. (a) The optimal tree for 2-MAXRTC induces 2 triplets from $\mathcal{R}$. (b) The tree returned by the BUILD algorithm from [2]. (c) The best tree obtainable by contracting all internal edges except one in the tree from (b) induces only 1 triplet from $\mathcal{R}$, so this method is not optimal for 2-MAXRTC.

set $\mathcal{R}$, denoted q-MAXRTC. We showed that q-MAXRTC is NP-hard for every fixed $q \geq 2$. For 2-MAXRTC, no polynomial-time approximation algorithm with a relative (resp. absolute) approximation ratio better than $\frac{16}{17} + \varepsilon$ (resp. $\frac{4}{27} + \varepsilon$) can exist. When $q \geq 3$, the inapproximability bound becomes $1 - 1/(34q) + \varepsilon$. We reduced 2-MAXRTC to MAX 3-AND and obtained several polynomial-time approximation algorithms that, however, could not scale with q. We then proposed a simple polynomial-time $\frac{4}{27}$-approximation algorithm that could be extended to scale with any $q \geq 3$. Finally, for two trees with one having q internal nodes and both being built on the same leaf label set of size n, we presented a $O(qn)$-time algorithm for the rooted triplet distance computation. We have also implemented the $O(qn)$-time algorithm described in Section 4.5. Our experiments indicate that our prototype implementation uses less space and is faster than the state-of-the-art, optimized implementation of the $O(n \log n)$-time algorithm from [11] for large inputs, e.g., when $n = 1,000,000$ and $q \leq 50$.

4.6.1 Open Problems

The optimal polynomial-time approximation ratio for any fixed $q \geq 3$ is an open problem, as well as the existence of algorithms achieving that ratio. Moreover, for the special case where all the triplets in $\mathcal{R}$ are consistent with a tree T, the computational complexity of q-MAXRTC is an open problem as well. Note that just applying BUILD [2] to obtain such a T and then trying every bipartition of L induced by an edge of T fails to produce an optimal solution to 2-MAXRTC (see Figure 4.11 for a counterexample). Another open problem is the existence of approximation algorithms for q-MAXRTC in the weighted case, where each triplet in $\mathcal{R}$ has a weight and the objective is to build a tree that maximizes the total weight of the triplets induced from $\mathcal{R}$. This addresses the case where some triplets in $\mathcal{R}$ are more important than others.

Moreover, another open problem is the following: given a set of triplets $\mathcal{R}$ on a leaf label set of size n and a parameter ℓ, build a tree T with ℓ leaves such that $|rt(T) \cap \mathcal{R}|$ is maximized. Just like q-MAXRTC is a combination of MINRS and MAXRTC, this new problem is a combination of the *maximum agreement supertree problem* studied in [47] and MAXRTC. Finally, for the rooted triplet distance computation, a major open problem [11, 12] is whether it can be computed in $O(n)$ time. When $q = O(1)$, our proposed algorithm runs in $O(n)$ time. If q_1 is the total number of internal nodes of one tree and q_2 of the other, is it possible to obtain an algorithm with a $O(q_1 q_2 + n)$ running time?

Appendix A

Additional Experiments for Chapter 2

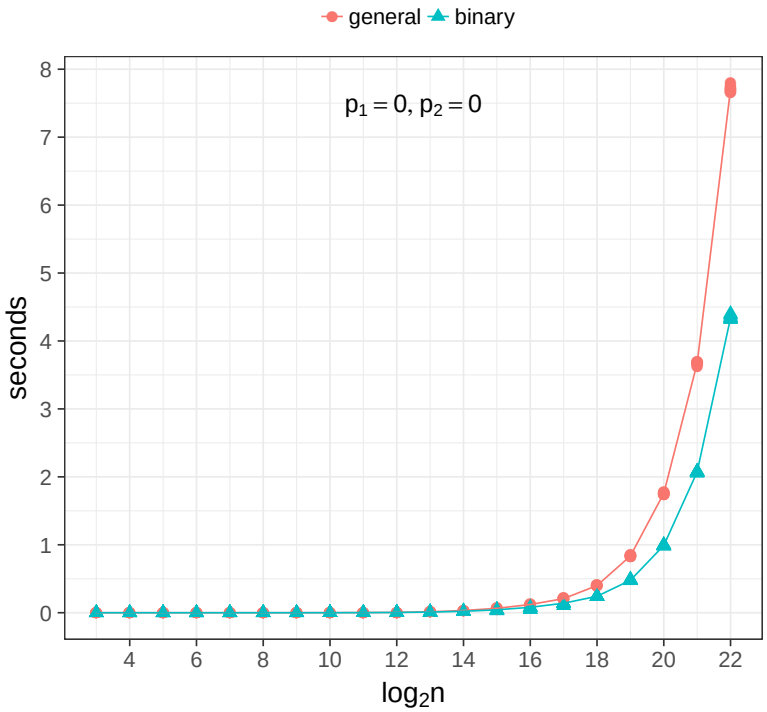


Figure A.1: CacheTD: performance of binary (Section 2.3) and general (Section 2.4) implementation on binary trees. All data points of the 10 runs are visible in the figure. Each run is on a different tree and the line connects the median of the runs.

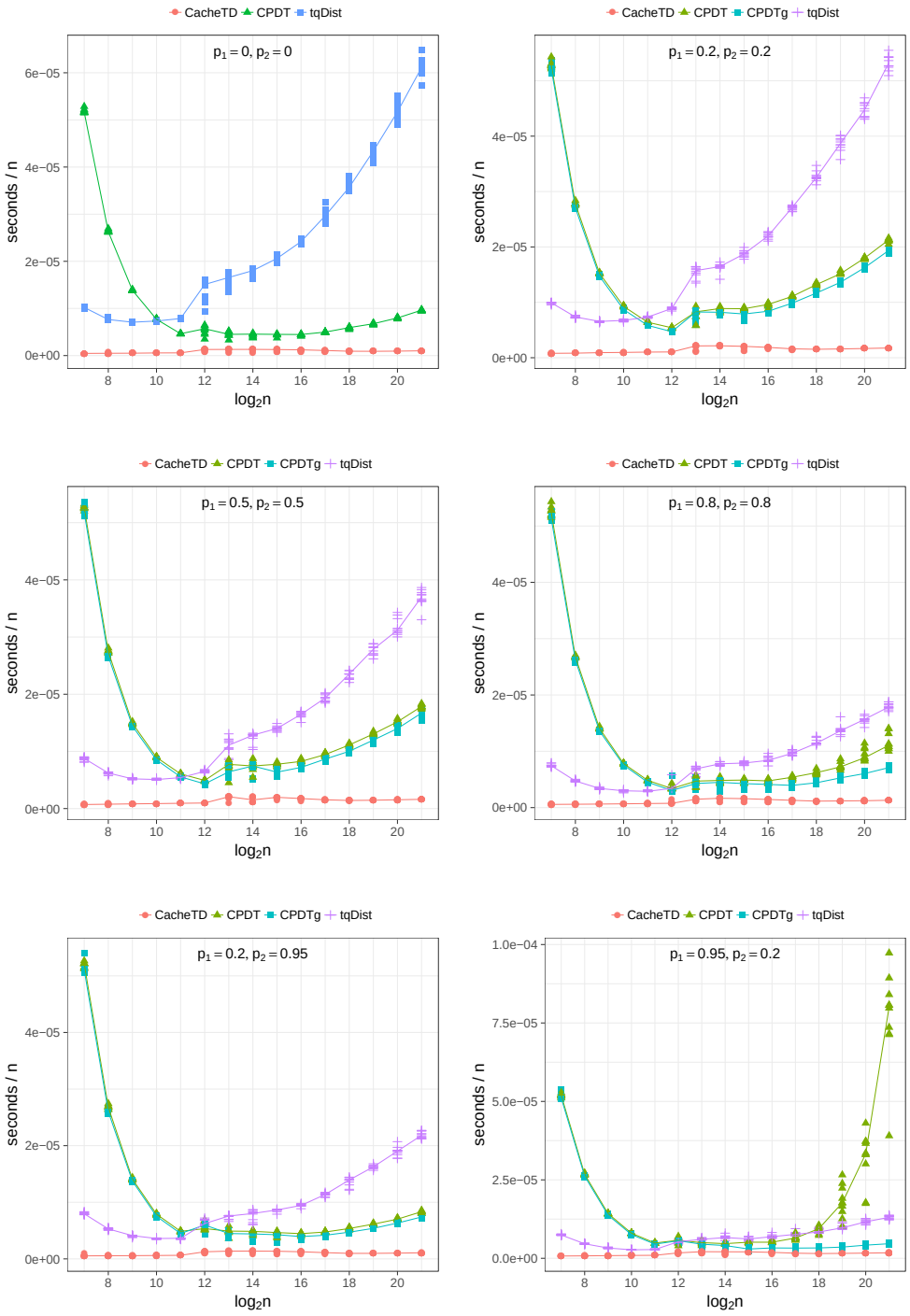


Figure A.2: Random model: time performance, where `CPDT` is compiled in g++ version 5.4.

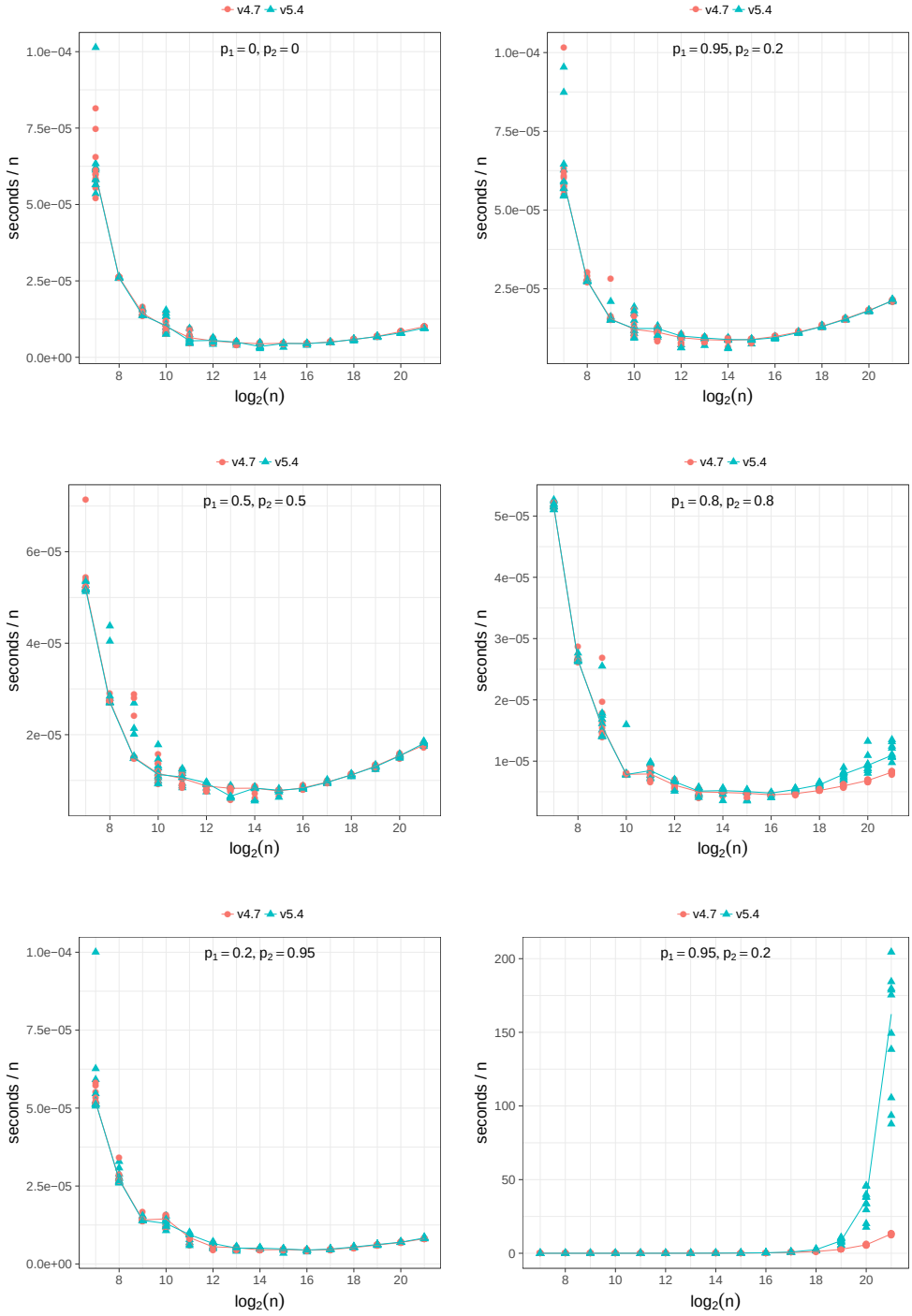


Figure A.3: Random model: time performance of `CPDT` when compiled with g++ 4.7 and g++ 5.4.

