EFFICIENT ALGORITHMS FOR COMPARING, STORING, AND SHARING LARGE COLLECTIONS OF EVOLUTIONARY TREES

A Dissertation

by

SUZANNE JUDE MATTHEWS

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2012

Major Subject: Computer Science

EFFICIENT ALGORITHMS FOR COMPARING, STORING, AND SHARING LARGE COLLECTIONS OF EVOLUTIONARY TREES

A Dissertation

by

SUZANNE JUDE MATTHEWS

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Chair of Committee,	Tiffani L. Williams
Committee Members,	Nancy M. Amato
	Jennifer L. Welch
	James B. Woolley
Head of Department,	Hank W. Walker

May 2012

Major Subject: Computer Science

ABSTRACT

Efficient Algorithms for Comparing, Storing, and Sharing Large Collections of Evolutionary Trees. (May 2012)

Suzanne Jude Matthews, B.S.; M.S., Rensselaer Polytechnic Institute

Chair of Advisory Committee: Dr. Tiffani L. Williams

Evolutionary relationships between a group of organisms are commonly summarized in a phylogenetic (or evolutionary) tree. The goal of phylogenetic inference is to infer the best tree structure that represents the relationships between a group of organisms, given a set of observations (e.g. molecular sequences). However, popular heuristics for inferring phylogenies output tens to hundreds of thousands of equally weighted candidate trees. Biologists summarize these trees into a single structure called the consensus tree. The central assumption is that the information discarded has less value than the information retained. But, what if this assumption is not true?

In this dissertation, we demonstrate the value of retaining and studying tree collections. We also conduct an extensive literature search that highlights the rapid growth of trees produced by phylogenetic analysis. Thus, high performance algorithms are needed to accommodate this increasing production of data. We created several efficient algorithms that allow biologists to easily compare, store and share tree collections over tens to hundreds of thousands of phylogenetic trees. Universal hashing is central to all these approaches, allowing us to quickly identify the shared evolutionary relationships contained in tree collections. Our algorithms MrsRF and Phlash are the fastest in the field for comparing large collections of trees. Our algorithm TreeZip is the most efficient way to store large tree collections. Lastly, we developed Noria, a novel version control system that allows biologists to seamlessly

manage and share their phylogenetic analyses.

Our work has far-reaching implications for both the biological and computer science communities. We tested our algorithms on four large biological datasets, each consisting of $20,000$ to $150,000$ trees over 150 to 525 taxa. Our experimental results on these datasets indicate the long-term applicability of our algorithms to modern phylogenetic analysis, and underscore their ability to help scientists easily exchange and analyze their large tree collections. In addition to contributing to the reproducibility of phylogenetic analysis, our work enables the creation of test beds for improving phylogenetic heuristics and applications. Lastly, our data structures and algorithms can be applied to managing other tree-like data (e.g XML).

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Dr. Tiffani L. Williams, for her guidance and support over the last six years. I first met Dr. Williams through the CRA-W DMP program, when she first became my mentor. Back then, she saw a potential in me that I didn't even know that I had. She has inspired me to always question my limits and to constantly strive for excellence. Her mentoring, compassion, and sacrifice has molded me into the researcher I am today. Though my time at Texas A&M is ending, her lessons and aphorisms will accompany me through all of life's journeys.

I would also like to thank my committee members, Dr. Nancy Amato, Dr. Jennifer Welch and Dr. James Woolley, for the countless conversations, and their support and faith in me. I have had so much fun interacting and learning from them over the years. Dr. Amato and Dr. Welch have been a constant source of inspiration for me, and I am truly grateful for all their advice and time. I credit Dr. Woolley for cementing my love and appreciation for Phylogeny, and I have several fond memories of the class I took with him at Texas A&M.

Next, I would like to thank the National Science Foundation for funding my research for the first three years of my PhD. I am also indebted to the Texas A&M Dissertation Fellowship program for funding me during the last year of my PhD. I am also grateful for the travel fellowships I have received over the years, including those from the Grace Hopper Conference, the Richard Tapia Conference, the SuperComputing Broader Engagement program, International Symposium on Bioinformatics Research and Applications (ISBRA), the Applied Mathematics Program workshop on Reproducible Research (AMP), and the NSF-ADVANCE program. The opportu-

nities I received to network and share my research with a broader audience are truly irreplaceable.

My work would also not be possible without the many wonderful collaborators I've had: Dr. Seung-Jin Sul, Dr. Marc Smith, Ana Dal Molin, Grant Brammer and Ralph Crosby. I am thankful for the discussions and friendships that have resulted from our partnerships over the years. Their separate insights allowed me to think about my work in novel ways, and have only added to the strength of this dissertation. Dr. Seung-Jin Sul was also my graduate mentor while I was in the DMP program. I am especially grateful to him for the patience, guidance and kindness he showed me while I was a new PhD student.

So many friends and student mentors have been a constant source of support for me over the years. I am especially grateful to Dr. Lydia Tapia and Charles Lively for their encouragement and support. They eased my transition to Texas, and were among my first friends here. I am also grateful to all the women I met and interacted with through the Aggie Women in Computer Science: you are a constant source of joy. AWICS kept me sane these last four years, and it was an honor to be able to serve both as an officer and as Graduate President.

Lastly, I would like to thank my family for their love, prayers and support over these last many years. Even though they didn't always understand what I was doing, my parents were always thrilled with any accomplishments I had to share and were a source of comfort when things got tough. Most of all, I am thankful to my fiancé Kevin. He has been a source of constant strength and unconditional love over the last five and half years. He stood by me as made the difficult decision to move to Texas, and helped me never regret my decision. His unwavering confidence in me and my abilities have kept me going even when I thought the challenges were too much to bear. To him, I lovingly dedicate this work.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

FIGURE Page

FIGURE Page

FIGURE Page

CHAPTER I

INTRODUCTION

The theory of evolution, which forms the foundation of modern biology, states that all organisms evolved from a common ancestor. Consequently, a fundamental question to biology and our understanding of life is *how are organisms related to each other?* The most common approach for depicting evolutionary relationships between a group of organisms is a phylogenetic tree.

Figure 1 shows an example phylogenetic tree between the six members of the pantherine lineage of cats, which was adapted from Davis *et. al.* [1]. The direction of evolution flows from the ancestral root to the leaves of the tree. In Figure 1, Clouded Leopard is used as the outgroup to root the tree. The big cats (Lion, Tiger, Jaguar, Leopard, Snow Leopard and Clouded Leopard) are the taxa and are located at the terminal nodes (or leaves) of the tree. Internal nodes ($a \ldots d$) represent common ancestors between the organisms. The tree in Figure 1 shows that Lion and Leopard are sister taxa, as they share a common ancestor labeled by node a.

Understanding the evolution of pantherine cats is critical to the creation of successful conservation programs for these rare and beautiful creatures, whose numbers have dwindled over the years due to poaching and habitat destruction [1]. In addition to improving our understanding of the big cats, phylogenies have many other applications, such as convicting and exonerating people suspected of transmitting the HIV virus [2, 3, 4, 5], drug discovery [6, 7], and tracing the origins of medieval texts [8, 9].

This dissertation follows the style of *IEEE Transactions on Computational Biology and Bioinformatics*.

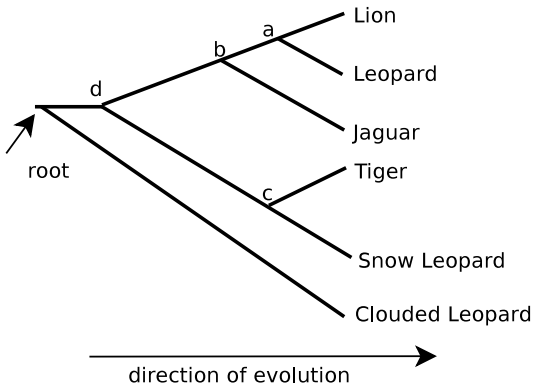


Fig. 1. An example phylogenetic tree showing the relationship between the pantherine lineage of cats. Extinct ancestral taxa are shown at internal nodes, and are labeled $a \ldots d$. Figure adapted from (Davis *et. al*) [1].

A. Research Objective

Our goal is to help biologists build better phylogenetic trees. However, inferring the phylogenetic relationships between a group of organisms is difficult. Since the process of evolution is not actively observed, the true paths of evolution cannot be known. Fully resolved and uncontroversial phylogenies are rare. Multiple groups have attempted to reconstruct the phylogeny of the pantherine cats. However, there is great disparity between these phylogenetic studies [1]. The latest hypothesis for the evolution of the pantherine cats is shown in Figure 1.

Furthermore, for a set of n taxa, there are $(2n-5)!!$ possible (or candidate) trees that exist, which cannot be studied exhaustively. This space of possible solutions is known as tree space. As a result, scientists use heuristics based on NP-hard optimization criteria (such as maximum parsimony and maximum likelihood) to limit the number of hypothetical trees to evaluate. However, instead of returning a single best hypothesis, phylogenetic heuristics often output tens to hundreds of thousands of candidate solutions. A common trend in phylogenetics is encapsulating the result

into a single consensus tree, which is published in a scientific journal. Unfortunately, the large number of trees used to generate a consensus tree are often discarded and rarely shared, making it difficult to reproduce results. Furthermore, the assumption for building a consensus tree is that the information discarded is less important than the information retained. But, what if this assumption is not true?

Given the difficulties involved with leveraging large collections of trees, *how can we design new approaches that enable biologists better leverage the information contained in their phylogenetic analyses and easily share their results with each other?* In this dissertation, we present several novel algorithms that allow biologists to (a) quickly compare large collections of trees, (b) efficiently store large collections of trees and (c) easily manage and share their phylogenetic analyses with each other. The methods discussed in this work represent the fastest and most efficient approaches in the field, are applicable to a wide variety of phylogenetic tree collections, and represent several novel ideas for improving phylogenetic analysis.

B. Our Contributions

We first demonstrate that evaluating trees based solely on scores can belie underlying topological diversity [10, 11]. Evaluating trees topologically allows us to understand how heuristics travel through tree space, and helps us ascertain what value (if any) a slower heuristic may have. Furthermore, assessing the level of tree diversity in a collection of trees returned by phylogenetic search allows us to determine if the search converged. In two case studies [12] we performed, we discovered that trees produced from the same run cluster more closely with each other than with trees derived from different runs of MrBayes [13], a very popular phylogenetic search heuristic in use today. This suggests that tree topology should be incorporated as a possible criterion

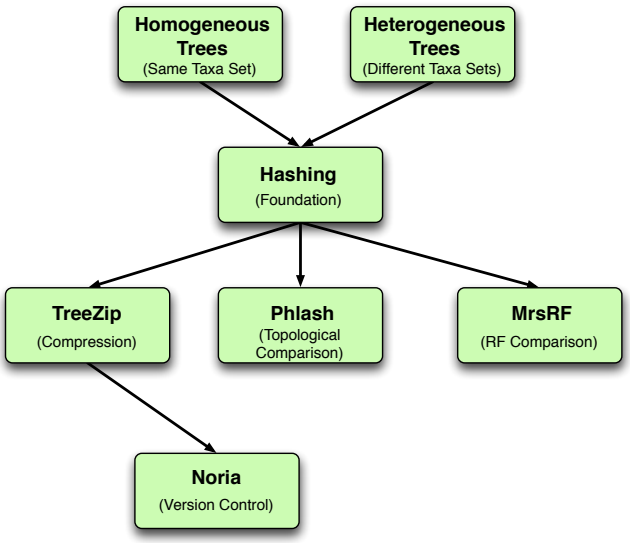


Fig. 2. An overview of our research contributions.

for convergence detection. These examples strongly disprove the notion that tree collections lack value, and underscore the importance of retaining and studying large collections of trees.

To aid scientists in the analysis of large collections of evolutionary trees, we developed four novel algorithms capable of comparing, storing and sharing tree collections composed of tens to hundreds of thousands of trees. We tested our approaches on four large biological datasets, with each collection containing between $20,000$ and $150,000$ trees over 150 to 525 taxa. An extensive literature search that we conducted confirmed the unusual size of these tree collections in the context of modern phylogenetic analysis, and emphasized that these collections represent our best view of the future. Thus, the ability of our algorithms to efficiently handle these large datasets highlights the long-term applicability of our work.

Figure 2 shows an overview of our research contributions. All of our algorithms take as input a collection of evolutionary trees that can either be homogeneous or

heterogeneous. Unlike homogeneous tree collections, every tree in a heterogeneous collection of trees can contain a unique set of taxa. To the best of our knowledge, no methods have been created that exploit the relationships between heterogeneous collections of trees. As biological research becomes increasingly collaborative and scientists begin to merge their results, an ability to identify relationships between heterogeneous collections of trees will become vital to the community.

Tree collections are first fed into a series of universal hashing functions, which form the foundation of our work. We extended a previously established Monte Carlo randomized hashing algorithm for discovering shared topological relationships in a collection of trees to be a Las Vegas approach. We also show how these hash functions can be used to discover relationships in heterogeneous collections of trees. Lastly, we described and implemented two new universal hashing functions for detecting the set of unique trees in a tree collection. Ultimately, these hash functions allow us to quickly discover the smallest set of discrete topological units that define a tree collection, and are critical to the design of our MrsRF, Phlash, TreeZip, and Noria systems.

We first use our universal hashing functions to develop powerful algorithms that quickly compare tens to hundreds of thousands of evolutionary trees. Our algorithm MrsRF [12] is capable of comparing large groups of phylogenetic trees in real-time. It is based on the MapReduce paradigm, popularized by Google and used by large companies such as Amazon, Yahoo! and Facebook. When run sequentially, MrsRF is up to 2.8 times faster than the previous fastest RF matrix algorithm, HashRF [14]. When run in parallel, MrsRF is able to compare $33,306$ phylogenetic trees in approximately 35 seconds. We also present Phlash, a powerful algorithm that is currently the fastest method for comparing large groups of trees. Phlash's novelty lies in its ability to quickly compare trees in numerous ways not yet used in the context of

topological analysis of phylogenetic trees. The real-time comparison of large groups of phylogenetic trees allows us to learn more about the underlying collections and the heuristics that produce them.

Next, we show how to use universal hashing to efficiently compress and store large collections of phylogenetic trees in our TreeZip algorithm [15, 16]. Tree collections are currently stored in the Newick format, in which trees can be represented in an exponential number of ways. The Newick representation of our biological datasets occupy between 67 MB and 533 MB of disk space. TreeZip is capable of representing the information in a collection of trees uniquely and in a space that is 98% (74%) smaller than what can be achieved by an unweighted (weighted) Newick file. When combined with standard compression methods such as `gzip`, `bzip,` and `7zip`, we achieve in excess of 98% space savings. Thus, the TreeZip+`7zip` representation of our large biological tree collections can now be e-mailed. Lastly, the textual representation of the TreeZip (TRZ) compressed format allows for tree operations to be performed on the file directly in real-time, without any loss of space savings.

Our work with MrsRF, Phlash, and TreeZip motivates the design of Noria, a distributed version control system for managing phylogenetic analyses. Noria shares many features of standard distributed version control systems, such as Mercurial [17], Subversion [18] and Git [19]. For example, Noria allows biologists to easily edit their tree collections and share them with each other.

However, software version control systems were not designed to manage tree collections, since they treat trees as unstructured text. Thus, they do not properly detect when a tree has changed, resulting in duplicate information being stored. In contrast, Noria uses the TRZ format to efficiently and uniquely represent collections of trees. Biologists can run phylogenetic analyses in the context of the Noria system, the results of which are automatically and implicitly stored. This simplifies the often

tedious process of managing multiple experimental analyses.

In addition to providing benefits to the biology community, our work makes several contributions of interest to the computer science community. We have augmented an earlier established randomized hashing algorithm to be a Las Vegas approach, added the abilities to detect unique trees, and relate heterogeneous collections of trees. Our new algorithms improve the speed and efficiency of tree operations, as they are now performed directly on a compact and unique representation of tree collections. The real-time performance of these operations allows them to be performed dynamically in the context of other systems, like Noria. The ability to perform real-time tree operations will be useful in the design of other systems that manage tree-like data, such as graphs and XML [20] files.

C. Dissertation Organization

The rest of this document is organized as follows. Chapter II describes the principles behind phylogenetic trees and phylogenetic analyses in detail. In Chapter III, we discuss current methods for analyzing and storing trees, and their limitations. Chapter IV discusses our universal hashing functions and how we use them to store tree data uniquely. In Chapter V we discuss our MrsRF and Phlash algorithms, the fastest algorithms for comparing large collections of trees. Chapter VI describes our algorithm TreeZip, the most efficient approach for storing trees. In Chapter VII we discuss the Noria system. Lastly, we summarize in Chapter VIII.

CHAPTER II

PHYLOGENETIC ANALYSIS

The goal of phylogenetic analysis is to infer the best tree that describes the relationships between a group or organisms of interest, given a set of observations (e.g. molecular sequences). In this chapter, we give a high-level overview of phylogenetic analysis, establish the value of evaluating tree collections, and present a literature search that describes the current state of phylogenetic analysis. The work presented in this chapter motivates the creation of the high performance algorithms and version control system discussed in this dissertation.

A. Motivation

Why do biologists conduct phylogenetic analyses? In Chapter I, we presented the most recent hypothesis for the evolution of the pantherine cats. Understanding the phylogeny of the big cats and other endangered species allows for the creation of successful conservation programs, thus promoting biodiversity. However, the applications of phylogenetic trees extend far beyond assisting conservation efforts. Phylogenies have also been used successfully for selecting proteins for drug interactions [6, 7], ecological studies [21, 22], and to trace the origins of medieval texts, such as the Canterbury Tales [9] and Germanic poetry [8].

Reconstructing phylogenies has been critical for several forensic applications [2, 3, 4, 5] related to HIV transmission, and have been used to convict and exonerate people suspected of purposefully transmitting the virus. In the recent case *State of Texas vs. Philippe Padieu* [2], a phylogenetic tree was used to convict the defendant, Philippe Padieu of "aggravated assault with a deadly weapon" against his six female victims, resulting in five 45-year sentences and a 25-year sentence to be served con-

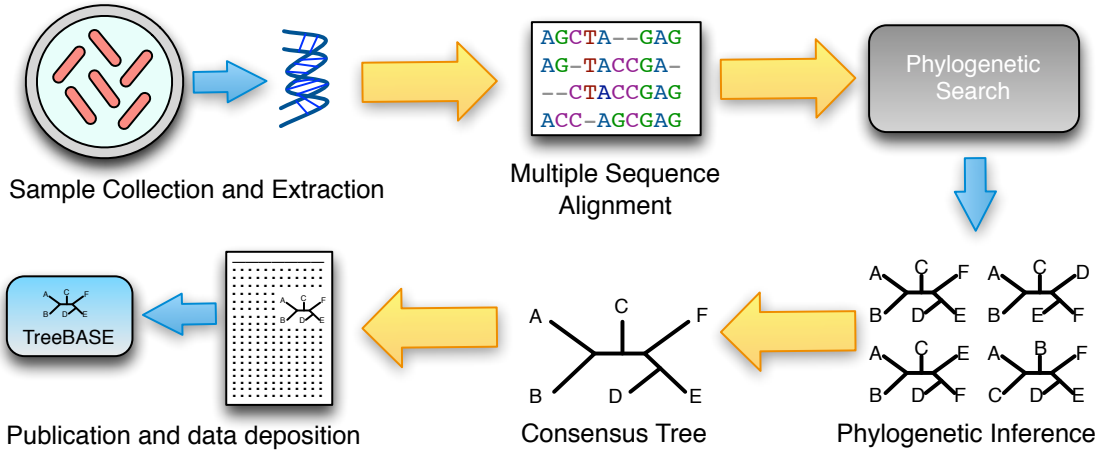


Fig. 3. An overview of the process of phylogenetic analysis.

secutively [2]. Given the high stakes, it is very important for phylogenetic trees to be robust. As phylogenetic analyses become used for increasingly crucial applications, their experimental reproducibility will become imperative.

B. A Brief Overview of Phylogenetic Analysis

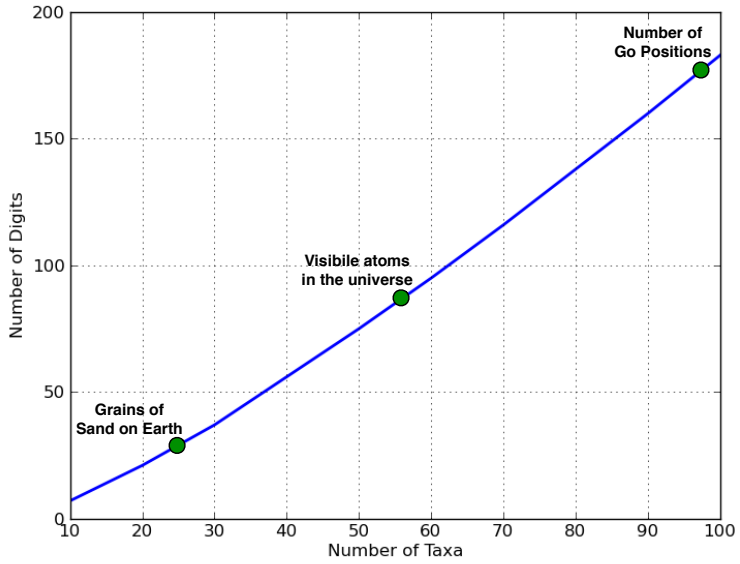
Phylogenetic analysis is a complex process that can take years to complete. Figure 3 gives a pictorial overview of the major steps involved. Blue arrows (small) depict actions within a step, while yellow arrows (large) depict transitions between phases of the analysis. We briefly discuss each of these in turn.

The first step is sample collection and extraction, in which samples from each organism under study are collected. These organisms of interest are commonly referred to as *taxa*. For molecular studies, biological sequences such as DNA, RNA, and amino acid (protein) are extracted and sequenced. For morphological studies, a series of *characters*, or informative traits of interest, are tabulated for all the data.