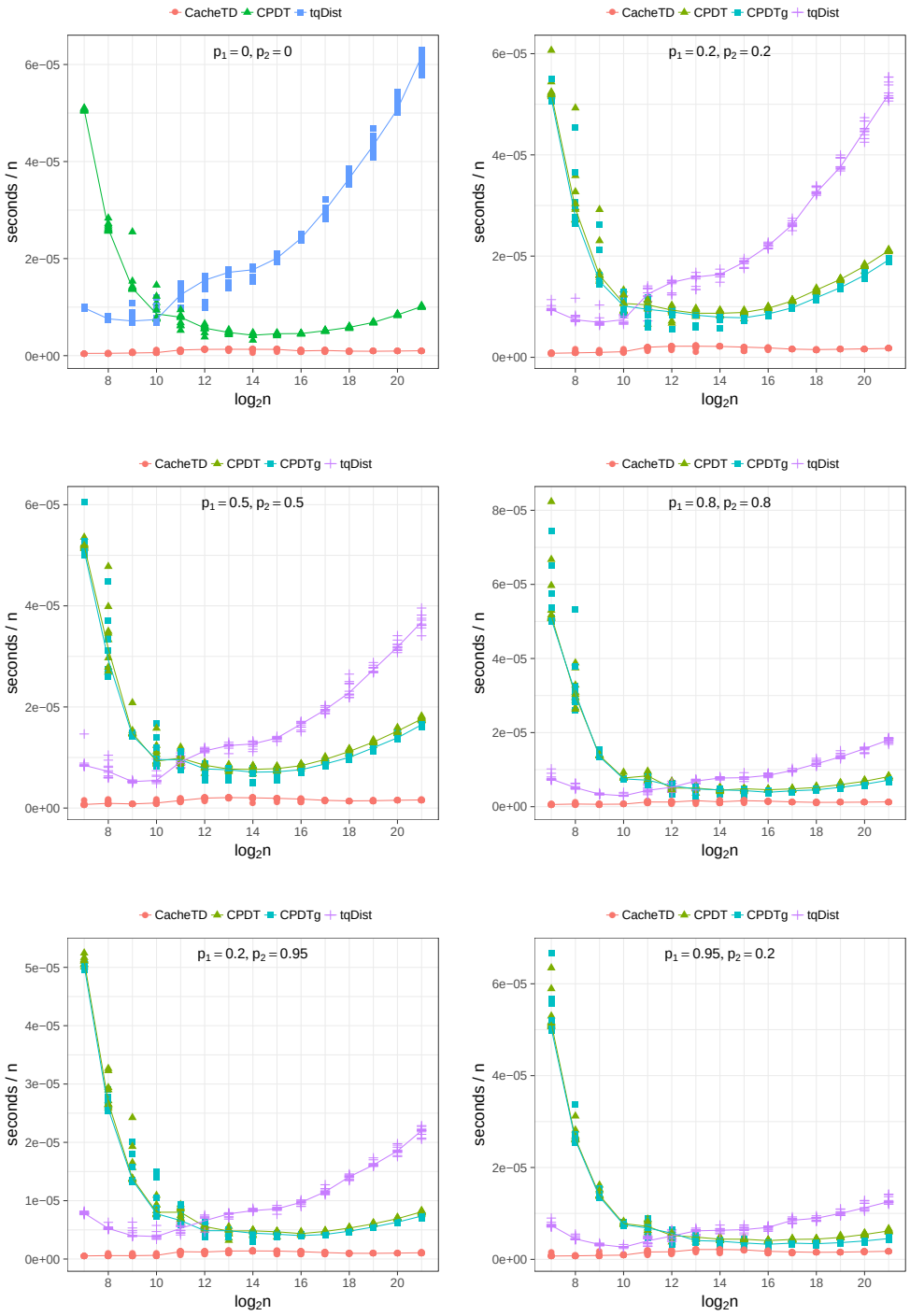


Figure A.4: Random model: time performance, where `CPDT` is compiled in g++ version 4.7.

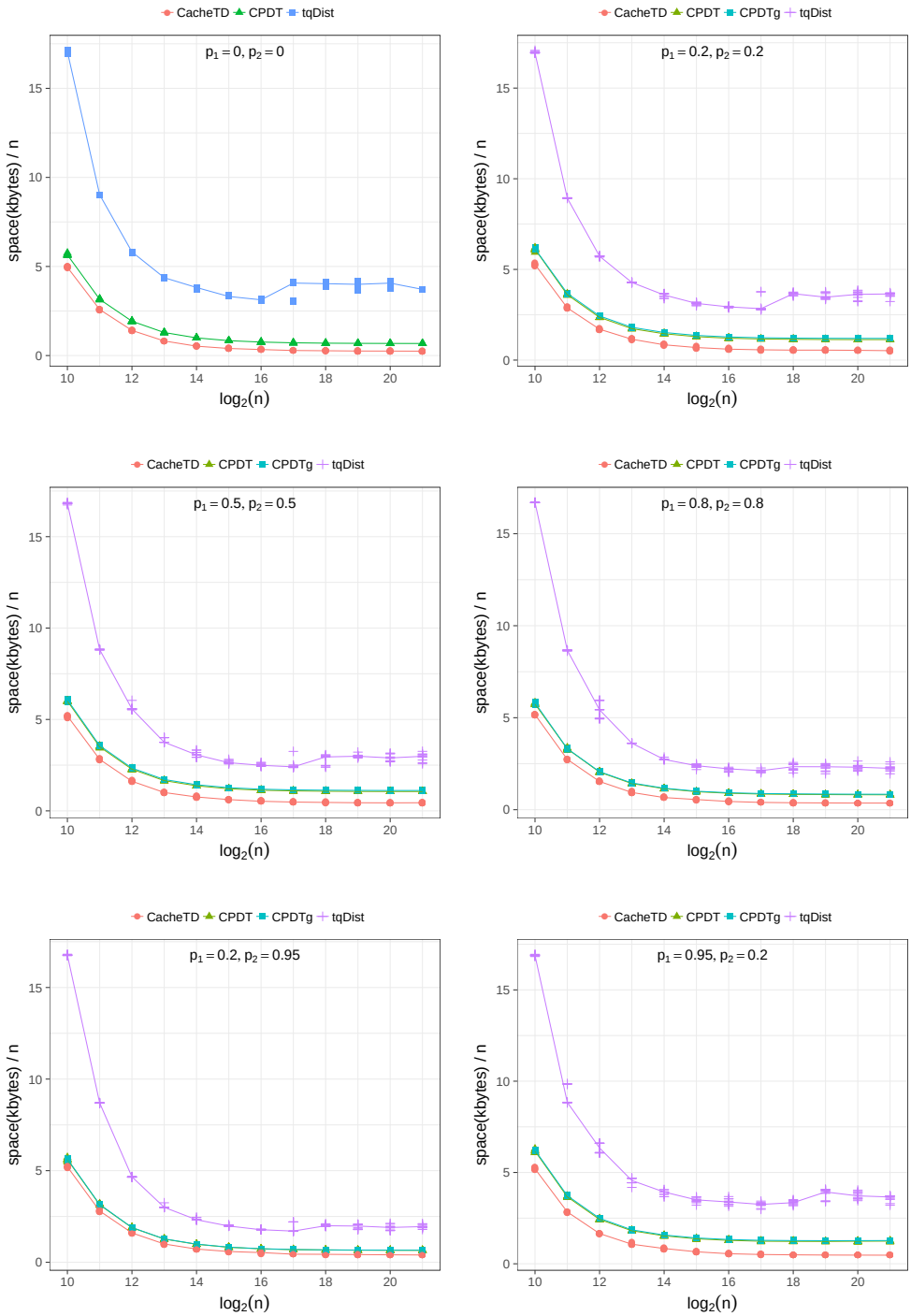


Figure A.5: Random model: space performance.

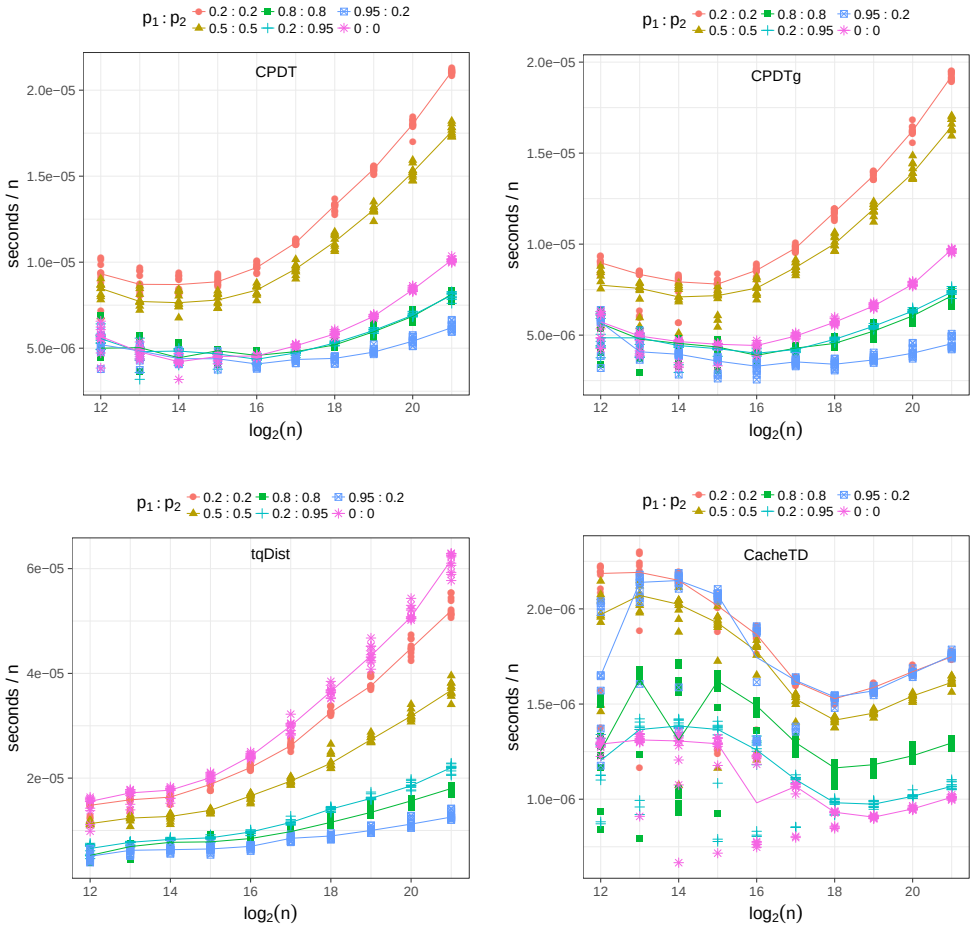


Figure A.6: Random model: how the contraction parameter affects execution time.

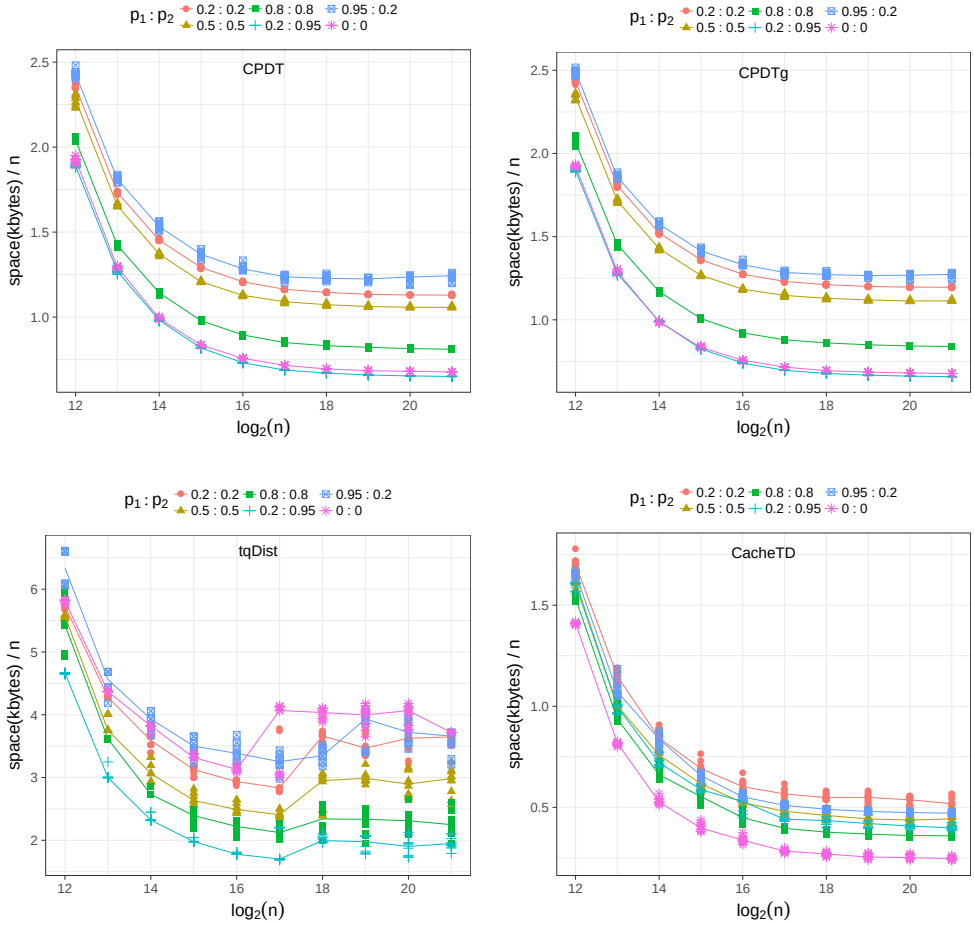


Figure A.7: Random model: how the contraction parameter affects space.

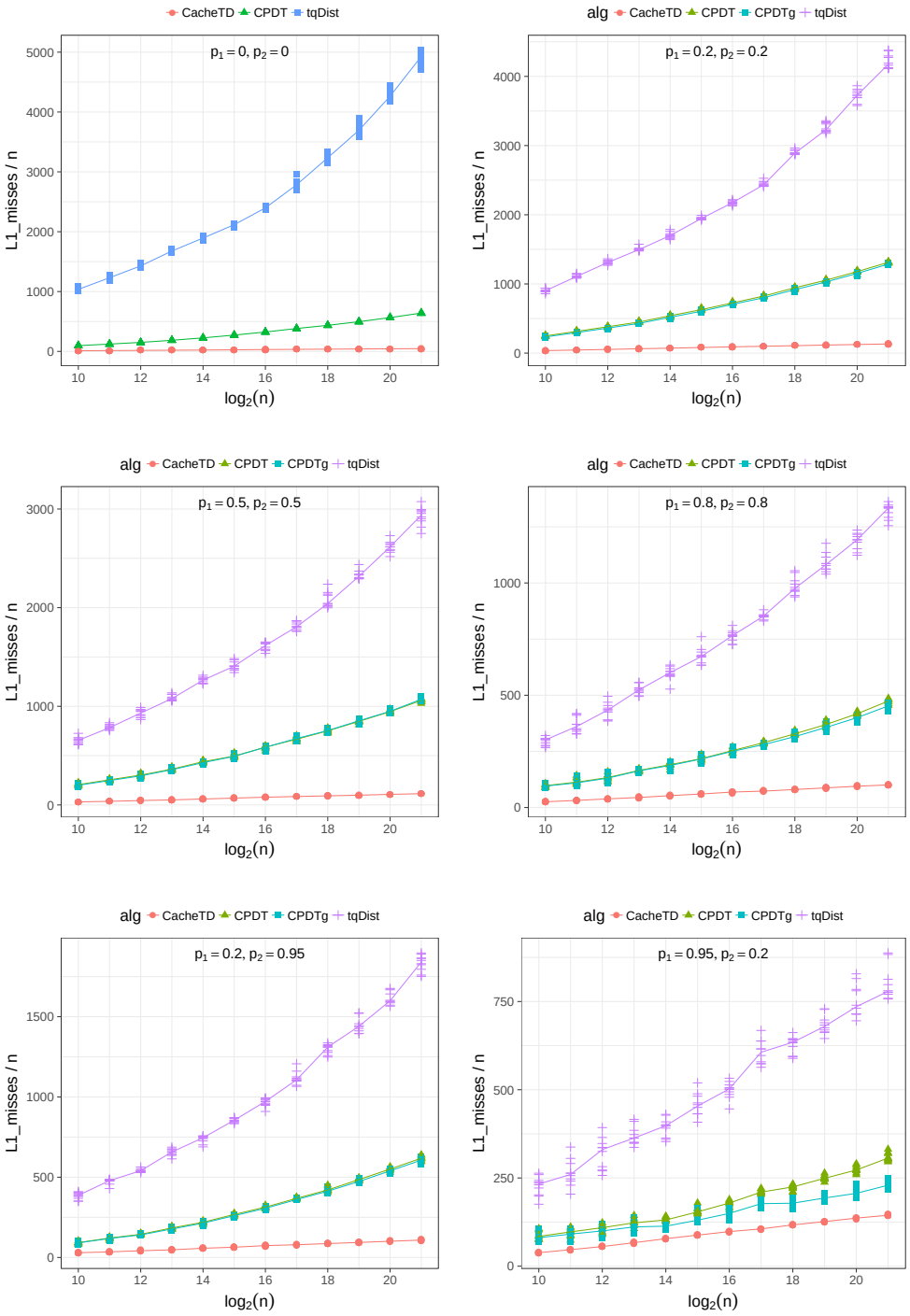


Figure A.8: Random model: L1 cache misses.