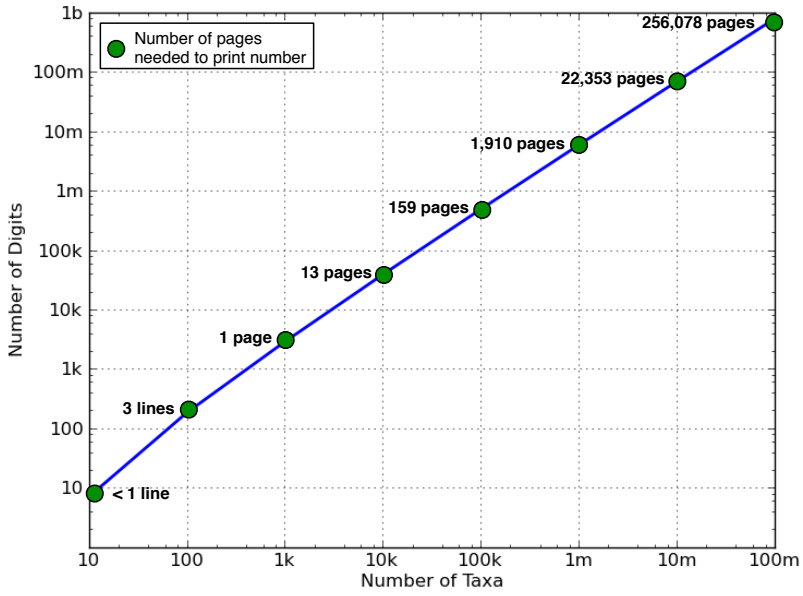
Most journals require authors to deposit their sequences into the NCBI GenBank database [23], which currently contains 150 million sequences. While NCBI performs some level of quality control, the data is not curated [24]. Molecular data can be prone to sequencing error or contain missing data. Given the difficulties involved in sample collection and sequence extraction, this is perhaps the most time-consuming step of phylogenetic analysis, and can take years to complete.

Once the set of observational data is collected, the next step for molecular phylogenetics is multiple sequence alignment. The processing of sequence alignment takes as input a set of biological sequences of different length and constructs a matrix in which each sequence is of the same length, through the insertion of gaps in the sequences. Gap regions reflect the process of insertions or deletions in the evolutionary process. The goal is to maximize the number of putatively homologous positions within a set of sequences, while minimizing the number of gap regions. A multitude of algorithms for performing multiple sequence alignment exist and are freely available, from progressive alignment algorithms such as CLUSTAL-W [25], to simultaneous alignment/tree-estimation algorithms such as POY [26] and SATé [27]. For morphological studies, a character matrix is created, in which a series of numbers denote the presence or absence of characters of interest for each organism under study.

The multiple sequence alignment serves as input for phylogenetic analysis. The goal of phylogenetic analysis is to infer the best tree of interest, given a set of observations (i.e., the input data). Since the "true" (or actual) tree is not known, each tree represents a hypothesis of how a group of organisms are related. Let n denote the number of taxa under study. For small numbers of taxa ($n < 10$), the set of possible trees is easily enumerable and can be evaluated individually. However, as n grows large, the problem quickly becomes intractable: there are $(2n-5)!!$ possible unrooted trees that can reflect the relationships between n taxa of interest. Figure 4 shows the



(a) 10 taxa to 100 taxa



(b) 10 to 100 million taxa

Fig. 4. The growth of tree space (in digits) with respect to the number of taxa.

growth of tree space (in digits) with respect to the number of taxa under study. In Figure 4(a), we show the relationship between the number of taxa (from 10 to 100) and the size of tree space with respect to some commonly quoted large numbers. For example, there are approximately 1.7×10^{182} possible trees to represent the relationships between 100 taxa, requiring 183 digits. This is roughly on the same order of magnitude of the estimated number of Go positions (10^{170}), and is far greater than the number of estimated atoms in the visible universe (10^{80}) and the estimated number of grains of sand on earth (10^{20}).

For $1,000$ taxa, $2,861$ digits are required to represent the number of hypothetical trees, whose digital representation in long form would approximately occupy a page of text. The number of hypothetical trees for $100,000$ taxa is so large that it would take approximately 159 pages to write out completely, approximately the length of this dissertation (Figure 4(b)). If we were to calculate the size of tree space for 100 million organisms (the high end of the number of estimated species on earth), over $256,000$ pages will be needed just to represent the digital representation of this number. That is over 220 times the length of the Bible, and approximately 52 times the length of the Mahabharata, the longest epic in the world.

To navigate this space, current phylogenetic analysis software uses fast heuristic approaches that treat tree inference as an NP-hard optimization problem. Unlike a deterministic approach, heuristics are not guaranteed to return the same solution (i.e. the same set of trees) with every run, but will return a set of "good" solutions (trees). The number of phylogenetics trees returned by these searches typically represent a very tiny fraction of the total space. Methods are usually classified by the optimization criterion used. The most common criteria are maximum parsimony (MP), maximum likelihood (ML) and Bayesian inference (BI).

Maximum parsimony. Maximum parsimony favors the hypothesis that requires the least amount of evolutionary change (known as the principle of parsimony, or *Occam's razor*). It is one of the most trusted methods in comparative biology for correctly establishing homology, especially for fossil and morphological data. Each tree is assigned a parsimony score based on the number of evolutionary changes needed to explain it. The most parsimonious tree is the tree with the lowest parsimony score. The most popular packages for inferring phylogenies via the parsimony criterion include PAUP * [28] and TNT [29].

Maximum likelihood. Maximum likelihood seeks the tree that maximizes the likelihood of the observed data, given a particular tree and model. It is a popular alternative to parsimony methods, as likelihood methods are statistically consistent [30]. Under the maximum likelihood criterion, each tree is assigned a log-likelihood score, and the tree with the lowest log-likelihood score (and hence, the highest likelihood) is considered the most likely hypothesis. Popular packages for maximum likelihood include PAUP *and PhyML [31]. However, likelihood calculations are very computationally expensive. The recent availability of high performance parallel methods that implement maximum likelihood such as RAxML [32] has helped alleviate this computational burden.

Bayesian inference. The process of Bayesian inference (also known as Bayesian analysis) evaluates how likely a tree is, given the observed data and chosen model. While it uses a likelihood function and the models used in maximum likelihood, it operates under a very different philosophy. The goal in Bayesian analysis is to maximize the posterior distribution, which specifies the probability of a particular tree based on the model, the data (usually in the form of a sequence alignment), and

the prior knowledge (specified by the prior distribution).

Like maximum likelihood methods, Bayesian analysis is heavily dependent on the correctness of model chosen, along with any other information from the prior. Using this information, Bayesian inference returns a set of equally credible trees from the posterior distribution. Bayesian analysis traditionally used a Monte-Carlo Markov Chain (MCMC) algorithm, though a variant referred to as Metropolis-coupled Monte-carlo Markov chain (MC^3) is commonly implemented. The most popular program used for Bayesian inference is MrBayes [13].

While the goal is to derive the tree that best describes the relationships between a group of organisms, phylogenetic search heuristics tend to return tens to hundreds of thousands of equally weighted candidate trees, which can further complicate the process. This collection of trees is commonly used to create a consensus tree, in which evolutionary relationships that appear in the majority of the trees (majority consensus) or all of the trees (strict consensus) in the collection appear in the final tree representation. Biologists present this tree in their published paper describing the work, and may deposit the tree in a tree repository such as TreeBASE [33].

However, the collection of trees produced by the phylogenetic analysis is often lost or discarded. One reason for this is the dearth of software systems that make it easy for scientists to maintain and share large tree collections. However, the primary assumption here is that the trees produced by the phylogenetic analysis can be discarded, as they have little to no value. But, is this really the case?

C. Establishing the Value of Phylogenetic Tree Collections

What information can a phylogenetic tree collection tell us? Tree collections represent a "trace" of how a phylogenetic search heuristic traveled through tree space. We can

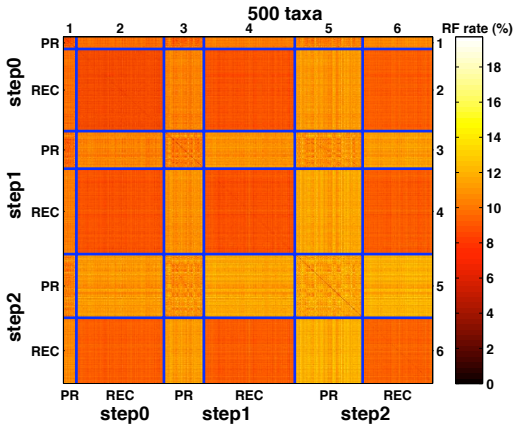


Fig. 5. Heatmap showing the RF distance between the top scoring trees returned from Rec-I-DCM3 and Pauprat.

therefore use this data to make conclusions about the effectiveness of the heuristic that produced them. For example, in [10, 11], we studied the collection of trees returned by two phylogenetic heuristics: Rec-I-DCM3 [34] and Pauprat [35]. Rec-I-DCM3 is faster than Pauprat. Our goal was to answer two questions: (i) what value (if any) do slower heuristics provide? and (ii) how effective are parsimony scores in distinguishing between different tree topologies?

In Figure 5, we show a heatmap representation comparing the topologies between the best scoring trees returned from Rec-I-DCM3 and Pauprat on a biological dataset of 500 seed plants [36]. To illustrate how these algorithms progressed through tree space, we compare the $5,194$ trees returned from the search that have the top three scores ($step_0 \ldots step_2$). The best score parsimony known for this data set is $16,218$. Thus, $step_0$ trees are the trees returned from Rec-I-DCM3 and Pauprat with score $16,218$, $step_1$ trees returned with score $16,219$, and $step_2$ are the trees returned with score $16,220$. The heatmap is generated from an underlying $5,194 \times 5,194$ topological distance matrix, where each cell (i, j) represents the distance between trees i and j.

Each cell is assigned a color denoting how similar a pair of trees are to each other. Hot colors (red to black) denote trees that are very similar. Cold colors (yellow to white) denote areas that are very dissimilar.

Our visualization shows that Rec-I-DCM3 and Pauprat find topologically different trees that have the same parsimony score. Furthermore, as the parsimony score increases, there is more variety in the topological structure of the $step_1$ and $step_2$ trees. The top-scoring trees found by Rec-I-DCM3 algorithm are more similar to each other than their Pauprat counterparts. Thus, the Rec-I-DCM3 $step_i$ trees tend to form clusters that are distinct from the $step_i$ Pauprat trees. These results tell us two things. First, the parsimony score is not as informative as a topological comparison at detecting diversity in a collection of trees. Therefore, score alone should not be used to evaluate a collection of trees. This shows that there is merit in mining tree collections for additional information (such as the level of tree diversity) using techniques like the topological distance matrix. Second, the differences in the trees found by Rec-I-DCM3 and Pauprat suggest that the two heuristics traversed tree space differently, and returned different collections of trees. Therefore, a slower heuristic like Pauprat is valuable since it found different, but equally valuable trees. Both of these facts suggest that there is an incredible amount of wealth contained in a collection of trees, and that it is valuable to consider the merits of a collection as a whole instead of summarizing the results into a single tree.

Convergence detection. In terms of Bayesian inference, looking at the underlying topologies of a collection of trees can tell us how well the analyses that produced them converged. To lessen the chance that a search gets trapped in a locally optimal space, phylogenetic heuristics are often run multiple times. With multiple executions, the desired goal is for all the runs to converge to a mutual space.

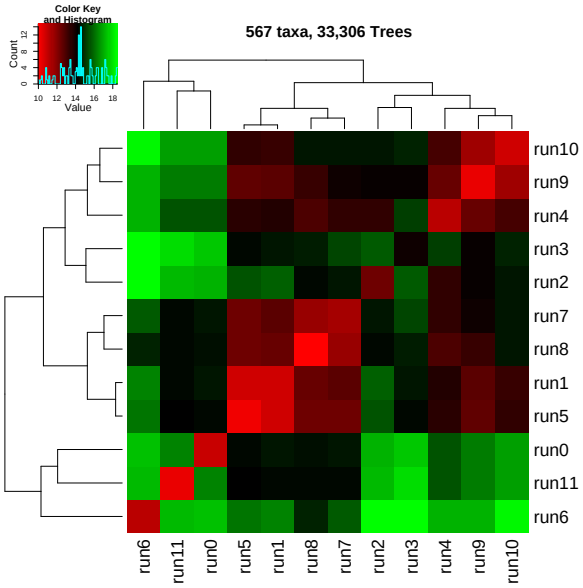


Fig. 6. Heatmap depicting the clustering of 12 runs of Bayesian analysis on a collection of $33,306$ angiosperm trees.

In [12] we performed a case study of how a collection of $33,306$ angiosperm trees [37] produced by 12 runs of Bayesian analysis converged. If the runs truly had converged, then the trees should be very similar to each other. Figure 6 shows a heatmap depicting the $33,306 \times 33,306$ topological distance matrix broken up into a 12×12 matrix, where each cell (i, j) denotes the average distance between all the trees in run i compared to run j. Each run consisted of $2,000$ to $3,000$ trees. Hot regions, colored in shades of red, denote highly similar trees. Cool regions, colored in shades of green, denote dissimilar trees. Hierarchical clustering, using the `hclust` function in R, is used to cluster highly similar cells in the heatmap to each other.

Contrary to our expectations, the heatmap in Figure 6 shows regions of high similarity among the trees within a run, and regions of dissimilarity across runs. The clustering also shows that runs 4, 9, 10 exhibit trees with high levels of similarity among them. One conclusion is that these runs converged to similar areas of tree

space in the phylogenetic search and the trees from those runs can be summarized by a single tree (such as a consensus tree). This is also true for runs 1, 5, 7, and 8. The clustering of the other runs (runs 2 and 3) and (runs 0, 6, 11) have lower levels of similarity. This suggestions that there exist several well-supported partitions of the trees and that each partition should be summarized separately in order to minimize information loss. Moreover, the data suggests that the 12 Bayesian runs did not converge to the same place in tree space. Thus, comparing trees that were outputted from a phylogenetic analysis can also be valuable for detecting convergence in the context of a phylogenetic analysis.

Our two examples strongly disprove the central assumption that the tree information discarded by consensus approaches has no value. Furthermore, there is no system in place that allows biologists to easily curate their tree collections. Sequence data is stored in GenBank. Sequence alignment and phylogenetic search algorithms can be accessed from the software authors and/or are available online. Tree repository systems such as TreeBASE only store the consensus trees produced by an analysis. This essentially results in a massive amount of information loss in the phylogenetic inference pipeline. The amount of information loss is commensurate with the growth of phylogenetic analysis. As analyses produce more trees, a greater number of trees are discarded after producing the corresponding consensus trees. But, how big is this problem? How fast are phylogenetic analyses growing?

D. The Current State of Phylogenetic Analysis

To study the rate of information loss and the growth of phylogenetic analyses, we conducted an extensive literature search over four highly ranked journals that publish phylogenetic analyses and encourage the sharing of trees. We are certainly not

the first to study the growth of phylogenetic analysis and make a call for creating methods for handling the burgeoning nature of phylogenetic data. A review [38] by Sanderson *et. al.* in 1993 looked at the growth of phylogenetic analysis in terms of the number of analyses that occurred per year, the types of organisms under study, and the methods being used. They concluded that phylogenetic data was rapidly accumulating, and called for the creation of a "phylogenetic data base of data and trees" to store the data related to phylogenetic analysis. This led to the construction of TreeBASE in 2000 [33][1]. Twelve years after its inception, TreeBASE boasts over "$2,946$ publications written by $6,106$ different authors. These studies analyzed $5,717$ matrices and resulted in $8,462$ trees with $465,762$ taxon labels that mapped to $82,043$ distinct taxa" [40]. TreeBASE clearly brings a much needed level of data sharing in the phylogenetic community. As a result, we make it a focal point around which we conduct our literature search.

Since Sanderson *et. al.*'s review over fifteen years ago, the landscape of phylogenetic analysis has undoubtedly changed. Bayesian inference and high performance phylogenetic heuristics did not exist then, and these methods play a large part in modern phylogenetic analysis. Furthermore, we wish to survey phylogenetic analyses in a generic sense, without restricting our search to any particular community. We are also mainly interested in studying how the number of taxa under study (n) and the number of trees (t) outputted by phylogenetic analysis have grown over the years, which was not an aspect of data collection performed in the previous review by Sanderson *et. al.* Data was collected from each journal from the last fifteen years (1997–2011). In the following subsections, we describe the methods behind journal selection and sampling, and the results of our literature search and subsequent analysis on the collected data.

[1] a prototype was released in 1994 [39]

1. Journals of Interest and their Population Sizes

Four biological journals were chosen: *Nature*, *Science*, *Systematic Biology*, and *Molecular Phylogenetics and Evolution*. The four journals were chosen from over 40 journals listed on the TreeBASE site. Each journal was evaluated based on their impact factor (IF) and the number of times their authors have deposited trees into TreeBASE (i.e, the corresponding number of TreeBASE entries). The impact factor reflects the average number of citations a particular journal has. Journals with higher impact factor are considered more "important" that those with lower ones. While the impact factor is not perfect, it is widely used as a measure of journal quality. For example, *Nature* and *Science* have impact factors of 36.101 and 31.634 respectively. The average impact factor of the journals listed on the TreeBASE journals page is 3.409, and the median is 2.492. We note that while *Nature* and *Science* do not require nor recommend their authors to submit their trees to TreeBASE, they were chosen due their stringent data sharing policies and for having the highest impact factors in the field.

Of the 40 journals indicated by TreeBASE as either recommending or requiring deposition of trees in their database, *Systematic Biology* had the highest impact factor of 11.159. *Systematic Biology* had an impressive 173 entries in TreeBASE. This is unsurprising, as the journal requires its authors to submit their final trees and data matrices to TreeBASE upon publication. It was not, however, in the top three in terms of entries in TreeBASE. The journals *Systematic Botany* and *Mycologia* had the most number of entries with 473 and 336 entries respectively. However, we excluded these two journals from our literature search due to their high degree of specialization to specific audiences and relatively low impact factors (< 2). *Systematic Botany* is the journal of the American Society of Plant Taxonomists (ASPT), which requires that at least one author of a submitted paper be a member of ASPT. *Mycologia* is

Table I. Population Sizes. For each journal, we show the number of papers identified that were relevant for our literature search per year. At the bottom of the table, we indicate the total number of papers that were relevant to our search from journal. Journal abbreviations are as follows: *Sys. Bio* is Systematic Biology, and *Mol. Phy. & Evol.* is Molecular Phylogenetics and Evolution.

Year	Nature	Science	Sys. Bio.	Mol. Phy. & Evol.	Total
1997	15	14	21	62	112
1998	20	15	10	95	140
1999	19	11	21	121	172
2000	22	12	24	155	213
2001	25	17	29	162	233
2002	17	23	19	161	220
2003	15	20	28	188	251
2004	28	13	24	329	394
2005	32	25	28	230	315
2006	36	32	23	270	361
2007	17	18	17	339	391
2008	29	14	17	378	438
2009	18	38	16	289	361
2010	21	30	19	420	490
2011	30	24	21	225	300
Total	344	306	317	3,424	4,391

the official journal of the Mycological Society of America, whose research focus is on fungus and lichens. The journal *Molecular Phylogenetics and Evolution* has the third highest number of entries in TreeBASE with 246 entries. It has a relatively high impact factor of 4.394 and accepts papers from a broad range of biological disciplines. For this reason, we picked *Molecular Phylogenetics and Evolution* to be the fourth journal that we review.

Table I show an estimate of the population size of relevant papers over the last fifteen years from each of our journals of interest. For each journal and year, we obtained a population of potential papers. Since *Nature* and *Science* publish on a wide number of scientific disciplines, we narrowed our search using the term "phylogeny" from the search feature available from each journal's website. The search term was purposefully chosen to be generic in order to ensure the maximum amount of coverage. In order to remove papers of irrelevance (i.e. those that did not contain a

phylogenetic analysis), we manually surveyed each paper to determine if it contained the information relevant to our literature search (e.g. number of taxa, number of trees, method of interest). For *Systematic Biology* and *Molecular Phylogenetics and Evolution*, numbers were estimated based on the manual search for relevant papers for each journal. Papers that discussed analyses performed by other authors or simply depicted a phylogenetic schematic not based on any type of analysis were not considered. Super-tree methods were also not considered.

From Table I, one can observe a great disparity in the number of papers published by *Nature*, *Science*, and *Systematic Biology* and those published by *Molecular Phylogenetics and Evolution*. This is due to the fact that *Molecular Phylogenetics and Evolution* focuses almost exclusively on phylogenetic analysis, while the other three journals have a broader focus. *Systematic Biology*, while also having a strong focus on phylogenetic analysis, releases only six issues a year (*Molecular Phylogenetics and Evolution* has 12), and also frequently publishes papers on algorithmic methods and phylogenetic theory. This disparity in population sizes also presents a challenge to us for sampling. How do we effectively sample from populations with so disproportionate sample sizes?

2. Sampling Details

In order to assure reasonable sampling coverage, we chose a modified sampling technique resembling stratified random sampling. Each journal can be subdivided by year into 15 strata (parts), N_h, where $h = 1 \ldots 15$. Thus, the total population size (N) for each journal is $N = \sum_{h=1}^{15} N_h$. For each strata (or year), we sample the maximum of either 10 papers or 10% of the population of papers. Thus, for year 2010 of *Molecular Phylogenetics and Evolution*, we sampled 42 papers ($N_{2010} = 420$). Meanwhile, for *Nature*, we sampled 10 papers ($N_{2010} = 21$). TreeBASE is used to guide our sampling.

If the population of papers contained in TreeBASE for a particular journal and year does not satisfy our sample size, we then use simple random sampling to fill the rest of the sample. To ensure that our samples are representative, we pick an additional three papers at random and ensure that the number of taxa and trees are within the mean and standard deviation. Outliers were detected and removed through visual inspection of a scatter plot of the data. In years where the population for a particular journal was equal to ten, we collect data from all the papers available for that year.

From each paper, we tabulated the number of taxa, the number of trees, the type of method used for phylogenetic analysis, and the software used for the analysis. In cases where multiple analyses were available for a paper, we collected as much data about each analysis performed. It is important to note here that despite the fact that several papers performed multiple analyses, there were many times that insufficient data was made available in the text or supplementary work of that paper to determine details of those analyses. In those cases, the analyses were omitted.

The methods of interest that we surveyed were maximum parsimony (MP), maximum likelihood (ML), Bayesian inference (BI) and neighbor-joining (NJ). We picked these four methods for our analysis as they are the most popularly used methods for building and inferring trees. We are keenly interested in the type of software used to conduct those analyses, and to see see how (if any) usage patterns have changed over time.