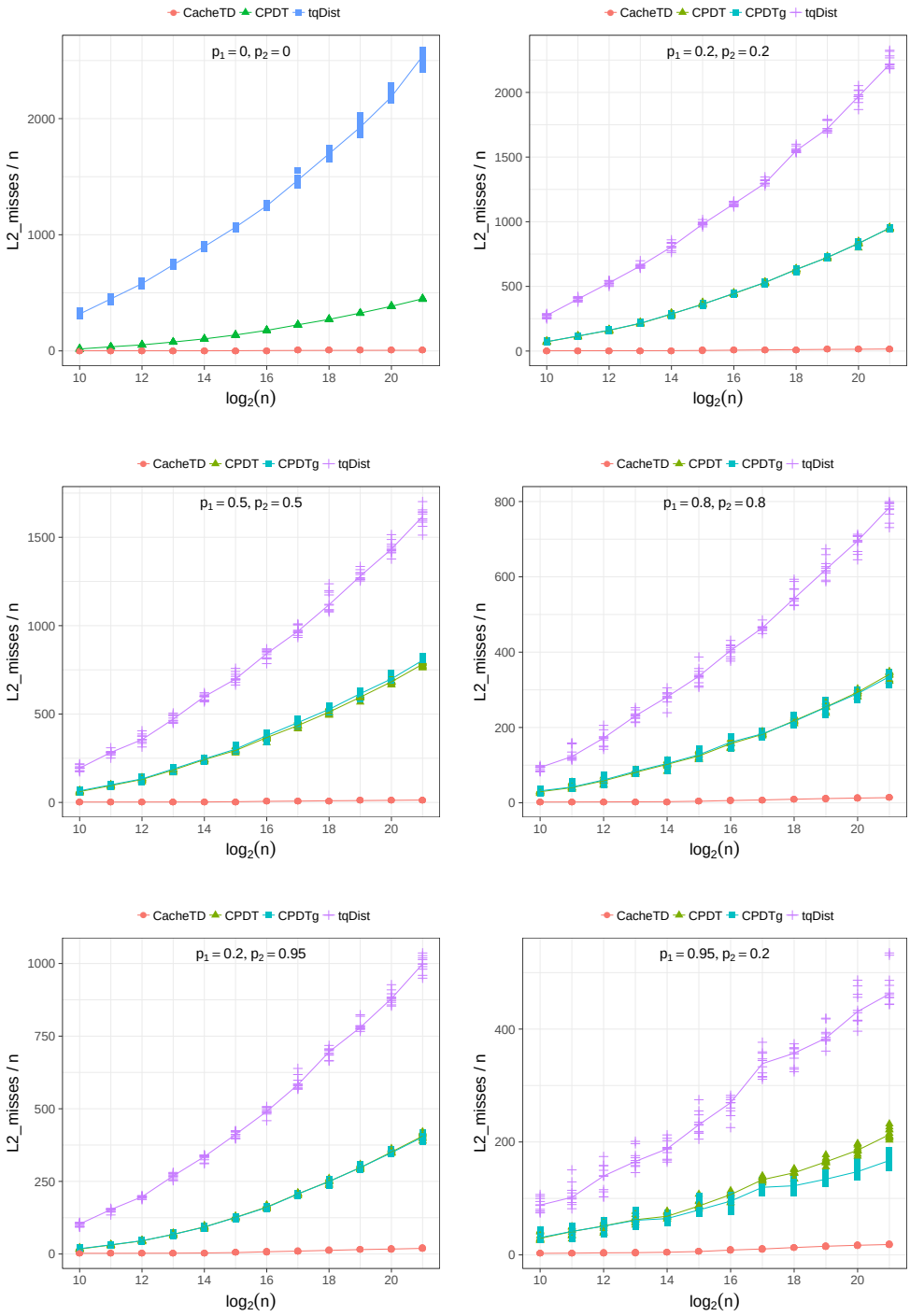


Figure A.9: Random model: L2 cache misses.

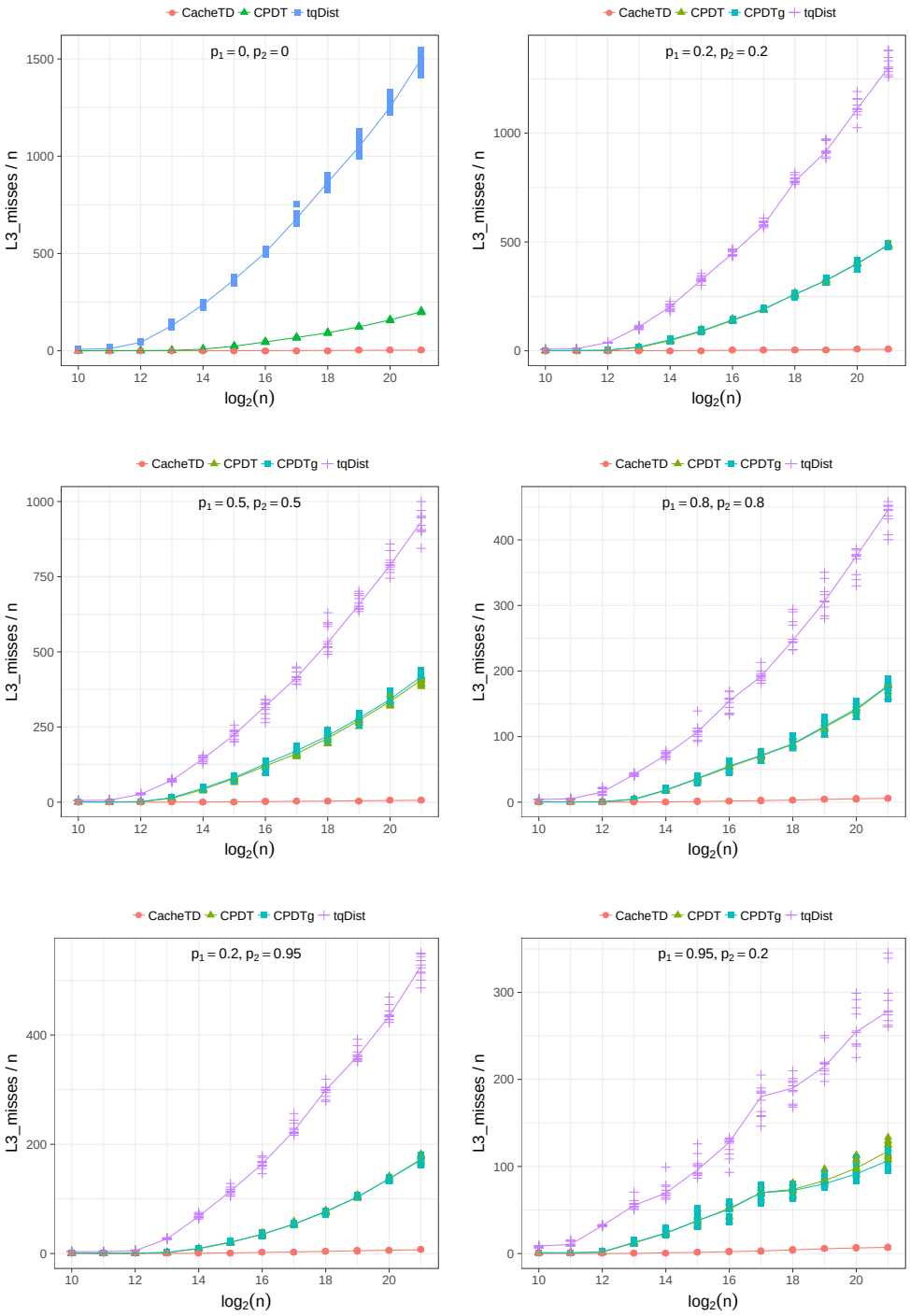


Figure A.10: Random model: L3 cache misses.

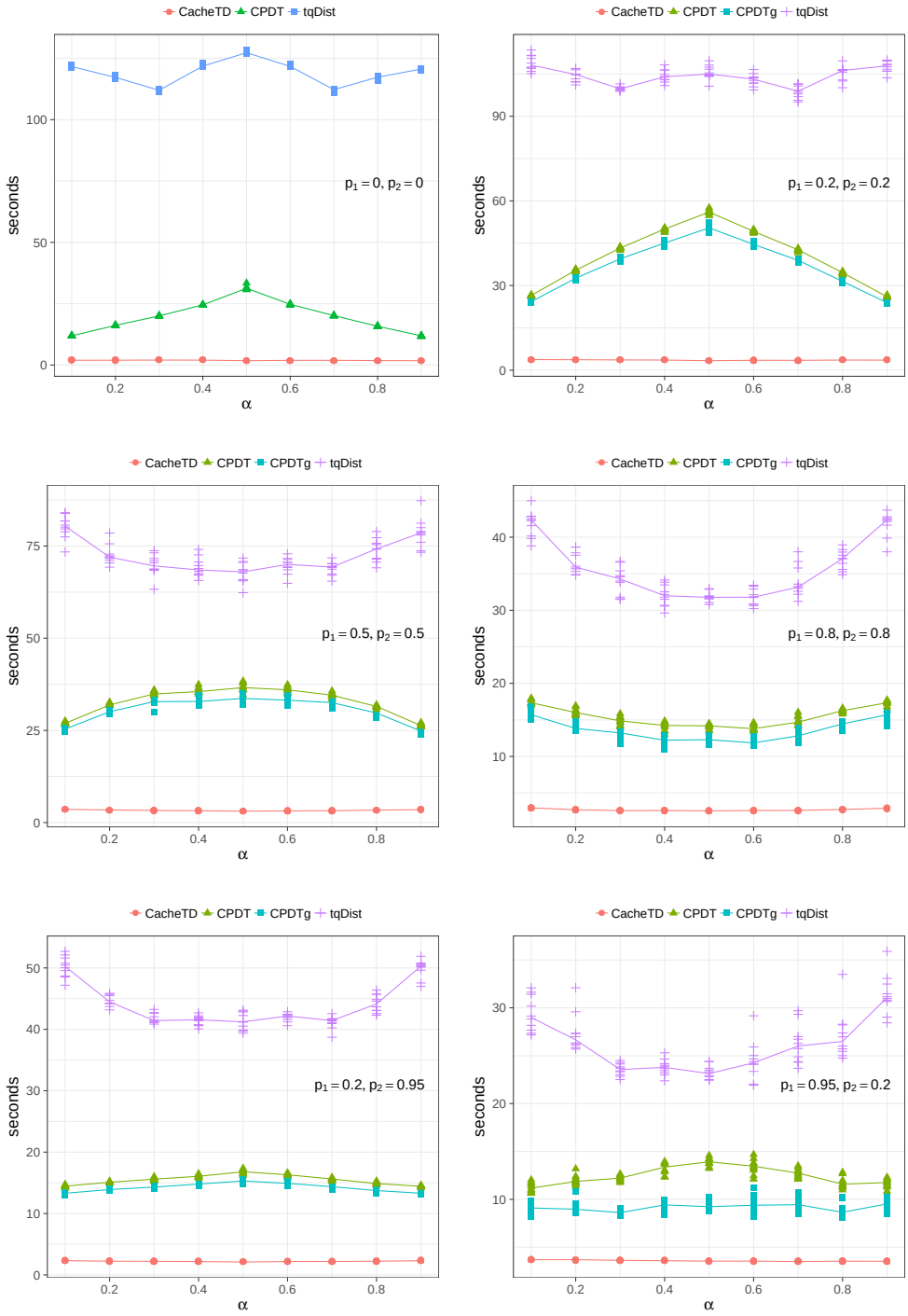


Figure A.11: Skewed model: running time ($n = 2^{21}$).