3. Analysis

All in all, $2,019$ phylogenetic analyses were surveyed from 858 sampled papers over our four journals of interest. This represents approximately 20% of our total population of papers. Analyses performed over 500 taxa or producing more than $500,000$ trees were conservatively removed as outliers. It is important to note, however, that these

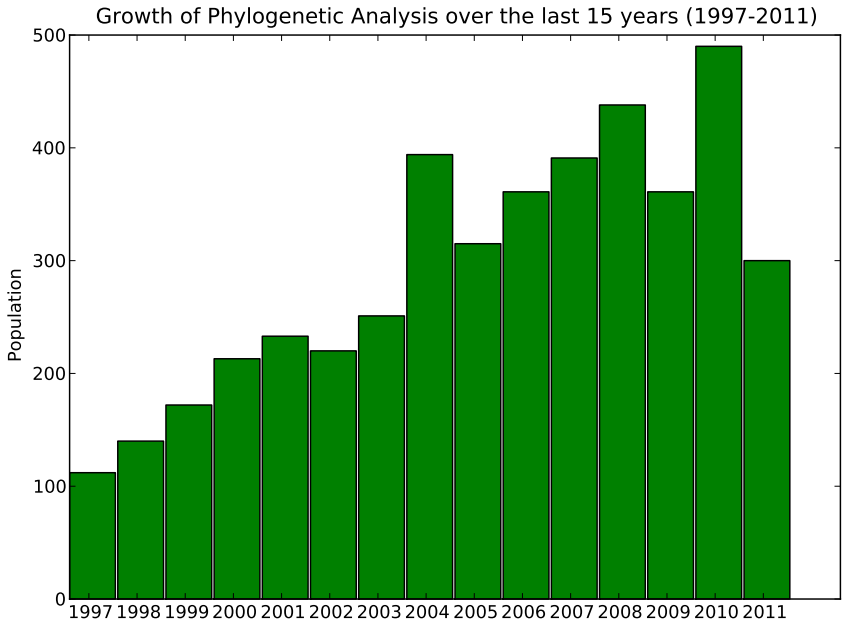


Fig. 7. Increase in the number of papers that perform phylogenetic analysis over the last fifteen years, based on population estimates of our four journals of interest.

outliers represent our best view of the future. Additionally, we removed analyses that we determined to be too "large" for a particular year. 35 analyses met the criteria for being outliers (1.7% of our data) and were removed from consideration, yielding $1,984$ analyses for further consideration. The average number of analyses performed per paper was 2.34.

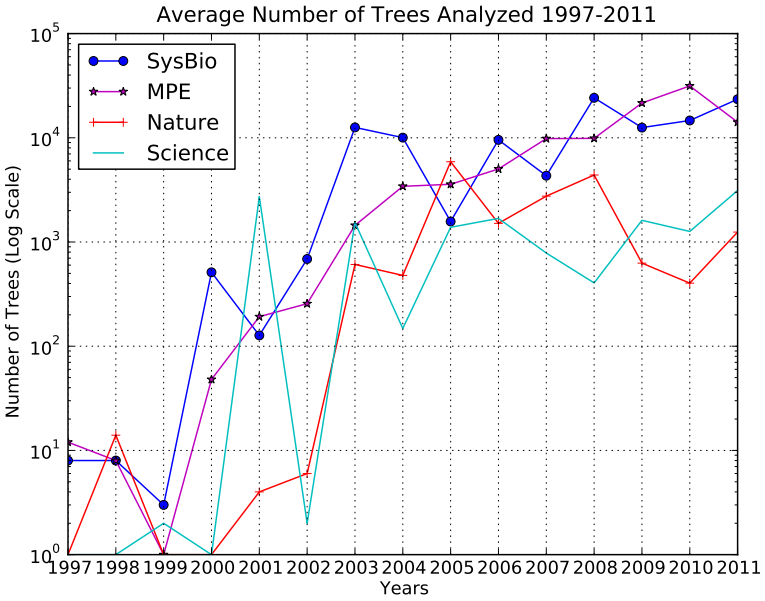
In Figure 7 we show how the number of papers containing relevant phylogenetic analyses has increased over time. The x-axis represents our years of interest, while the y-axis denotes the number of papers found per year. From Figure 7, we can see a steady increase in the number of papers performing phylogenetic analysis over our four papers of interest, at an average growth rate of 1.09. Over the last fifteen years there has been a five-fold increase in the number of papers producing phylogenetic analyses. If this trend were to continue, we would see another five-fold increase

(between $1,000 - 1,500$ papers) on phylogenetic analysis appearing annually over these four journals in the next fifteen years.

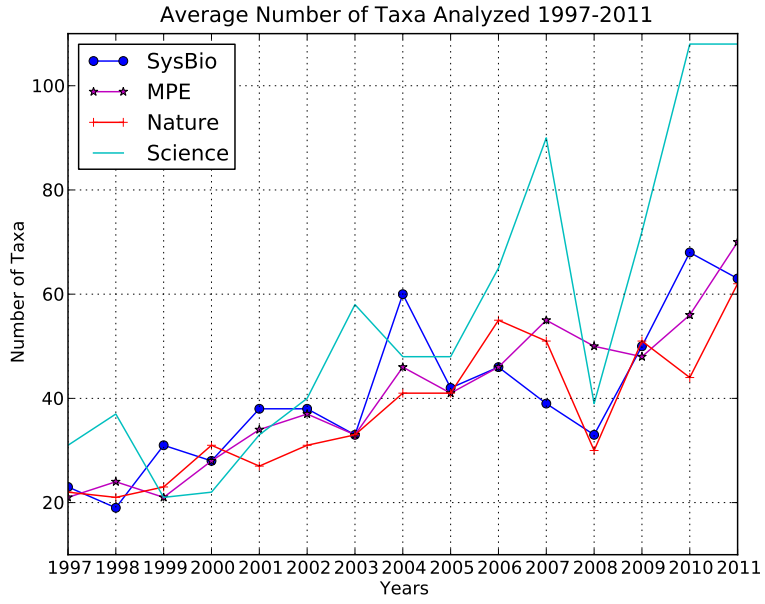
a. Trends in the number of taxa and trees produced over time

In Figure 8 we see how the average number of trees and taxa have changed over time over our different journals of interest. Figure 8(a) shows how the number of trees have changed over time. From this figure, we can clearly see an increase in the average number of trees produced by phylogenetic analyses. From 1997 to 2006, the average number of trees produced by phylogenetic analyses increased from around 100 trees to well over $10,000$ trees. However, from 2006 to today, the increase has been much more gradual. This suggests that the number of trees being produced by analyses are monotonically increasing. Over the last ten years, the number of trees produced by a phylogenetic analysis has increased by a factor of 100. While the rate of growth has slowed since then, it will not be surprising to see analyses in the next fifteen years producing over $100,000$ trees on average.

Figure 8(b) shows how the average number of taxa under study has changed over time over our different journals. Unlike the number of trees, the number of taxa under study between journals remained fairly consistent over the years. There are two spikes in years 2007 and 2010 for the journal *Science*. The data does not seem to suggest that these spikes are due to outliers. We do, however, note that *Science* published a special issue on phylogenetics in 2010. That said, the number of taxa has barely doubled over the last ten to fifteen years. It is safe to say that within the next fifteen years, the average number of taxa analyzed will be still the hundreds. Note that this prediction reflects the average number of taxa. On the high end, the number of taxa will likely be in the thousands.

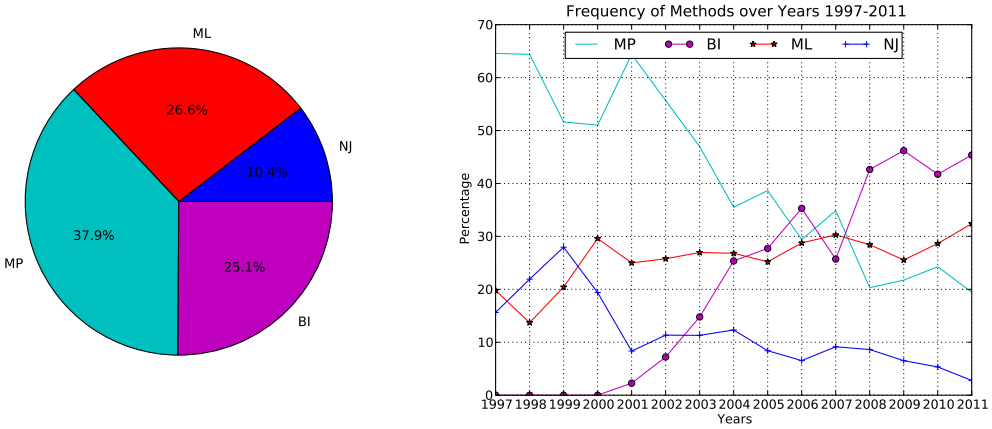


(a) Number of Trees Over Time



(b) Number of Taxa Over Time

Fig. 8. Number of Trees and Taxa varied over time. (a) show how the number of trees have varied over time. (b) shows how the number of taxa has varied over time.



(a) Overall Method Distribution (b) Method Distribution by Year

Fig. 9. Frequency of different methods across all collected data. Plot (a) shows an overall distribution of the data. Plot (b) shows how the distribution of analyses varies according to year.

b. Trends in the type of analysis method used over time

In Figure 9 we illustrate some trends on the types of analyses formed. Figure 9(a) shows the distribution of the types of analyses performed over the last fifteen years. Approximately 26.6% of the observed analyses employed maximum likelihood, and an additional 10.4% of analyses used neighbor-joining. Bayesian analyses made up another 25.1% of the analyses over the last fifteen years. Parsimony analyses made up the last 37.9% of analyses.

However, this plot does not take into account how the percentage of analyses has varied over time. In Figure 9(b), we show the distribution of methods as it varies over year. In the early years, maximum parsimony methods dominated the field, hovering around 60% fifteen years ago. Over time, we've seen a decline in the percentage of maximum parsimony methods performed, while we've seen a rise in popularity of other methods, especially Bayesian analyses. From 2001 to 2009, we've seen a

rapid rise in the number of Bayesian analyses being performed. However, in recent years, the number of Bayesian analyses has declined slightly. This is likely due to the criticism that Bayesian analyses tends to over-estimate branch lengths [41, 42, 43]. The decline of Bayesian methods has been commensurate with the rise of maximum likelihood in recent years. This is largely due to the availability of high performance maximum likelihood software packages, such as RAxML, which is better suited with handling the computational complexity of likelihood calculations. Neighbor-joining has gradually declined in use. In recent years, we notice it is used in studies in which the phylogenetic analysis is supplementary to another main result (e.g. sequencing the genome of an organism).

c. Trends in software used for phylogenetic analysis

In terms of software, the data revealed that PAUP * (85% usage) and MrBayes (88% usage) are the dominant software used for Maximum Parsimony and Bayesian Inference analysis respectively. For Parsimony analysis, the Goloboff suite of programs (TNT [29], NONA [44], and SPA [45]) come in second, but only consist of approximately 7.4% of all the analyses (but 33% of analyses performed in 2011). A similar story can be told for BEAST [46], which is the second-most popular package in use for Bayesian analysis. BEAST is only used in approximately 5.6% of the analyses (26.5% of analyses performed in 2011). For maximum likelihood analysis, RAxML has become a dominant player in recent years, used in 60% of the observed maximum likelihood analyses in 2011. PhyML is the next most popular method for likelihood analysis. Given the infrequency in which neighbor joining analysis is used today, we were unable to determine which package (between PAUP * and MEGA) is dominant today.

The data that we collected clearly suggests that MrBayes and Bayesian inference

is a very popular method for phylogenetic analysis in use today, composing almost half of the analyses performed in 2011. Our data also suggests that Bayesian inference produces the most trees. This is largely due to the fact that the number of trees produced by Bayesian analysis can easily be increased by increasing the sampling frequency along the posterior distribution. Given our results, we make comparing and storing tree collections produced by Bayesian analysis (and MrBayes) a priority in the algorithmic design of the methods presented in upcoming chapters.

E. Our Biological Datasets

In order to handle the burgeoning growth of phylogenetic tree collections, efficient algorithms are needed that can efficiently analyze and store large collections of trees. Given the results of our analysis, datasets commonly produced today have tens of thousands of trees over less than a hundred taxa. To ensure that our algorithms are capable of handling the current sizes of collections produced today and within the next several years, we obtained four large biological datasets:

`freshwater` $20,000$ trees ($20,000$ unique) obtained from a Bayesian analysis [47] of an alignment of 150 taxa (23 desert taxa and 127 others from freshwater, marine, and oil habitats) over two independent runs of MrBayes.

`angiosperms` $33,306$ trees ($33,169$ unique) obtained from a Bayesian analysis [37] of 567 taxa (560 angiosperms, seven outgroups) dataset over twelve runs of MrBayes.

`fish` $90,002$ trees ($47,649$ unique) obtained from an unpublished Bayesian analysis of 264 fish taxa over 2 runs of MrBayes.

`insects` $150,000$ trees ($32,300$ unique) obtained from a parsimony analysis [48] of

525 insect taxa over 5 runs of TNT.

By today's standards, these datasets would be considered atypical. However, they are very representative of what will be typical in the future. In the next chapters, we will show how our powerful algorithms are able to handle these large datasets.

F. Summary

In this chapter, we presented the reader with a brief overview of phylogenetic analysis, and demonstrated that tree collections are getting lost in the pipeline. This is due to the current assumption that tree collections are not useful, and that the consensus tree contains all the information needed to reflect the results of a phylogenetic analysis. We have shown this to be demonstratively false. In two applications, we showed how understanding the level of topological diversity in a collection of trees can tell us a lot about the algorithms that produced them. This is useful for elucidating differences among trees with the same score, establishing the value of slower heuristics and detecting convergence.

Given that this assumption does not hold and that tree collections are valuable, we then studied the rate of information loss through a study of how phylogenetic analysis has evolved over the last fifteen years. We found that the pace of phylogenetic analysis has rapidly quickened. The number of trees produced on average by analyses in 2011 is greater than the number of trees in the entirety of TreeBASE. We also showed that the large number of trees produced in recent years is largely due to the increased popularity of Bayesian analysis, and that the number of trees produced by phylogenetic analysis is on the rise and is expected to grow. Thus, new methods are needed that can handle the burgeoning growth of phylogenetic data, especially as it relates to tree collections.

We also presented the biological datasets that will be used to demonstrate the performance of many of the algorithms demonstrated in this dissertation. In the context of the results of our literature survey, these datasets are unusually large. The ability of any approach to efficiently handle such large datasets will speak for the long-term usability of such approaches. In the following chapters, we will show how our algorithms for analyzing, compressing and storing trees are capable of handling these large datasets.

CHAPTER III

CURRENT METHODS AND RELATED WORK

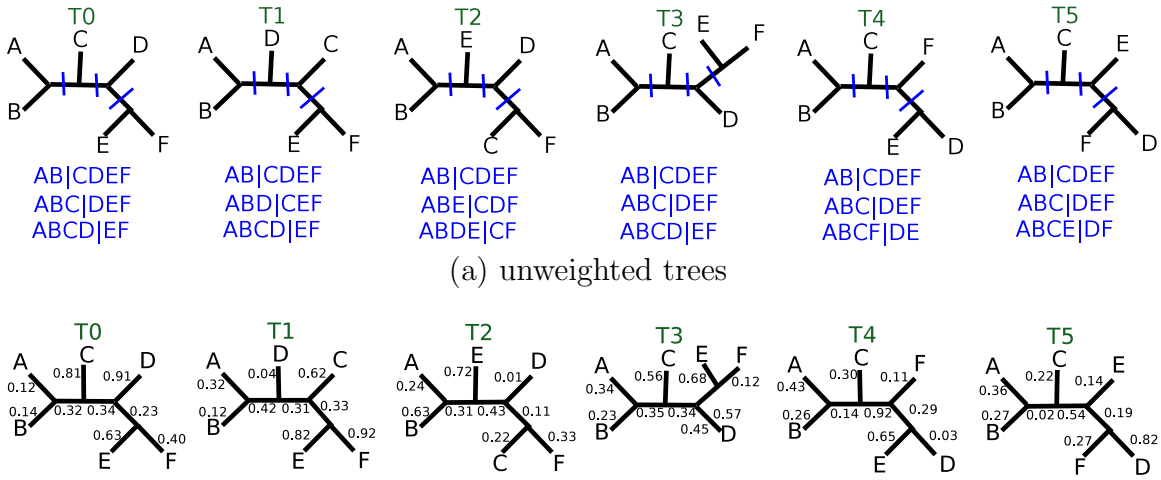
Now that we have established the value of tree collections, what work currently exists for comparing, storing, and sharing trees? In this chapter we discuss the most relevant algorithms and methods that have motivated this dissertation, and the limitations of each. In Section A we discuss current methods for comparing trees. In Section B, we discuss current methods for storing and representing trees on disk. Lastly, in Section C we discuss current methods for sharing trees. In the chapters that follow, we will demonstrate how our high performance algorithms have overcome many of the limitations presented here.

A. Comparing Phylogenetic Trees

Comparing evolutionary trees is an important component of analyzing and visualizing the results of a phylogenetic analyses [49, 50]. In Chapter II, we demonstrated two such applications. But, how are trees compared? In this section, we discuss the basics of tree topology and the most common metrics for comparing trees and collections of trees. We also discuss some challenges as it relates to assessing topological diversity among a set of trees, especially as tree collections get larger.

1. Bipartitions: Topological Units of a Tree

A phylogenetic tree can be decomposed into a set of bipartitions. A bipartition is a cut on an edge in the tree that partitions the set of taxa into two (hence, "bipartition"). Edges adjacent to taxa are considered *trivial* bipartitions, as they do not make a statement of evolutionary interest. Every tree necessarily has n trivial bipartitions. In Figure 10(a), bipartition $AB|CDEF$ splits the taxa in T_0 into two unordered sets,



(a) unweighted trees

(b) weighted trees

Fig. 10. A collection of six phylogenetic trees depicting the relationship between six taxa labeled from $A \ldots F$. Both (a) and (b) are identical. The difference is that (a) is an unweighted representation of the trees while (b) is a weighted representation.

{A,B} and {C,D,E,F} respectively. Every unrooted binary (or bifurcating) tree has exactly $n-3$ non-trivial bipartitions. Rooted binary representations have $n-2$ non-trivial bipartitions. Unrooted (rooted) trees with less than $n-3$ ($n-2$) bipartitions are referred to as multifurcating trees.

It is very common for trees to have weighted edges, where these weights represent evolutionary distances known as *branch lengths*. Figure 10(b) shows a weighted representation of six trees. With weighted trees, trivial bipartitions have weights. Thus, for weighted trees, $2n-3$ bipartitions are considered for unrooted trees, and $2(n-1)$ bipartitions are considered for rooted trees.

Quartets are another common building block for trees, which are based on subsets of four taxa [51, 52]. For a tree over n taxa, there are $3\times\binom{n}{4}$, or $O\binom{n}{4}$ possible quartets. While there are a greater number of possible bipartitions to represent the relationships between n taxa ($2^{n-1}-n-1$), each tree has $O(n)$ bipartitions. In contrast, every tree necessarily has $\binom{n}{4}$ quartets. Furthermore, since trees returned from phylogenetic

search are the "best" trees found by a particular heuristic, there theoretically should be a high degree of *bipartition sharing* amongst the trees in the collection, indicating that the search had converged. For example, in Figure 10, $AB|CDEF$ is a bipartition that is found in all six trees. One can also observe that the bipartition $ABC|DEF$ was found in four of the six trees. Thus, $AB|CDEF$ is represented six times in a collection of trees, and bipartition $ABC|DEF$ is represented four times. As a result, there are 8 unique bipartitions out of the 18 total that exist in this collection of trees.

2. Distance Methods

Comparing phylogenetic trees can generally be divided into two classes. One class of tree measures such as NNI and SPR distances compute the number of steps need to transform one phylogenetic tree into another [53, 54]. The other class of measures are based on decomposing the hierarchical structure of trees into simpler structures (such as bipartitions or quartets) and comparing the differences between them. While methods based on tree transformations (e.g. NNI and SPR) often lead to computationally intractable or NP-hard problems, distances based on comparing tree sets are quite tractable and are the most popular methods for computing tree distances in practice. Bipartition methods are more popular than quartet methods, since they are less computationally-intensive to calculate. In the scope of this dissertation, we focus on bipartition-based methods. However, it is important to note that trees can be compared using either bipartitions or quartets.

In 1978, Bourque [55] developed the partition metric (or symmetric difference) for comparing two trees. It is the number of bipartitions that are not shared with the other tree. Steel and Hendy [50] describe the partition metric as $i(T_1) + i(T_2) - 2v_s(T_1, T_2)$, where $i(T)$ denotes the number of internal edges (or bipartitions) and $v_s(T_1, T_2)$ indicates the number of shared bipartitions among the two trees.

The partition method is closely related to the Robinson-Foulds [56] distance metric, developed in 1981 by DR Robinson and LR Foulds. The RF distance is currently the most popular way to compare trees. It computes the topological distance between two trees by comparing their set of bipartitions. Let $\mathcal{B}_{\mathcal{T}}$ define the set of bipartitions found in tree T. The RF distance between two trees T_i and T_j is:

$$RF(T_i, T_j) = \frac{|\mathcal{B}_{T_i} - \mathcal{B}_{T_j}| + |\mathcal{B}_{T_j} - \mathcal{B}_{T_i}|}{2}. \tag{3.1}$$

The RF distance is obtained by dividing the partition metric by two. In 1985, WHE Days proposed a linear time algorithm [57] (Day's algorithm) for comparing trees. Several packages that compute the RF distance between trees, including PHYLIP [58], CMB [59], RAxML [32], PhyloNet [60] and HashRF [14, 61], implement derivations of Day's algorithm. The RF Rate is a normalized form of the RF distance, achieved by dividing the RF distance by the maximum RF distance, which is $n - 3$ for unrooted binary trees over n taxa.

For weighted phylogenies, there is a corresponding weighted RF distance [62]. For bipartition B in tree T, we denote its weight as $w(B)$. For two trees T_i and T_j the weighted RF distance (WRF) can be defined as:

$$WRF(T_i, T_j) = \sum_{B \in \sum T_i \cup \sum T_j} |w_i(B) - w_j(B)| \tag{3.2}$$

Here, $B \in \sum T_i \cup \sum T_j$ represents the total set of bipartitions over our two trees. Thus if tree T does *not* have a certain bipartition B, then $w(B) = 0$.

Another method for computing weighted phylogenies is called the Branch Score distance [63], proposed by Kuhner and Felsenstein in 1994 and implemented in the `treedist` [64] program in PHYLIP [58]. The Branch Score distance can be defined as:

$$BSD(T_i, T_j) = \sum_{B \in \sum T_i \cup \sum T_j} \sqrt{(w_i(B) - w_j(B))^2} \tag{3.3}$$

It is important to note that while PHYLIP implements the Branch Score distance as the sum of the *square root* of the squares of the differences in weights (as shown above), the original Branch Score distance proposed by Kuhner and Felsenstein is simply the sum of the squares of the differences [64].