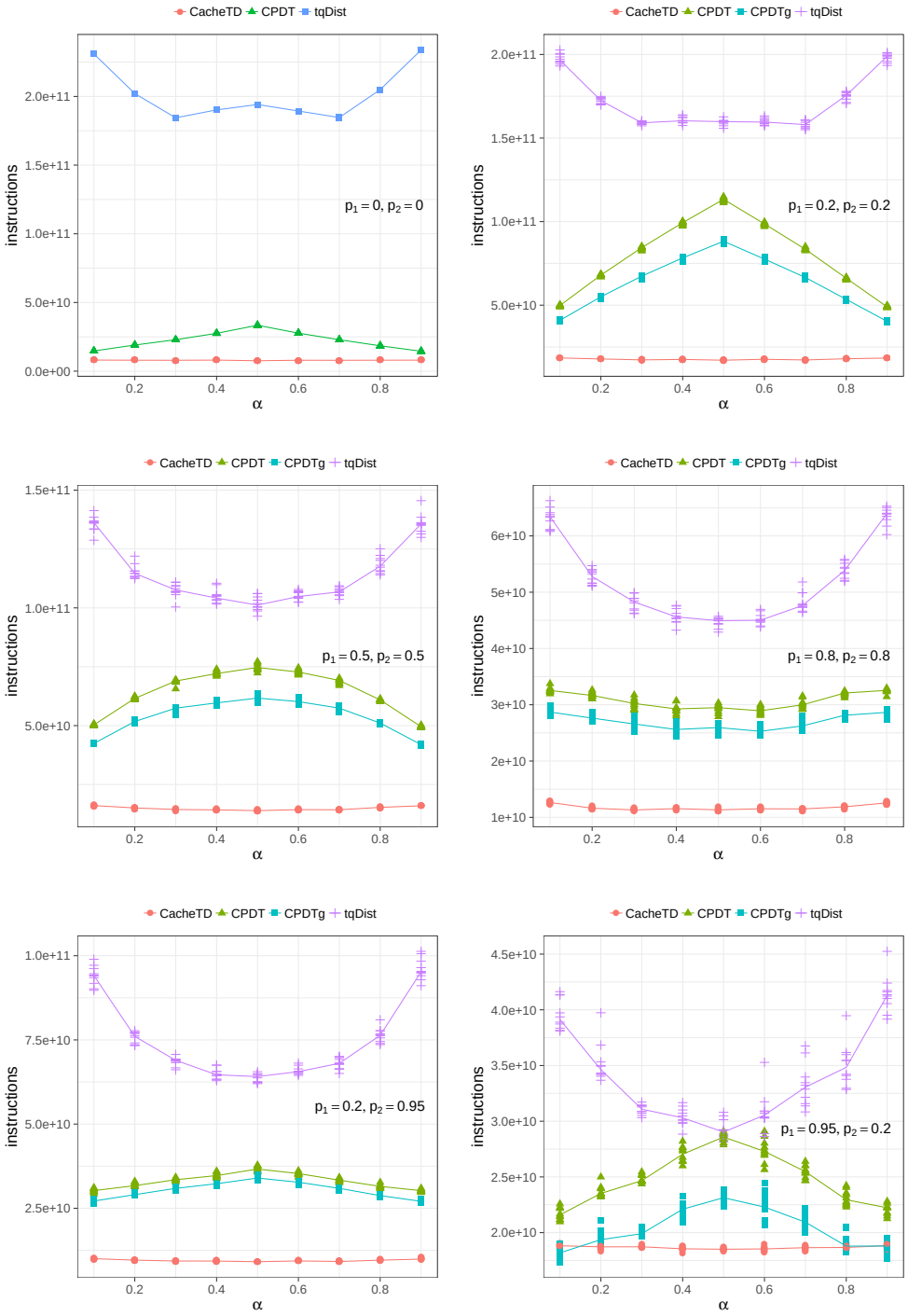


Figure A.12: Skewed model: instructions ($n = 2^{21}$).

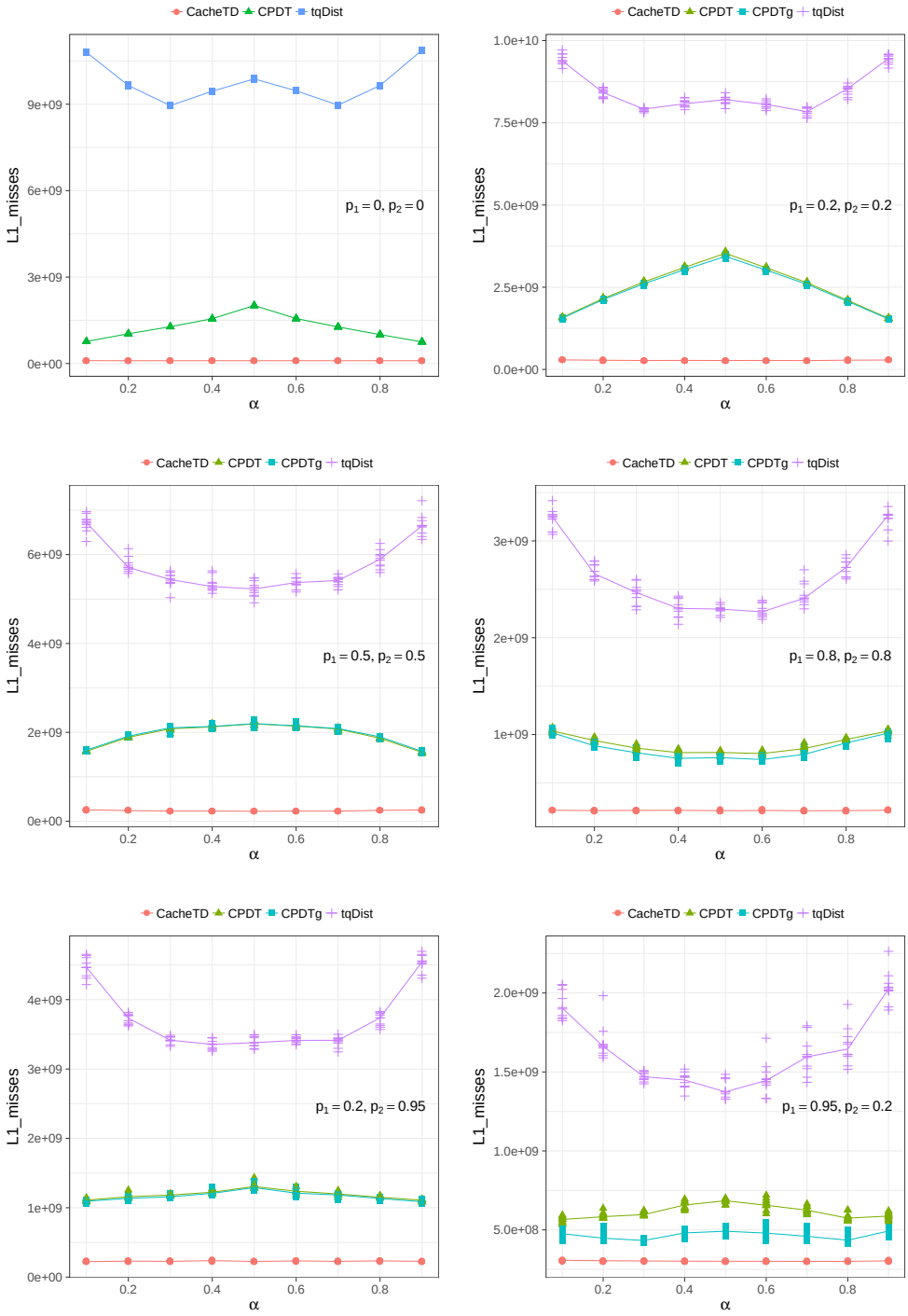


Figure A.13: Skewed model: L1 cache misses ($n = 2^{21}$).

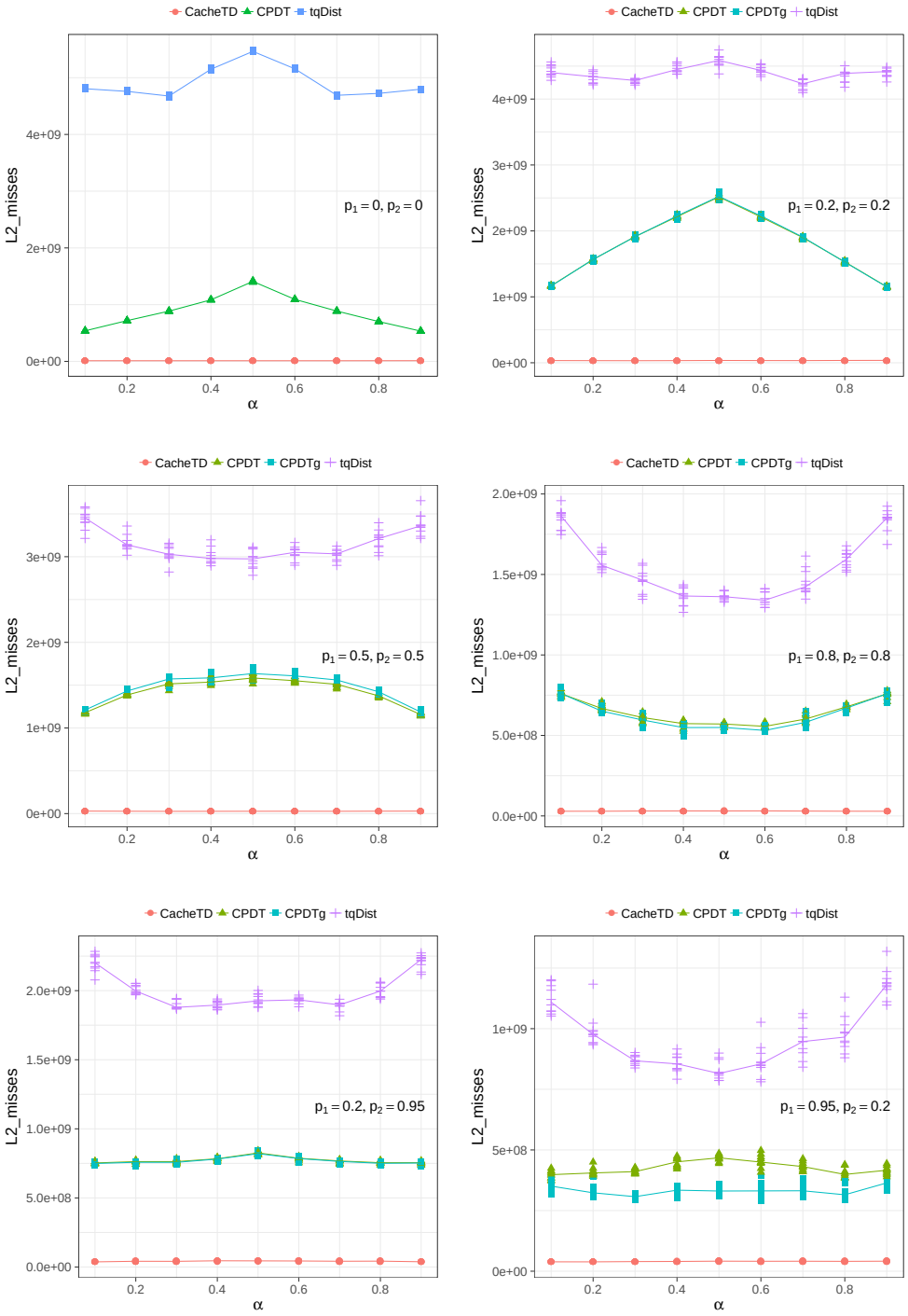


Figure A.14: Skewed model: L2 cache misses ($n = 2^{21}$).

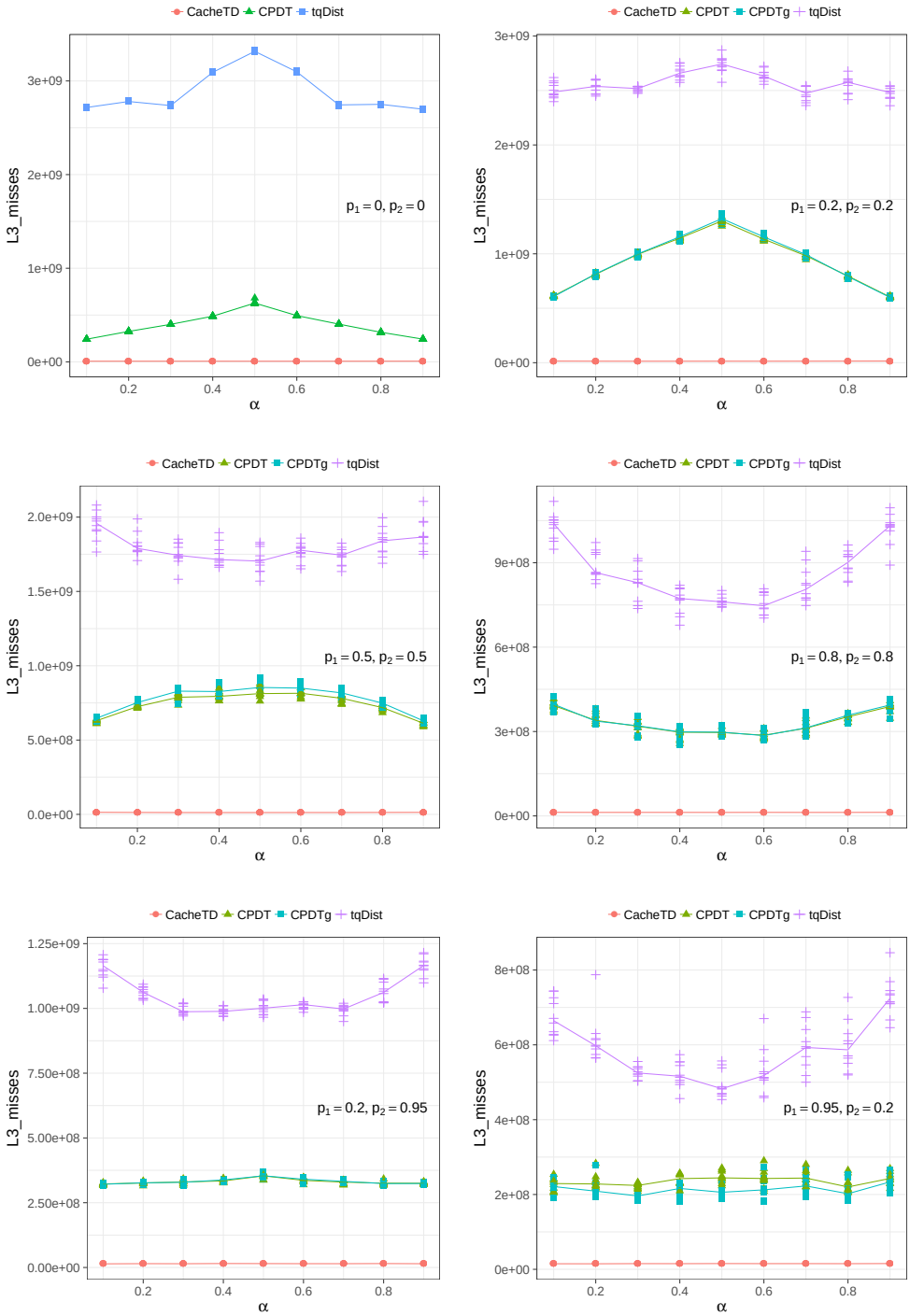


Figure A.15: Skewed model: L3 cache misses ($n = 2^{21}$).

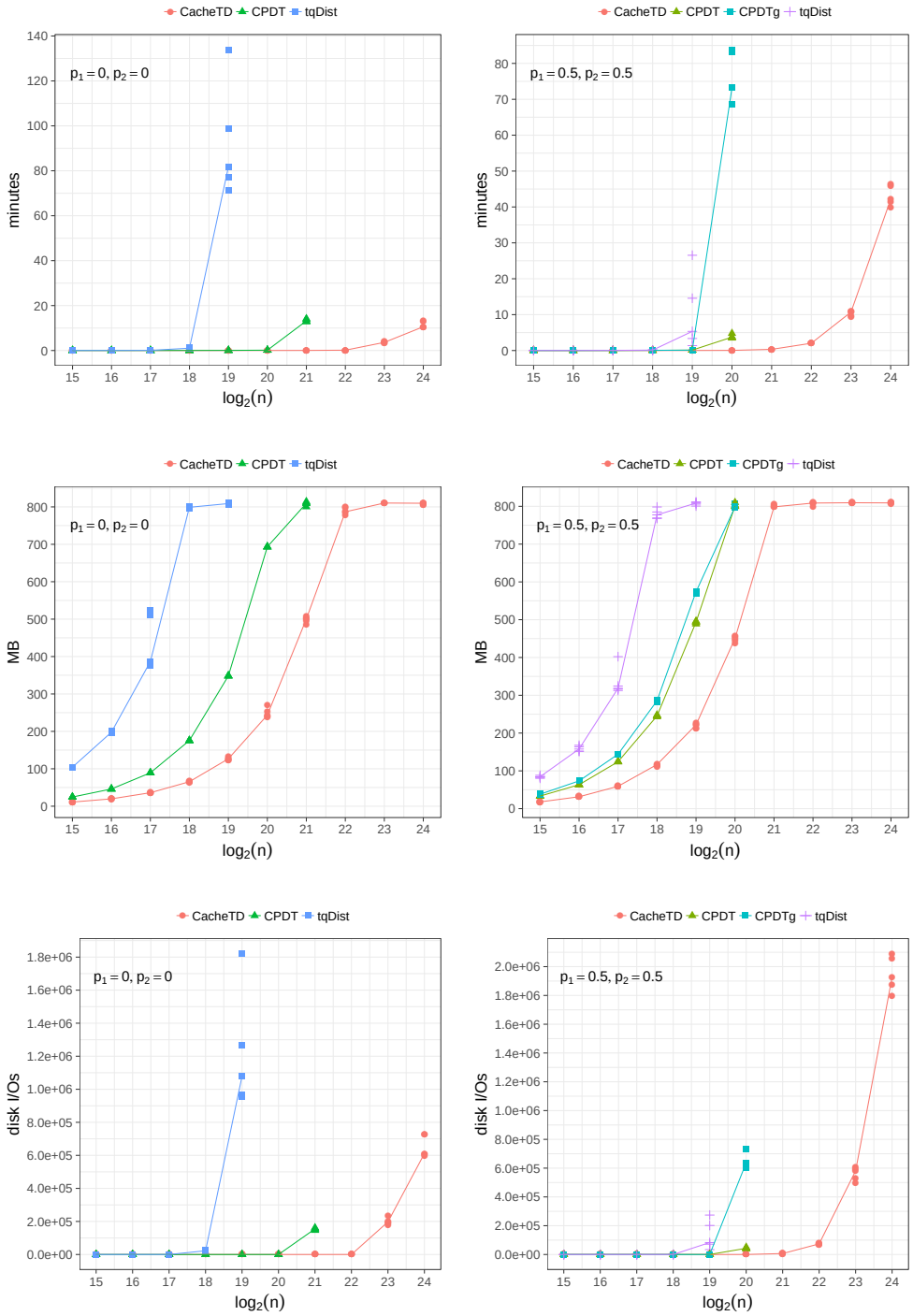


Figure A.16: Random model: I/O experiments.

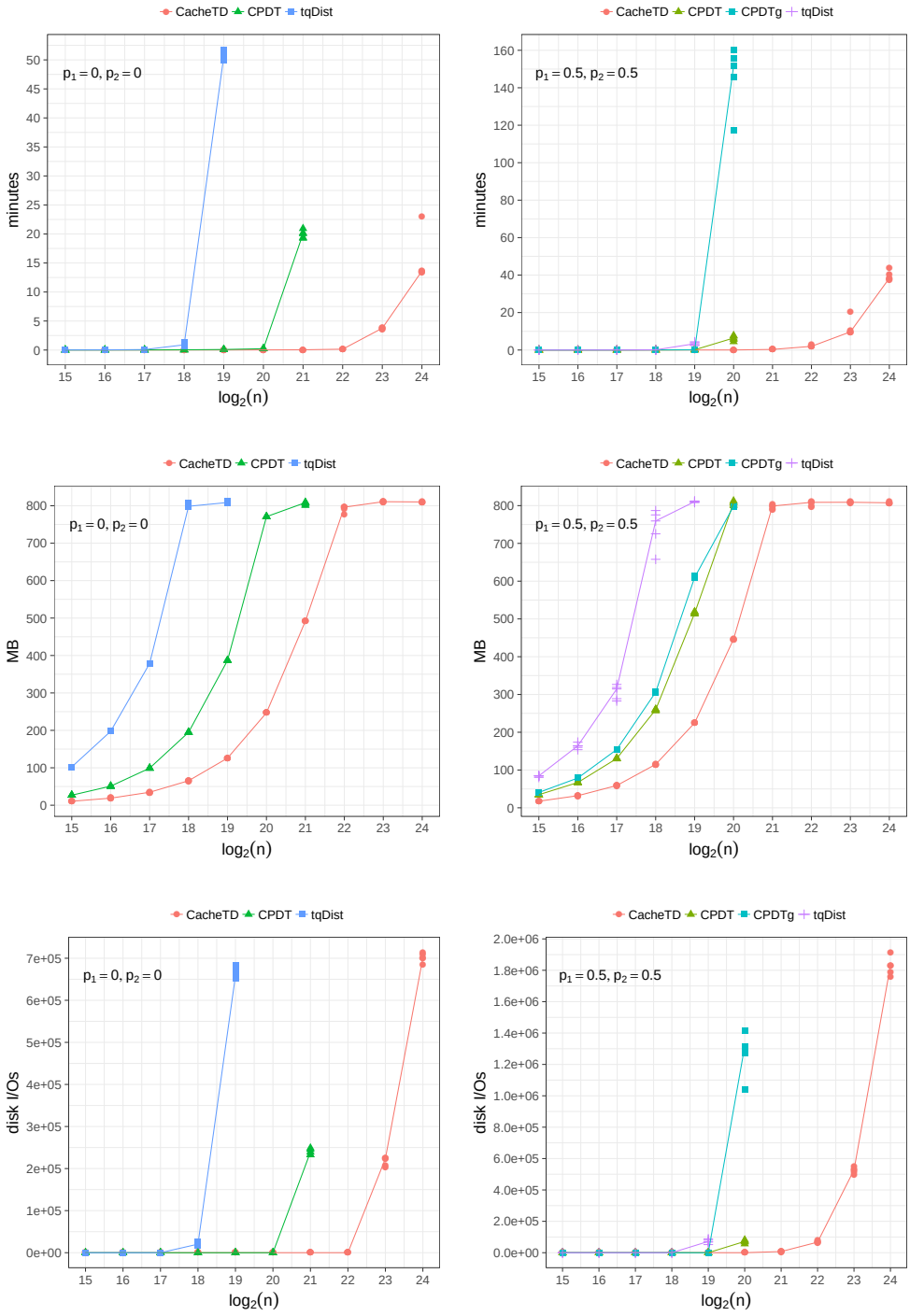


Figure A.17: Skewed model: I/O experiments with $\alpha = 0.5$.

Appendix B

Additional Algorithms and Experiments for Chapter 3

Algorithm 7 Checking if there exists a fan triplet $x|y|z$ consistent with B, assuming that the lowest common ancestor of every pair (x, y), (x, z), and (y, z) in T_i corresponds to B.

1: **procedure** IsFanInBlock($x|y|z$, N_i, B, C_B^f)
2: For every $l \in \{x, y, z\}$ we let $p_l = p_B(l)$, $p'_l = p'_B(l)$, $q_l = q_B(l)$, and h_l
3: be the height of q_l in N_i.
4: **if** $p_x = p_y = p_z$ **then**
5: **if** $h_x = h_y = h_z$ **then return true** ▷ e.g., $a_5|a_6|a_7$ in Fig. 3.4.
6: **if** $((h_x = h_y) \wedge (h_x > h_z)) \vee ((h_x = h_z) \wedge (h_x > h_y)) \vee ((h_y = h_z) \wedge (h_y > h_x))$ **then return true** ▷ e.g., $a_5|a_6|a_8$ in Fig. 3.4.
7: **if** $h_x \neq h_y \neq h_z$ **then return false** ▷ e.g., $a_{13}|a_{14}|a_{20}$ in Fig. 3.4.
8: **if** $((p_x = p_y) \wedge (p_x \neq p_z)) \vee ((p_x = p_z) \wedge (p_x \neq p_y)) \vee ((p_y = p_z) \wedge (p_y \neq p_x))$ **then** w.l.o.g. assume $(p_x = p_y) \wedge (p_x \neq p_z)$
9: **if** $h_x = h_y$ **then**
10: **if** $\exists s \rightsquigarrow (p'_x, p_x, p_z)$ in C_B^f **then**
11: **return true** ▷ e.g., $a_8|a_9|a_{15}$ in Fig. 3.4.
12: **else return false** ▷ e.g., $a_8|a_9|a_{11}$ in Fig. 3.4.
13: **else return false** ▷ e.g., $a_7|a_8|a_{15}$ in Fig. 3.4.
14: **if** $p_x \neq p_y \neq p_z$ **then**
15: **if** $\exists s \rightsquigarrow (p_x, p_y, p_z)$ in C_B^f **then**
16: **return true** ▷ e.g., $a_8|a_{11}|a_{16}$ in Fig. 3.4.
17: **else return false** ▷ e.g., $a_{14}|a_{16}|a_{17}$ in Fig. 3.4.

Algorithm 8 Checking if there exists a resolved triplet $xy|z$ consistent with B, assuming that the lowest common ancestor of every pair (x, y), (x, z), and (y, z) in T_i corresponds to B.

procedure IsResolvedInBlock($xy|z$, N_i, B, C_B^r, C_B^f)
  For every $l \in \{x, y, z\}$ we let $p_l = p_B(l)$, $p'_l = p'_B(l)$, $q_l = q_B(l)$, and h_l be the height of q_l in N_i.
  if $p_x = p_y = p_z$ **then**
    if $(h_z > h_x) \wedge (h_z > h_y)$ **then return true** ▷ e.g., $a_8a_9|a_6$ in Fig. 3.4.
    else return false ▷ e.g., $a_8a_6|a_9$ in Fig. 3.4.
  if $p_x = p_y \wedge p_x \neq p_z$ **then**
    if $\exists s \rightsquigarrow (p_z, p_x, p'_x)$ in C_B^r **then**
      return true ▷ e.g., $a_5a_8|a_{17}$ in Fig. 3.4.
    else return false ▷ e.g., $a_5a_8|a_{15}$ in Fig. 3.4.
  if $((p_x = p_z) \wedge (p_x \neq p_y)) \vee ((p_y = p_z) \wedge (p_y \neq p_x))$ **then**
    w.l.o.g. assume $(p_x = p_z) \wedge (p_x \neq p_y)$
    if $h_z > h_x$ **then**
      if $\exists s \rightsquigarrow (p_x, p'_x, p_y)$ in C_B^f **then**
        return true ▷ e.g., $a_{14}a_{17}|a_{13}$ in Fig. 3.4.
      else return false ▷ e.g., $a_{14}a_{16}|a_{13}$ in Fig. 3.4.
    else return false ▷ e.g., $a_{14}a_{17}|a_{20}$ in Fig. 3.4.
  if $p_x \neq p_y \neq p_z$ **then**
    if $\exists s \rightsquigarrow (p_z, p_x, p_y)$ in C_B^r **then**
      return true ▷ e.g., $a_{12}a_{13}|a_{18}$ in Fig. 3.4.
    else return false ▷ e.g., $a_{12}a_{18}|a_{13}$ in Fig. 3.4.

Algorithm 9 Checking if the fan triplet $x|y|z$ is consistent with N_i.

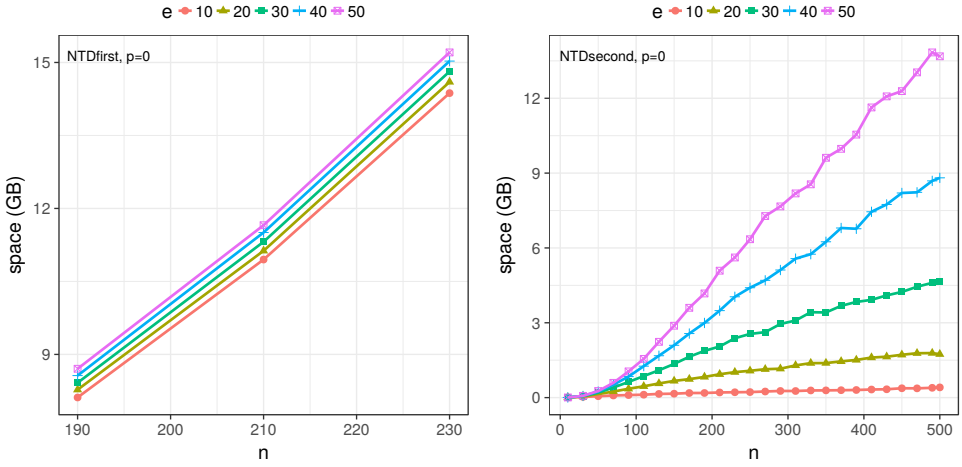
1: **procedure** IsFan($x|y|z$, N_i, T_i)
2: **if** $x|y|z$ is consistent with T_i **then**
3: $w \leftarrow lca(x, y, z)$
4: **if** w is the root of T_i **then return true** ▷ e.g., $a_{23}|a_9|a_{20}$ in
5: Fig. 3.5.
6: **else** let B be the block of w
7: **return** IsFanInBlock($x|y|z$, N_i, B, C_B^f) ▷ e.g., $a_3|a_9|a_{12}$
8: in Fig. 3.5.
9: **if** $xy|z$ **or** $xz|y$ **or** $yz|x$ is consistent with T_i **then**
10: Assume w.l.o.g. that $xy|z$ is consistent with T_i
11: $w \leftarrow lca(x, y)$
12: $\mu \leftarrow lca(x, z)$
13: **if** μ is the parent of w in T_i **then**
14: Let B be the block of w
15: Let M be the block of μ
16: **if** $p_B(x) = p_B(y)$ **then**
17: **return false** ▷ e.g., $a_2|a_3|a_4$ in Fig. 3.5.
18: **else**
19: **if** μ is the root of T_i **then**
20: **if** $\exists s \rightsquigarrow (r', p_B(x), p_B(y))$ in C_B^f **then**
21: **return true** ▷ e.g., $a_1|a_{11}|a_{15}$ in Fig. 3.5.
22: **else return false** ▷ e.g., $a_{11}|a_{13}|a_{15}$ in Fig. 3.5.
23: **else**
24: **if** $p_F(x) = p_F(z)$ **then**
25: **if** $h_F(z) \leq h_F(x)$ **then**
26: **if** $\exists s \rightsquigarrow (r', p_B(x), p_B(y))$ in C_B^f **then**
27: **return true** ▷ e.g., $a_1|a_4|a_8$ in Fig. 3.5.
28: **else return false** ▷ e.g., $a_1|a_{24}|a_8$ in Fig. 3.5.
29: **else return false** ▷ e.g., $a_1|a_4|a_{21}$ in Fig. 3.5.
30: **else**
31: **if** $\exists s \rightsquigarrow (r', p_B(x), p_B(y))$ in C_B^f
32: **and** $\exists s \rightsquigarrow (p_F(x), p'_F(x), p_F(z))$ in C_F^f **then**
33: **return true** ▷ e.g., $a_1|a_4|a_9$ in Fig. 3.5.
34: **else return false** ▷ e.g., $a_1|a_4|a_{12}$ in Fig. 3.5.
35: **else return false** ▷ e.g., $a_2|a_4|a_{13}$ in Fig. 3.5.