3. Topological Matrices for Comparing Trees

To compare t trees, any number of the above metrics can be applied to form a $t \times t$ distance matrix, where each cell (i, j) in our matrix D represents the topological distance between trees T_i and T_j. There are several applications for using RF matrices such as visualizing collections of trees [10, 49] and clustering tree collections [65].

The inspiration for much of our work is HashRF [14, 61], a fast, sequential algorithm for computing an all-to-all RF matrix to compare t unrooted trees on n taxa. Previous work [14, 61] has shown HashRF to be faster than algorithms implemented in other packages. Thus, it is our focus. For a bipartition B, HashRF uses a global hash table H to store that bipartition along with the identities of the trees (TIDs) that contain that bipartition. HashRF uses two uniform hash functions h_1 and h_2, where the h_1 value represents the hash table location for storing the bipartition B and h_2 provides a shortened bipartition identity (BID) for this bipartition. Each unique bipartition is represented uniquely in the hash table as a $(key, value)$ pair, where the *key* is the bipartition of interest, and the *value* is a list of tree ids (TIDS) of the trees containing that bipartition. We discuss these hashing functions and this hashing procedure in detail in Chapter IV.

To compute the RF matrix D, every location representing a unique bipartition is

visited and its list of tree identities (TIDs) are extracted. For each pair of trees T_i and T_j in the list of TIDs, entry $D[T_i, T_j]$ is incremented by one to compute a similarity matrix. Once the hash table has been traversed, each entry $D[i, j]$ is subtracted from $n-3$, the maximum RF distance, to produce the RF matrix. The worst-case running time of HashRF is $O(nt^2)$.

4. Limitations

As phylogenetic analyses continue to increase in size, they produce more trees. As the number of outputted trees t grows very large, it becomes exceedingly difficult to compute the $t \times t$ matrix. One major bottleneck is memory. As t grows large, the amount of memory needed to allocate the $t \times t$ matrix grows quadratically. For example, over 3 GB of memory is needed to compare $33,306$ trees of our `angiosperms` dataset. HashRF and other previous approaches cannot compare our `fish` and `insects` datasets composed of $90,002$ and $150,000$ trees each, as their matrices require 27 GB and 60 GB of memory respectively. Thus, more memory efficient approaches are needed to compare datasets that are relevant to modern phylogenetic analyses.

While RF distance is the most popular way to compare trees, a common criticism is that it is easy to construct an example where the presence of a single rogue taxon yields the maximum distance between trees. As an example, consider the two trees in Figure 11. If we remove taxon E, the trees clearly have the same underlying topology. However, given the placement of E, the two trees have completely different bipartitions, thus yielding the maximum RF distance of 3 for this example. This suggests that the biological community should look beyond current measures such as RF distance to compare trees: *what other metrics can be used to compare trees?*.

Furthermore, comparing weighted phylogenies is very difficult, due to the com-

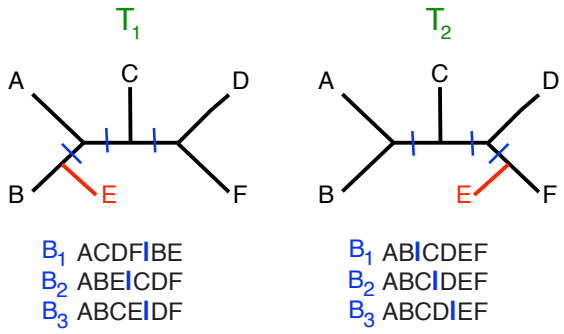


Fig. 11. How a single "rogue" taxon can produce the maximum RF distance between two trees.

putational complexity of the operation. While several packages claim to compute the weighted RF distance, we find that none of them do so correctly. WHashRF, the weighted version of HashRF, does not consider the full scope of weighted bipartitions needed to properly calculate the weighted RF distance between trees. RAxML and PhyloNet both claim to calculate weighted RF, but actually implement RF rate, which does not require branch lengths for its computation. MESQUITE [66] also claims to calculate weighted RF, but we were unable to verify this. While the `treedist` program in PHYLIP does not implement weighted RF distance, it does implement the Branch Score distance. However, our testing has shown that `treedist` is limited to comparing approximately $1,000$ trees, and that a weighted computation of the corresponding $1,000 \times 1,000$ matrix over 150 taxa takes approximately 38 minutes.

Given the computational complexity of comparing weighted phylogenies, it is not surprising that no existing package can compare large groups of weighted trees. However, popular phylogenetic search heuristics such as MrBayes do output weighted phylogenetic trees. Therefore, it is imperative that methods be created that can effi-

ciently compare large groups of weighted phylogenies: *how can we efficiently/quickly compare weighted trees?*

Lastly all the methods and examples discussed so far are over homogeneous collections of trees, in which every tree in the collection contain an identical leaf set. In heterogeneous collections of trees, every tree in the collection can have a unique set of leaves or a subset of leaves that are contained in other trees in the collection. To operate on such collections, it is essential that one is able to discover shared relationships between the trees in the collection. To the best of our knowledge, there are no currently available methods to achieve this. *How can we discover shared relationships in heterogeneous collections of trees?*

B. Storing Phylogenetic Trees

Methods are needed that store phylogenetic tree collections efficiently. In this section, we address the most common ways available for representing and storing phylogenetic trees, the Newick and NEXUS formats. We also discuss TASPI, a method for compressing trees on disk. We do not discuss XML formats for trees such as PhyloXML [67] or NeXML [68], as they were not designed to represent large collections of trees.

1. The Newick Format

The Newick format [69], first proposed in 1986, is the most widely used file format to represent phylogenetic trees. In this format, the topology of the evolutionary tree is represented using a notation based on balanced parentheses. A Newick formatted tree uses nested parentheses to represent the evolutionary relationships (or subtrees) within a phylogenetic tree. Matching pairs of parentheses symbolize inter-

nal nodes in the evolutionary tree. Every tree string is terminated by a semi-colon. For example, a sample Newick string representation for the phylogenetic tree shown in Figure 1 is `(((Lion, Leopard), Jaguar),((Tiger, Snow Leopard), Clouded Leopard));`. To represent t trees, it is common to create a file of t Newick formatted strings, with each string on a separate line. This file is referred to as a Newick file or a PHYLIP tree file.

2. The NEXUS Format

Phylogenetic trees are rarely created in a vacuum. They are most commonly the product of phylogenetic analysis, and thus are associated with an input data set. In order to standardize the way phylogenetic analysis programs handle input files, Maddison *et. al* developed the NEXUS format [70] in 1997. The file format consists of a series of modular units, referred to as blocks, which contain a certain type of phylogenetic data. For example, sequence data are commonly stored in either `CHARACTERS` or `DATA` blocks. For phylogenetic trees, there is a corresponding `TREES` block. However, it is important to note that `TREES` blocks contain Newick-formatted trees. Thus, in terms of representing phylogenetic trees, there is no real difference between PHYLIP and NEXUS files.

That said, the NEXUS format does allow one to associate the input of a phylogenetic analysis (e.g. the data) with the output (the sets of trees). Thus, it is widely used as a standard for input to phylogenetic programs and for representing their output. The format is very flexible and extensible, allowing a program to pick and choose which blocks they wish to handle. Several phylogenetic analysis tools, such as PAUP * and MrBayes, have also declared their own NEXUS blocks to contain batch information for running their heuristics. These are known as `PAUP` and `MRBAYES` blocks respectively.

3. TASPI

One way to reduce the amount of space a set of trees utilizes on disk is through the use of compression. The Texas Analysis of Symbolic Phylogenetic Information (TASPI) [71, 72] shows the benefits of having a compressed representation of phylogenetic trees. To the best of our knowledge, it is the only previously described approach for compressing evolutionary trees. It is written in the ACL2 language. At a low-level, TASPI uses hash-consing [73] to achieve decreased storage requirements and improved accessing speeds. One key advantage of TASPI over standard compression methods is that the TASPI compressed form is text, allowing it to be used as input for other phylogenetic programs while retaining its space savings. Compression methods such as `gzip`, `bzip` and `7zip`, on the other hand, would require the input Newick file to be decompressed before it could be processed. For files that need to be frequently analyzed, the amortized space savings will be 0%. However, there is no public implementation of TASPI available. Given that the primary purpose of TASPI was to compute consensus trees, it is also not clear if a decompression routine is provided.

4. Limitations

The largest limitation to Newick-formatted trees and file formats that utilize the Newick standard, is that the Newick string representation for a tree is never unique. For example, for the tree in Figure 1, another valid Newick representation is `((Jaguar, (Leopard, Lion)),((Tiger, Snow Leopard), Clouded Leopard));`. For a particular tree with n taxa, there are $O(2^{n-1})$ different (but equivalent) Newick strings to represent its topology. We will refer to these different, but equivalent representations as *commutations*.

Consequently, general-purpose data compression techniques cannot leverage domain-specific information regarding the Newick file. In other words, consider a Newick-formatted file that contains $100,000$ commutations of a single tree topology. This file would appear random to standard compression methods, which will consequently compress it poorly. While TASPI does look for common subtrees in a collection of Newick strings, the designers note that TASPI can't always differentiate between the multiple Newick string representations a tree can have. Therefore, it is imperative to create an efficient, yet unique representation of large collections trees in order to store them minimally on disk.

C. Sharing Phylogenetic Trees

As phylogenetic methods increase in popularity and the outputs from their analyses continue to grow in size, better methods will be needed to encourage the sharing of the results of phylogenetic analyses. Sharing of phylogenetic data has many uses. In addition to promoting the reproducibility of experimentation and allowing fellow scientists to independently verify the results of a phylogenetic analysis, making tree data available allows scientists to make statements about co-evolution of species, and offers a test-bank of real data for software designers to use [38]. In this section, we consider some of the most common methods for sharing phylogenetic data.

1. Electronic Transfer

If a tree collection is small enough, it can be transferred between scientists via e-mail. However, for large datasets such as the ones we study, this is not often possible, as the files are several megabytes in size, even when compressed. More computationally savvy users can try to share trees by using an FTP server or uploading it to a personal

web-space. However, this requires a certain amount of technical know-how, and most scientists (including computer scientists) do not have the time or patience to set up their own FTP servers. DropBox [74] is a service for sharing large files that emerged in 2007 and has enjoyed a steadily increasing popularity. Users can sign up for an account that allows for 2 GB of space for free. Larger accounts can be obtained for a monthly service fee.

2. Tree Databases

One way to encourage scientists to share their collections of trees is through the use of tree databases. TreeBASE [33, 39, 40] is the most popular repository for phylogenetic trees. Due to the burgeoning nature of phylogenetic data, Sanderson et. al proposed in 1993 the construction of a phylogenetic tree database that will promote sharing of phylogenetic data amongst scientists [38]. A prototype of TreeBASE was released in 1994, and over the last seven years the database has significantly matured and is endorsed by several venues [33]. Several journals, most notably *Systematic Biology* and *Nature*, encourage or require their users to deposit their tree data in TreeBASE. Dryad [75] is another noteworthy database, which also interfaces with TreeBASE. While Dryad's stated purpose is to offer a "home" for data that is not housed elsewhere, we noticed a handful of studies that deposit tree data in Dryad.