Algorithm 10 Checking if the resolved triplet $xy|z$ is consistent with N_i.

procedure IsResolved($xy|z$, N_i, T_i)
  if $x|y|z$ is consistent with T_i **then**
    $w \leftarrow lca(x, y, z)$
    if w is the root of T_i **then return false** ▷ e.g., $a_{23}a_9|a_{20}$ in Fig. 3.5.
    else let B be the block of w
      return IsResolvedInBlock($xy|z$, N_i, B, C_B^r) ▷ e.g., $a_1a_9|a_{12}$ in Fig. 3.5.
  if $xy|z$ **or** $xz|y$ **or** $yz|x$ is consistent with T_i **then**
    Assume w.l.o.g. that $xy|z$ is consistent with T_i
    $w \leftarrow lca(x, y)$
    $\mu \leftarrow lca(x, z)$
    if μ is the parent of w in T_i **then**
      Let B be the block of w
      Let F be the block of μ
      if $p_B(x) = p_B(y)$ **then**
        return true ▷ e.g., $a_2a_3|a_4$ in Fig. 3.5.
      else
        if μ is the root of T_i **then**
          if $\exists s \rightsquigarrow (r', p_B(x), p_B(y))$ in C_B^r **then**
            return true ▷ e.g., $a_{11}a_{13}|a_{15}$ in Fig. 3.5.
          else return false ▷ e.g., $a_1a_{11}|a_{15}$ in Fig. 3.5.
        else
          if $p_F(x) = p_F(z)$ **then**
            if $h_F(z) \leq h_F(x)$ **then**
              if $\exists s \rightsquigarrow (r', p_B(x), p_B(y))$ in C_B^r **then**
                return true ▷ e.g., $a_1a_4|a_8$ in Fig. 3.5.
              else return false ▷ e.g., $a_1a_{25}|a_{22}$ in Fig. 3.5.
            else return true ▷ e.g., $a_1a_4|a_{21}$ in Fig. 3.5.
          else
            if $\exists s \rightsquigarrow (r', p_B(x), p_B(y))$ in C_B^r
                or $\exists s \rightsquigarrow (p_F(z), p_F(x), p'_F(x))$ in C_F^r **then**
                return true ▷ e.g., $a_1a_4|a_{12}$ in Fig. 3.5.
            else return false ▷ e.g., $a_1a_{25}|a_{26}$ in Fig. 3.5.
    else return true ▷ e.g., $a_2a_4|a_{13}$ in Fig. 3.5.

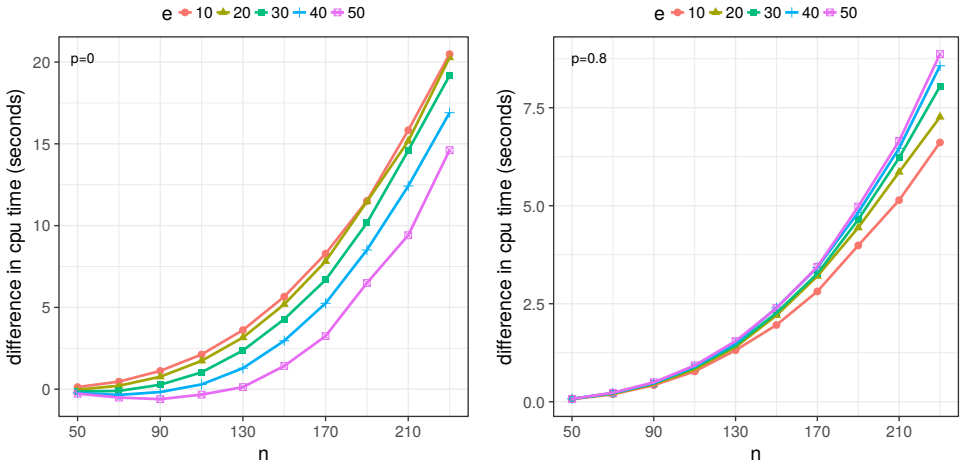
Algorithm 11 Computing $D(N_1, N_2)$ using the data structures from Section 3.3.

procedure PREPROCESSING(N_1, N_2) ▷ Building the data structures
  for $i \in \{1, 2\}$ **do**
    build T_i using Lemma 10
    build a $n \times n$ table to support *lca* queries between pairs of leaves in T_i
    build all contracted block networks of N_i as shown in Lemma 12
    for every contracted block network C_B, build the C_B^f and C_B^r graphs from Lemmas 6 and 7 respectively, as well as the two tables A_B^f and A_B^r, like A^f and A^r from Algorithm 2 to be able to answer path existence queries in the algorithms 7, 8, 9, and 10 in O(1) time.
procedure $S_f(N_1, N_2)$ ▷ Finding the shared fan triplets
  $sharedFan = 0$
  for $x, y, z \in \Lambda$ **do**
    if IsFan($x|y|z$, N_1) $\wedge$ IsFan($x|y|z$, N_2) **then**
      $sharedF = sharedF + 1$
  return *sharedF*
procedure $S_r(N_1 N_2)$ ▷ Finding the shared resolved triplets
  $sharedR = 0$
  for $x, y, z \in \Lambda$ **do**
    if IsResolved($xy|z$, N_1) $\wedge$ IsResolved($xy|z$, N_2) **then**
      $sharedR = sharedR + 1$
    if IsResolved($xz|y$, N_1) $\wedge$ IsResolved($xz|y$, N_2) **then**
      $sharedR = sharedR + 1$
    if IsResolved($yz|x$, N_1) $\wedge$ IsResolved($yz|x$, N_2) **then**
      $sharedR = sharedR + 1$
  return *sharedR*
procedure $S(N_1, N_2)$ ▷ Finding the shared triplets
  return $S_f(N_1, N_2) + S_r(N_1, N_2)$
procedure $D(N_1 = (V_1, E_1), N_2 = (V_2, E_2))$ ▷ Computing $D(N_1, N_2)$
  PREPROCESSING(N_1, N_2)
  return $S(N_1, N_1) + S(N_2, N_2) - 2S(N_1, N_2)$



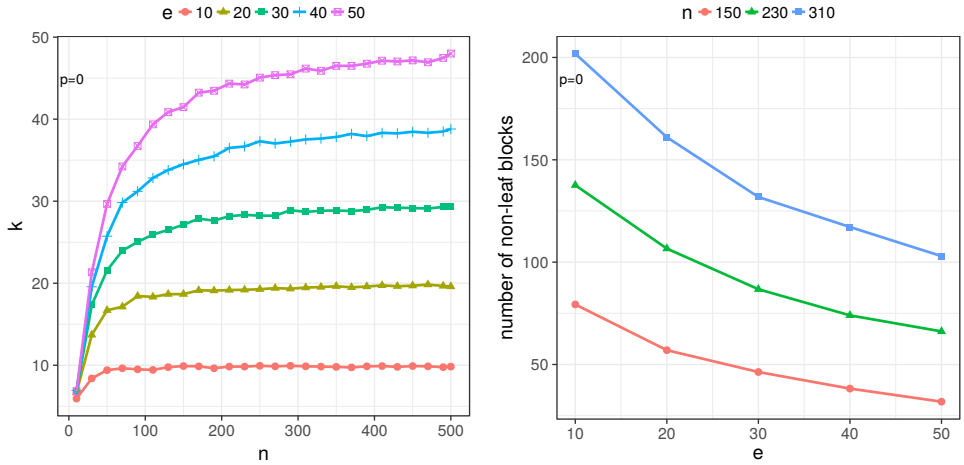
(a) (b)

Figure B.1: Simulated Datasets: Space usage of the two algorithms for different values of e and when $p = 0$, as shown by the *Maximum Resident Size* parameter when calling the executable of each algorithm with `/usr/bin/time -v`.



(a) (b)

Figure B.2: Simulated Datasets: Subtracting the running time of `NTDsecond` from the running time of `NTDfirst` for different values of e and when $p = \{0, 0.8\}$. (a) For $n = 110$ and $e = 50$, `NTDfirst` is faster than `NTDsecond`. (b) When p is large, the number of possible edges we can add (determined by e) is small.



(a) (b)

Figure B.3: Simulated Datasets: The effect of e on the parameter k and the amount of non-leaf blocks.

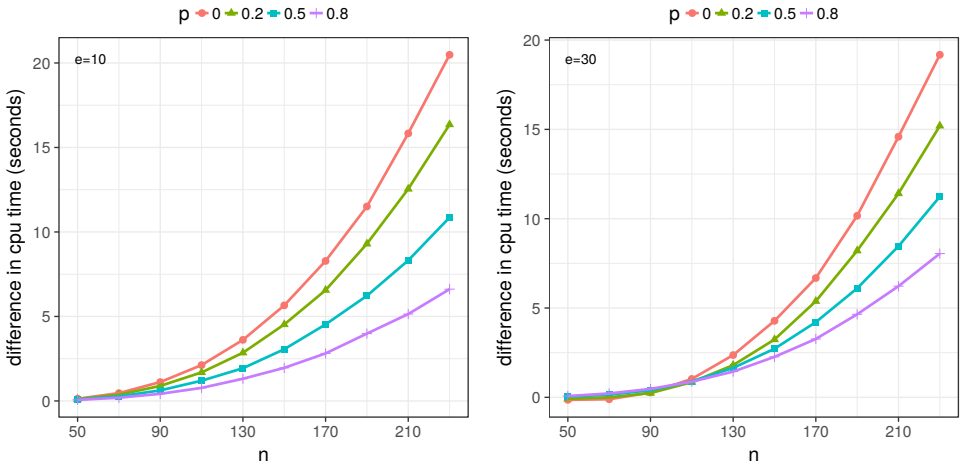


Figure B.4: Simulated Datasets: The effect of p for $e \in \{10, 30, 50\}$ when subtracting the running time of `NTDsecond` from the running time of `NTDfirst`. As expected, the larger the value of p, the closer become the running times of the two algorithms.

Appendix C

Additional Algorithms and Experiments for Chapter 4

Algorithm 12 The $O(q|\mathcal{R}|)$-time algorithm for q-MAXRTC when $q > 2$ (Theorem 7)

procedure PRQ($xy|z$, q, k) ▷ Computing $Pr[xy|z \in rt(T)|N_i]$
  if $x \leftarrow \emptyset$ **and** $y \leftarrow \emptyset$ **and** $z \leftarrow \emptyset$ **then** return $1/3 - 4/(3(q+1)^2)$
  if $x \leftarrow u$ **and** $y \leftarrow \emptyset$ **and** $z \leftarrow \emptyset$ **then** return $(u_{\uparrow d} + k)/(k+1)^2$
  if $x \leftarrow \emptyset$ **and** $y \leftarrow v$ **and** $z \leftarrow \emptyset$ **then** return $(v_{\uparrow d} + k)/(k+1)^2$
  if $x \leftarrow \emptyset$ **and** $y \leftarrow \emptyset$ **and** $z \leftarrow w$ **then** return $(k + 2w_{\uparrow s})/(k+1)^2$
  if $x \leftarrow u$ **and** $y \leftarrow v$ **and** $z \leftarrow \emptyset$ **then**
    $p = lca(u, v)$
    return $(k + 1 - p_{\downarrow})/(k+1)$
  if $x \leftarrow u$ **and** $y \leftarrow \emptyset$ **and** $z \leftarrow w$ **then**
    $p = lca(u, w)$
    if $p == u$ **then** return 0
    $m =$ `child of p whose subtree contains` u
    return $m_{\downarrow}/(k+1)$
  if $x \leftarrow \emptyset$ **and** $y \leftarrow v$ **and** $z \leftarrow w$ **then**
    $p = lca(v, w)$
    if $p == v$ **then** return 0
    $m =$ `child of p whose subtree contains` v
    return $m_{\downarrow}/(k+1)$
  if $x \leftarrow u$ **and** $y \leftarrow v$ **and** $z \leftarrow w$ **then**
    return $(lca(x, z) == lca(y, z)$ **and** $lca(x, y) \neq lca(x, z))$

procedure Q-MAXRTC($\mathcal{R}$, q, k) ▷ The main procedure
  $prev = |\mathcal{R}|(1/3 - 4/(3(q+1)^2))$ ▷ Storing $E[W|N_0]$, where $N_0 = \emptyset$
  for $i = 1$ `to` n **do**
    for $m = 1$ `to` $k+1$ **do**
      $nValue[m] = prev$
    for $j = 1$ `to` $|\mathcal{R}[x_i]|$ **do**
      for $m = 1$ `to` $k+1$ **do**
        $x_i \leftarrow \emptyset$
        $nValue[m] = nValue[m] - \text{PRQ}(\mathcal{R}[x_i][j])$
        $x_i \leftarrow u_m$
        $nValue[m] = nValue[m] + \text{PRQ}(\mathcal{R}[x_i][j])$
    $x_i \leftarrow u_1$
    $curBest = nValue[1]$
    for $j = 2$ `to` $k+1$ **do**
      if $nValue[j] > curBest$ **then**
        $curBest = nValue[j]$
        $x_i \leftarrow u_j$
        $prev = curBest$

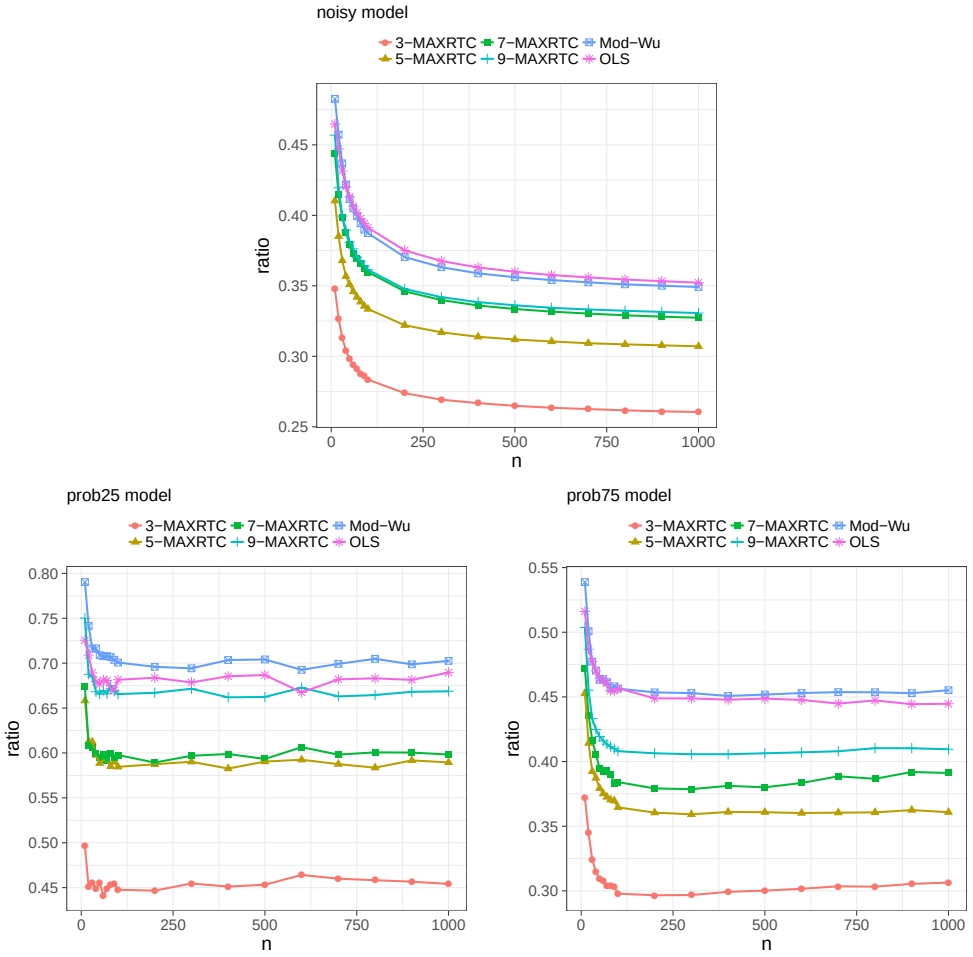


Figure C.1: Performance of $\{3, 5, 7, 9\}$-MAXRTC with the performance of `OLS` and `Mod-Wu` on the noisy, `prob25`, and `prob75` models. Every data point corresponds to the mean of 100 runs. The larger the value of q, the better the performance of q-MAXRTC. For $q = 9$, the performance approaches that of `OLS` & `Mod-Wu`. Observe that in the noisy model, the performance of q-MAXRTC approaches the theoretical guarantee.

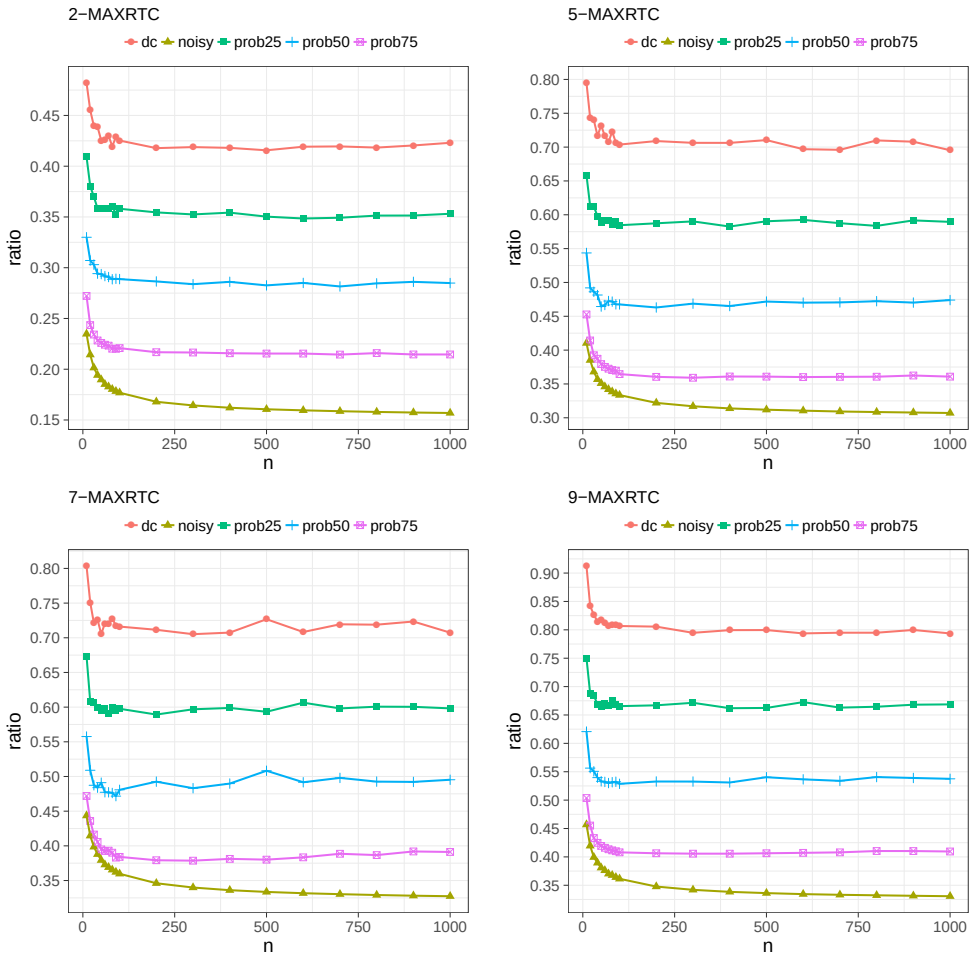


Figure C.2: Performance of {2,5,7,9}-MAXRTC on the different simulated models. Every data point corresponds to the mean of 100 runs.

Algorithm 13 O(q)-time algorithm for computing the counters of every internal node u in T_1 (see Lemma 22).

procedure CountersHelper(u, C, R, B, G)
  Let $R = [a_{\text{red}}, \ldots, a'_{\text{red}}]$, $B = [a_{\text{blue}}, \ldots, a'_{\text{blue}}]$, $G = [a_{\text{green}}, \ldots, a'_{\text{green}}]$
  Let u_l denote the total number of leaves in $T_1(u)$
  totalWhite $= n-(a'_{\text{red}}-a_{\text{red}}+1)-(a'_{\text{blue}}-a_{\text{blue}}+1)-(a'_{\text{green}}-a_{\text{green}}+1)$
  if u has no children that are internal nodes **then** ▷ Base case
    for every color $i \in \{\text{red}, \text{blue}, \text{green}\}$ **do**
      $u_i = C[u][a'_i] - C[u][a_i - 1]$
      $u_{iI} = 0, u_{iL} = u_i$
    $u_{\text{white}} = \text{totalWhite} - u_l + u_{\text{red}} + u_{\text{blue}} + u_{\text{green}}$
    $u_{\text{red,blue}} = u_{\text{red,green}} = u_{\text{blue,green}} = 0$
    $u_{\text{red,blue,green}} = 0$
    return
  Let I denote the set of children of u that are internal nodes in T_1
  for every node w in I **do**
    CountersHelper(w, C, R, B, G)
  pick some node w from I
  for every color $i \in \{\text{red}, \text{blue}, \text{green}\}$ **do** ▷ Base case for dynamic programming
    $u_i = C[u][a'_i] - C[u][a_i - 1]$
    $u_{iI} = w_i, u_{iL} = u_i$
    for every node w' in I **do**
      $u_{iL} = u_{iL} - (C[w'][a'_i] - C[w'][a_i - 1])$
  $u_{\text{white}} = \text{totalWhite} - u_l + u_{\text{red}} + u_{\text{blue}} + u_{\text{green}}$
  $u_{\text{red,blue}} = u_{\text{red,green}} = u_{\text{blue,green}} = 0$
  $u_{\text{red,blue,green}} = 0$
  remove w from I
  while I is not empty **do**
    pick some node w from I
    $u_{\text{red,blue,green}} = u_{\text{red,blue,green}} + u_{\text{red,blue}} \cdot w_{\text{green}} + u_{\text{red,green}} \cdot w_{\text{blue}} + u_{\text{blue,green}} \cdot w_{\text{red}}$
    $u_{\text{red,blue}} = u_{\text{red,blue}} + u_{\text{red}I} \cdot w_{\text{blue}} + u_{\text{blue}I} \cdot w_{\text{red}}$
    $u_{\text{red,green}} = u_{\text{red,green}} + u_{\text{red}I} \cdot w_{\text{green}} + u_{\text{green}I} \cdot w_{\text{red}}$
    $u_{\text{blue,green}} = u_{\text{blue,green}} + u_{\text{blue}I} \cdot w_{\text{green}} + u_{\text{green}I} \cdot w_{\text{blue}}$
    $u_{\text{red}I} = u_{\text{red}I} + w_{\text{red}}$
    $u_{\text{blue}I} = u_{\text{blue}I} + w_{\text{blue}}$
    $u_{\text{green}I} = u_{\text{green}I} + w_{\text{green}}$
    remove w from I
procedure Counters(T_1, C, R, B, G)
  let r be the root of T_1
  CountersHelper(r, C, R, B, G)

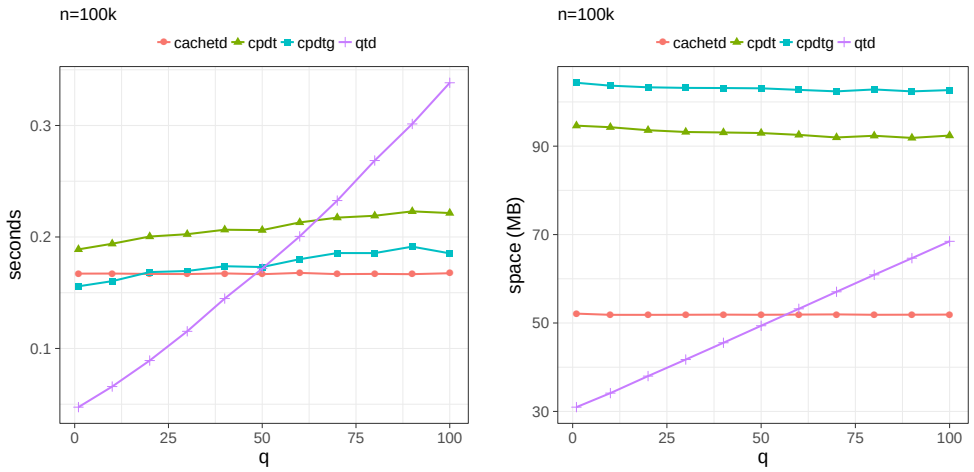


Figure C.3: Model B: Running time and space usage of the different triplet distance algorithms when $n = 10^5$.

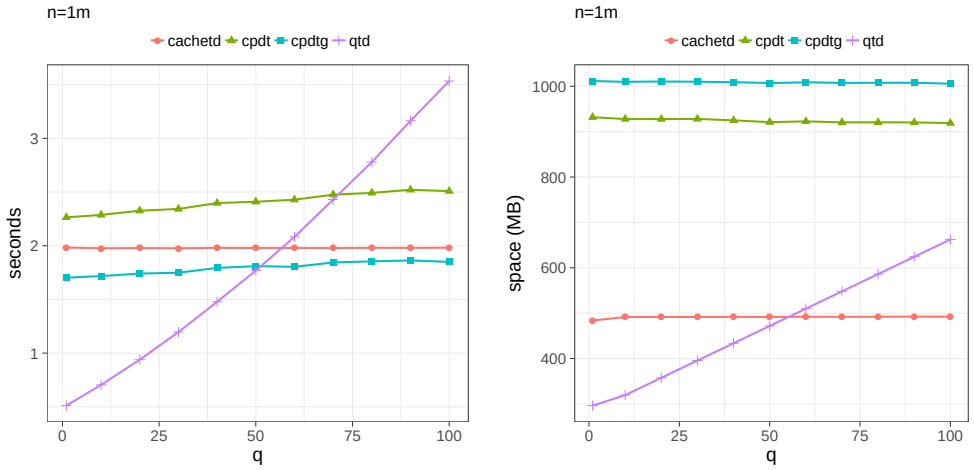


Figure C.4: Model B: Running time and space usage of the different triplet distance algorithms when $n = 10^6$.

Bibliography

[1] A. Aggarwal and J. S. Vitter. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM*, 31(9):1116–1127, 1988. 4, 30

[2] A. V. Aho, Y. Sagiv, T. G. Szymanski, and J. D. Ullman. Inferring a Tree from Lowest Common Ancestors with an Application to the Optimization of Relational Expressions. *SIAM Journal on Computing*, 10(3):405–421, 1981. 24, 93, 114

[3] P. Alimonti. New Local Search Approximation Techniques for Maximum Generalized Satisfiability Problems. *Information Processing Letters*, 57 (3):151–158, 1996. 98

[4] L. Arge. *Efficient External-Memory Data Structures and Applications.* PhD thesis, Aarhus University, Denmark, 1996. 4

[5] M. S. Bansal, J. Dong, and D. Fernández-Baca. Comparing and Aggregating Partially Resolved Trees. *Theoretical Computer Science*, 412(48): 6634–6652, 2011. 7, 10, 31, 32, 33, 64

[6] N. H. Barton. The Role of Hybridization in Evolution. *Molecular Ecology*, 10(3):551–568, 2001. 14

[7] M. A. Bender and M. Farach-Colton. The LCA Problem Revisited. In *LATIN 2000: Theoretical Informatics*, pages 88–94. Springer Berlin Heidelberg, 2000. 6, 21, 103

[8] V. Berry and O. Gascuel. Inferring Evolutionary Trees with Strong Combinatorial Evidence. *Theoretical Computer Science*, 240(2):271–298, 2000. 30

[9] O. R. P. Bininda-Emonds. The Evolution of Supertrees. *Trends in Ecology and Evolution*, 19(6):315–322, 2004. 2, 22, 25, 92, 93, 94

[10] M. Bordewich and C. Semple. Computing the Minimum Number of Hybridization Events for a Consistent Evolutionary History. *Discrete Applied Mathematics*, 155(8):914–928, 2007. 86

[11] G. S. Brodal and K. Mampentzidis. Cache Oblivious Algorithms for Computing the Triplet Distance Between Trees. In *25th Annual European Symposium on Algorithms (ESA 2017)*, volume 87 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017. v, 19, 27, 29, 64, 94, 105, 107, 108, 112, 114, 115

[12] G. S. Brodal, R. Fagerberg, C. N. S. Pedersen, T. Mailund, and A. Sand. Efficient Algorithms for Computing the Triplet and Quartet Distance Between Trees of Arbitrary Degree. In *Proceedings of the Twenty-fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1814–1832. Society for Industrial and Applied Mathematics, 2013. 9, 10, 19, 30, 31, 32, 33, 34, 56, 64, 94, 107, 115

[13] D. Bryant. *Building Trees, Hunting for Trees, and Comparing Trees - Theory and Methods in Phylogenetic Analysis*. PhD thesis, University of Canterbury, New Zealand, 1997. 2, 24, 25, 93

[14] A. R. Burmeister. Horizontal Gene Transfer. *Evolution, Medicine, and Public Health*, 2015(1):193–194, 2015. 14

[15] J. Byrka, P. Gawrychowski, K. T. Huber, and S. Kelk. Worst-case Optimal Approximation Algorithms for Maximizing Triplet Consistency Within Phylogenetic Networks. *Journal of Discrete Algorithms*, 8(1): 65–75, 2010. 17, 18, 19, 20, 25, 26, 64, 65, 71, 77, 84, 85, 93, 94, 99

[16] J. Byrka, S. Guillemot, and J. Jansson. New Results on Optimizing Rooted Triplets Consistency. *Discrete Applied Mathematics*, 158(11): 1136–1147, 2010. 26, 93, 94, 103

[17] G. Cardona, F. Rosselló, and G. Valiente. Extended Newick: It Is Time For a Standard Representation of Phylogenetic Networks. *BMC Bioinformatics*, 9(1):532, 2008. 86

[18] G. Cardona, M. Llabres, F. Rossello, and G. Valiente. Metrics for Phylogenetic Networks II: Nodal and Triplets Metrics. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 6(3):454–469, 2009. 14, 88

[19] B. Chor, M. Hendy, and D. Penny. Analytic Solutions for Three Taxon ML Trees with Variable Rates Across Sites. *Discrete Applied Mathematics*, 155(6):750–758, 2007. 92

[20] C. Choy, J. Jansson, K. Sadakane, and W.-K. Sung. Computing the Maximum Agreement of Phylogenetic Networks. *Theoretical Computer Science*, 335(1):93–107, 2005. 63

[21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. 3

[22] J. A. Cotton and R. D. Page. Rates and Patterns of Gene Duplication and Loss in the Human Genome. *Proceedings of Biological Sciences*, 272 (1560):277–283, 2005. 14

[23] D. E. Critchlow, D. K. Pearl, and C. L. Qian. The Triples Distance for Rooted Bifurcating Phylogenetic Trees. *Systematic Biology*, 45(3): 323–334, 1996. 6, 7, 9, 10, 31, 32, 33, 64

[24] W. H. E. Day. Optimal Algorithms for Comparing Trees with Labeled Leaves. *Journal of Classification*, 2(1):7–28, 1985. 30

[25] A. J. Dobson. Comparing the Shapes of Trees. In *Combinatorial Mathematics III*, pages 95–100. Springer Berlin Heidelberg, 1975. 1, 5, 30, 31, 62, 64

[26] G. F. Estabrook, F. R. McMorris, and C. A. Meacham. Comparison of Undirected Phylogenetic Trees Based on Subtrees of Four Evolutionary Units. *Systematic Zoology*, 34(2):193–200, 1985. 5, 30, 62

[27] J. Felsenstein. *Inferring Phylogenies*. Sinauer Associates, Inc., Sunderland, Massachusetts", 2004. 62, 92

[28] C. R. Finden and A. D. Gordon. Obtaining Common Pruned Trees. *Journal of Classification*, 2(1):255–276, 1985. 62

[29] S. Fortune, J. Hopcroft, and J. Wyllie. The Directed Subgraph Homeomorphism Problem. *Theoretical Computer Science*, 10(2):111–121, 1980. 18, 19, 64, 65

[30] A. R. Francis and M. Steel. Which Phylogenetic Networks are Merely Trees with Additional Arcs? *Systematic Biology*, 64(5):768–777, 2015. 85

[31] M. L. Fredman and D. E. Willard. BLASTING Through the Information Theoretic Barrier with FUSION TREES. In *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*, STOC '90, pages 1–7. ACM, 1990. 3

[32] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-Oblivious Algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–297. IEEE Computer Society, 1999. 4, 30

[33] P. Gambette and K. T. Huber. On Encodings of Phylogenetic Networks of Bounded Level. *Journal of Mathematical Biology*, 65(1):157–180, 2012. 1, 14, 16, 62, 63, 88

[34] L. Gąsieniec, J. Jansson, A. Lingas, and A. Östlin. On the Complexity of Constructing Evolutionary Trees. *Journal of Combinatorial Optimization*, 3(2):183–197, 1999. 25, 26, 93, 94, 103

[35] C. Giuseppe and I. Giuseppe. Algorithm Engineering. *ACM Computing Surveys*, 31(3es):3, 1999. 5

[36] D. Gusfield, S. Eddhu, and C. Langley. Optimal, Efficient Reconstruction of Phylogenetic Networks with Constrained Recombination. *Journal of Bioinformatics and Computational Biology*, 2(1):173–213, 2004. 16, 63

[37] J. Håstad. Some Optimal Inapproximability Results. *Journal of the ACM*, 48(4):798–859, 2001. 96

[38] J. Hein, T. Jiang, L. Wang, and K. Zhang. On the Complexity of Comparing Evolutionary Trees. *Discrete Applied Mathematics*, 71(1):153–169, 1996. 62

[39] M. R. Henzinger, V. King, and T. Warnow. Constructing a Tree from Homeomorphic Subtrees, with Applications to Computational Evolutionary Biology. *Algorithmica*, 24(1):1–13, 1999. 92

[40] M. K. Holt, J. Johansen, and G. S. Brodal. On the Scalability of Computing Triplet and Quartet Distances. In *Proceedings of the Meeting on Algorithm Engineering and Expermiments*, pages 9–19. Society for Industrial and Applied Mathematics, 2014. 9, 32

[41] J. Hopcroft and R. Tarjan. Algorithm 447: Efficient Algorithms for Graph Manipulation. *Communications of the ACM*, 16(6):372–378, 1973. 75, 78

[42] L. A. Hug, B. J. Baker, K. Anantharaman, C. T. Brown, A. J. Probst, C. J. Castelle, C. N. Butterfield, A. W. Hernsdorf, Y. Amano, K. Ise, Y. Suzuki, N. Dudek, D. A. Relman, K. M. Finstad, R. Amundson, B. C. Thomas, and J. F. Banfield. A New View of the Tree of Life. *Nature Microbiology*, 1:16048, 2016. 105

[43] D. H. Huson and D. Bryant. Application of Phylogenetic Networks in Evolutionary Studies. *Molecular Biology and Evolution*, 23(2):254–267, 2005. 1, 14

[44] J. Jansson and A. Lingas. Computing the Rooted Triplet Distance Between Galled Trees by Counting Triangles. *Journal of Discrete Algorithms*, 25:66–78, 2014. 16, 17, 18, 63, 64

[45] J. Jansson and R. Rajaby. A More Practical Algorithm for the Rooted Triplet Distance. In *Algorithms for Computational Biology*, pages 109–125. Springer International Publishing, 2015. 9, 10, 31, 32, 33, 34, 54, 56, 57, 64

[46] J. Jansson and R. Rajaby. A More Practical Algorithm for the Rooted Triplet Distance. *Journal of Computational Biology*, 24(2):106–126, 2017. 27, 32, 112

[47] J. Jansson, J. H.-K. Ng, K. Sadakane, and W.-K. Sung. Rooted Maximum Agreement Supertrees. *Algorithmica*, 43(4):293–307, 2005. 28, 115

[48] J. Jansson, R. S. Lemence, and A. Lingas. The Complexity of Inferring a Minimally Resolved Phylogenetic Supertree. *SIAM Journal on Computing*, 41(1):272–291, 2012. 2, 24, 25, 93, 96

[49] J. Jansson, R. Rajaby, and W.-K. Sung. An Efficient Algorithm for the Rooted Triplet Distance Between Galled Trees. In *Algorithms for Computational Biology*, pages 115–126. Springer International Publishing, 2017. 17

[50] J. Jansson, R. Rajaby, and W.-K. Sung. Minimal Phylogenetic Supertrees and Local Consensus Trees. *AIMS Medical Science*, 5(medsci-05-02-181): 181, 2018. 93

[51] J. Jansson, K. Mampentzidis, and S. T. Puthiyaveedu. Building a Small and Informative Phylogenetic Supertree. In *19th International Workshop on Algorithms in Bioinformatics (WABI 2019)*, volume 143 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. v, viii, 91

[52] J. Jansson, K. Mampentzidis, and R. Rajaby W.-K. Sung. Computing the Rooted Triplet Distance Between Phylogenetic Networks. In *Combinatorial Algorithms*, pages 290–303. Springer International Publishing, 2019. v, 61

[53] J. Jansson, R. Rajaby, and W.-K. Sung. An Efficient Algorithm for the Rooted Triplet Distance Between Galled Trees. *Journal of Computational Biology, to appear (2019)*, 2019. 17, 19, 64

[54] L. Jetten and L. van Iersel. Nonbinary Tree-Based Phylogenetic Networks. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 1(1):205–217, 2018. 85, 87

[55] V. Kann, S. Khanna, J. Lagergren, and A. Panconesi. On the Hardness of Approximating Max k-Cut and Its Dual. *Chicago Journal of Theoretical Computer Science*, 1997. 26, 95, 96

[56] J. M. Lang, A. E. Darling, and J. A. Eisen. Phylogeny of Bacterial and Archaeal Genomes Using Conserved Genes: Supertrees and Supermatrices. *PLoS ONE*, 8(4):e62510, 2013. 105

[57] K. J. Locey and J. T. Lennon. Scaling Laws Predict Global Microbial Diversity. *Proceedings of the National Academy of Sciences*, 113(21): 5970–5975, 2016. 94

[58] T. Marcussen, L. Heier, A. K. Brysting, B. Oxelman, and K. S. Jakobsen. From Gene Trees to a Dated Allopolyploid Network: Insights from the Angiosperm Genus Viola (Violaceae). *Systematic Biology*, 64(1):84–101, 2015. 85, 87

[59] A. McKenzie and M. Steel. Distributions of Cherries for Two Models of Trees. *Mathematical Biosciences*, 164(1):81–92, 2000. 85, 103, 112

[60] G. W. Moore, M. Goodman, and J. Barnabas. An Iterative Approach from the Standpoint of the Additive Hypothesis to the Dendrogram Problem Posed by Molecular Data Sets. *Journal of Theoretical Biology*, 38(3): 423–457, 1973. 62

[61] L. Nakhleh, T. Warnow, D. Ringe, and S. N. Evans. A Comparison of Phylogenetic Reconstruction Methods on an Indo-European Dataset. *Transactions of the Philological Society*, 103(2):171–192, 2005. 92

[62] D. Penny, E. E. Watson, and M. A. Steel. Trees from Languages and Genes are Very Similar. *Systematic Biology*, 42(3):382–384, 1993. 62

[63] D. F. Robinson. Comparison of Labeled Trees with Valency Three. *Journal of Combinatorial Theory, Series B*, 11(2):105–119, 1971. 62

[64] D. F. Robinson and L. R. Foulds. Comparison of Phylogenetic trees. *Mathematical Biosciences*, 53(1):131–147, 1981. 5, 30, 62

[65] N. Saitou and M. Nei. The Neighbor-Joining Method: A New Method for Reconstructing Phylogenetic Trees. *Molecular Biology and Evolution*, 4(4):406, 1987. 30

[66] A. Sand, G. S. Brodal, R. Fagerberg, C. N. S. Pedersen, and T. Mailund. A practical $O(n \log^2 n)$ time algorithm for computing the triplet distance on binary trees. *BMC Bioinformatics*, 14(2):S18, 2013. 7, 8, 9, 10, 11, 31, 32, 33, 34, 56

[67] A. Sand, M. K. Holt, J. Johansen, R. Fagerberg, G. S. Brodal, C. N. S. Pedersen, and T. Mailund. Algorithms for Computing the Triplet and Quartet Distances for Binary and General Trees. *Biology - Special Issue on Developments in Bioinformatic Algorithms*, 2(4):1189–1209, 2013. 32

[68] A. Sand, M. K. Holt, J. Johansen, G. S. Brodal, T. Mailund, and C. N. S. Pedersen. tqDist: A Library for Computing the Quartet and Triplet Distances Between Binary or General Trees. *Bioinformatics*, 30(14):2079, 2014. 9, 32, 54, 56, 64

[69] P. Sanders. Efficient algorithms. chapter Algorithm Engineering — An Attempt at a Definition, pages 321–340. Springer-Verlag, 2009. 4, 5

[70] Y. Shiloach and Y. Perl. Finding Two Disjoint Paths Between Two Pairs of Vertices in a Graph. *Journal of the ACM*, 25(1):1–9, 1978. 66, 67

[71] J. Stapley, P. G. D. Feulner, S. E. Johnston, A. W. Santure, and C. M. Smadja. Recombination: The Good, the Bad and the Variable. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 372 (1736), 2017. 14

[72] T. Griebel and M. Brinkmeyer and S. Böcker. EPoS: a Modular Software Framework for Phylogenetic Analysis. *Bioinformatics*, 24(20):2399–2400, 2008. 64

[73] K.-C. Tai. The Tree-to-Tree Correction Problem. *Journal of the ACM*, 26(3):422–433, 1979. 62

[74] L. Trevisan. Parallel Approximation Algorithms by Positive Linear Programming. *Algorithmica*, 21(1):72–88, 1998. 26, 98

[75] J. von Neumann. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993. 3

[76] D. P. Williamson and D. B. Shmoys. *The Design of Approximation Algorithms*, pages 108–109. Cambridge University Press, 1st edition, 2011. 99

[77] B. Y. Wu. Constructing the Maximum Consensus Tree from Rooted Triples. *Journal of Combinatorial Optimization*, 8(1):29–39, 2004. 25, 93

[78] U. Zwick. Approximation Algorithms for Constraint Satisfaction Problems Involving at Most Three Variables Per Constraint. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '98, pages 201–210. Society for Industrial and Applied Mathematics, 1998. 26, 98