3. Limitations

While tree repositories such as TreeBASE and Dryad have increased in popularity, most trees deposited in these repositories are the single consensus trees produced at the end of the analysis. As of December 2011, there are only $8,462$ total trees contained in TreeBase [40]. It is worth noting that all our large datasets of trees contain more trees individually than the whole of TreeBASE.

This massive data loss severely hampers the ability of scientists to independently reproduce and verify the results of a phylogenetic analyses. Furthermore, it limits the access to large test datasets for scientists who care about creating more efficient methods and algorithms. While the literature survey in Chapter II demonstrates the burgeoning nature of phylogenetic analysis, our research group has had considerable difficulty procuring large phylogenetic datasets derived from real data. Therefore for the sake of reproducibility and creating test beds, methods that allow scientists to easily share their tree collections with each other will be of great value to the community.

D. Summary

In this chapter, we discussed current methods for comparing, storing and sharing trees. We focus on bipartition-based methods for comparing trees. The most popular way for comparing a pair trees is the Robinson-Foulds (RF) distance metric. To compare t trees, RF distances can be summarized in a $t \times t$ RF distance matrix. While several methods exist for computing RF distance matrices, this becomes a difficult problem as trees become fairly large. How can we efficiently compare $150,000$ trees, for example? We also discussed an oft-mentioned limitation of the RF distance metric itself, in which a "rogue" placement of a single taxon can cause two similar trees to be maximally dissimilar under RF distance. This suggests that work should be done on exploring other measures for comparing trees.

While several formats have been proposed for storing trees (most popularly the Newick format), they do not represent trees uniquely. For any tree over n taxa, there are $O(2^{n-1})$ Newick strings that can represent it. While TASPI has shown the benefits of compressing phylogenetic trees, the code is not publicly available, and the authors

can not always handle the multiple Newick representations that a tree can have. For the sake of efficiently storing trees, a format that efficiently represents a collection of trees in a normalized fashion would be of great value to the community.

Lastly, despite the availability of tree databases and other electronic methods for sharing trees, it is very difficult to procure the tree collection that was outputted by a phylogenetic analysis. TreeBase, while it contains data from over $2,500$ publications and $4,500$ authors, still has fewer trees in its entire database than many modern Bayesian analyses output in a single run. While Dryad purports to store data that is not stored in other sources, it too does not contain tree collections. This represents a massive loss of data, which could otherwise be used for verification of scientific results and as test data for software developers.

The achievements and limitations discussed in this chapter motivate much of the work in this thesis. Inspired by the strides of fellow researchers, we have sought to address the many challenges facing the comparison, storage and exchange of phylogenetic trees. In the following chapters, we will present methods that address and provide a solution for each of these current challenges.

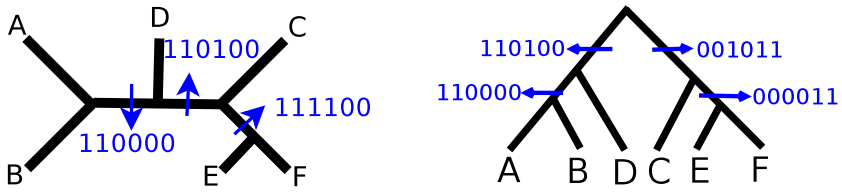
CHAPTER IV

UNIVERSAL HASHING OF PHYLOGENETIC TREES

The first step in helping biologists manage and share their large tree collections is to quickly extract the evolutionary relationships contained within the trees. We represent the shared relationships (or bipartitions) in a collection of trees through our novel use of hash functions. Hashing allows us to capture both the set of unique bipartitions and the set of unique trees. We also show how these hashing functions can be used to capture relationships between heterogeneous collections of trees. The hash functions studied in this section form the foundation of the powerful phylogenetic algorithms that analyze tens of thousands of trees presented in later chapters.

A. Universal Hashing of Phylogenetic Trees

How do we use universal hashing functions to determine the set of unique bipartitions in a collection of trees? Unlike regular hashing techniques, universal hashing selects a hash function h randomly from a class of hash functions, thus guaranteeing that the probability of collision is at most $\frac{1}{m}$, where m is the length of our hash table. The idea of using universal hashing functions to capture bipartition information contained in a collection of trees was first proposed by Amenta *et. al.* [76], as a method to construct a majority consensus tree in linear time. These functions were extended and heavily used in the design of the HashRF (computes the Robinson-Foulds distance matrix) [14, 61, 77] and HashCS (computes consensus trees) [78] algorithms developed by Sul *et. al.*, and upon which the work presented in this dissertation is based. This prior work demonstrated the power of these universal hashing functions for computing tree operations over large collections of trees. We extend the Monte Carlo randomized hashing scheme to be a Las Vegas approach, and show how it can be used to detect



(a) unrooted (b) rooted

Fig. 12. An illustration of how bipartitions are collected on an (a) unrooted and a (b) rooted representation of T_1 from Figure 10. For unrooted trees, $n-3$ bipartitions are collected. For rooted trees, $n-2$ bipartitions are collected. The extra collected bipartition determines the root.

unique trees and support heterogeneous collections of trees.

B. Using Hashing to Detect Unique Bipartitions

Bipartitions are first collected from each tree using depth-first traversal. We use a bitstring representation for our bipartitions in order to prepare them as input for our hashing functions. A particular bipartition can be represented uniquely as an n-bit bitstring, where each location in the bitstring corresponds to a particular taxon of interest. Taxa are ordered lexicographically for consistency, and unrooted trees are rooted according to their Newick representation. Thus, in Figure 12, the first bit represents taxon 'A', the second represents taxon 'B', and so forth. For unrooted trees, we collect $n-3$ bipartitions. For rooted trees, $n-2$ bipartitions are collected, with the extra bipartition used to establish the root (see Figure 12).

Prior to depth-first traversal, each trivial bipartition is assigned a bitstring of all zeros, save for the location corresponding to its taxa. Thus, trivial bipartition $A|BCDEF$ is assigned the bitstring 100000, bipartition $B|ACDEF$ is assigned the bitstring 010000 and so forth. During depth first traversal, the bitstring for each bipartition is recursively computed using the OR operation. Thus, bipartition $AB|CDEF$ has a bitstring representation of 110000. To ensure that no errors occur in the un-

rooted case, we enforce that the left most bit (in this case 'A') always has the value of 1. Hence for T_1's bipartitions of $AB|CDEF$, $ABD|CEF$, and $ABCD|EF$, the bitstring representations will be 110000, 110100, and 111100, respectively (Figure 12(a)).

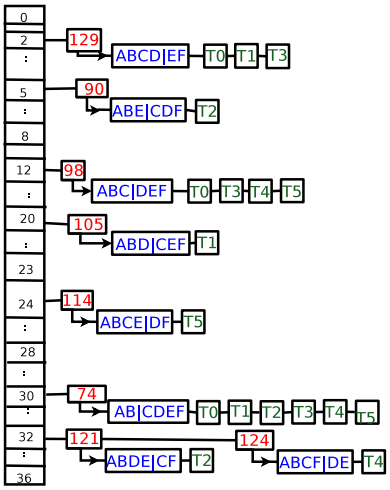
Let t denote the number of trees in our collection and n denote the number of taxa contained in each tree. For each taxon i from $1 \ldots n$, two random numbers are generated, r_i and s_i. These are used in conjunction with the bitstring and fed into universal hash functions, h_1 and h_2, which are used to calculate the hash table location of a bipartition and the corresponding bipartition identifier (BID) respectively. Let $\mathcal{B}_\mathcal{T}$ denote the set of bipartitions of T, and let B denote a bipartition of interest of tree T, such that $B \in \mathcal{B}_\mathcal{T}$. Then h_1 can be defined as

$$h_1(B) = \sum b_i \times r_i \mod m_1. \tag{4.1}$$

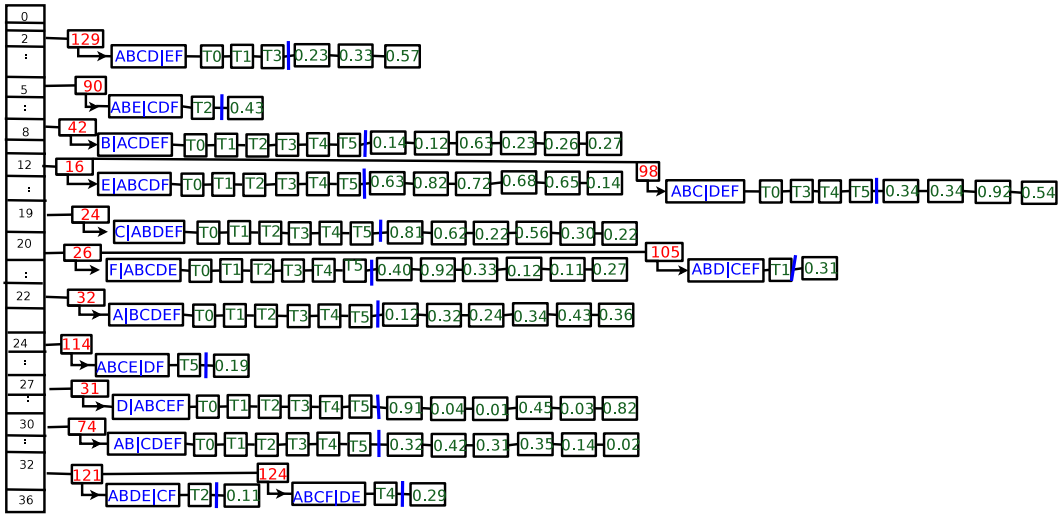
Here, b_i is the bit located in position i of B and r_i is the random number from set $\mathcal{R}$ corresponding with the taxa denoted by index i. Since there are at most $O(nt)$ unique bipartitions in a collection of t trees, m_1 is the first prime number larger than $n \times t$ and denotes the length of our hash table. The BID calculation is similar. h_2 is defined as

$$h_2(B) = \sum b_i \times s_i \mod m_2, \tag{4.2}$$

where s_i is the unique random number from set $\mathcal{S}$ associated with taxon b_i. Our calculation of h_2 will output a BID that can be stored in the hash table. m_2 denotes the maximum value of a bipartition identifier. We calculate m_2 as the first prime greater than $t \times n \times c$, where c is a large constant that bounds the probability of error arbitrarily close to 0 ($O(\frac{1}{n \times t \times c})$). Together, the h_1 and h_2 values represent a unique identifier for a particular bipartition in the tree. In the example that follows, $\mathcal{R} = (22, 45, 19, 27, 12, 20)$, $\mathcal{S} = (32, 42, 24, 31, 16, 26)$, $m_1 = 37$, and $m_2 = 3,607$.



(a) unweighted



(b) weighted

Fig. 13. An illustration of stored bipartitions in the hash table for the (a) unweighted and (b) weighted tree collections depicted in Figure 10.

Figure 13(a) shows the contents of the hash table after the bipartition information from our tree collection in Figure 10 is extracted and fed through our universal hashing functions. For example, bipartition $ABCD|EF$ (Figure 10) will be stored in hash table location $H[h_1(ABCD|EF)]$ (or $H[2]$). $h_2(ABCD|EF)$ (or 129) will be used as the BID[1], and the TIDs at this location are T_0, T_1 and T_3, indicating that this bipartition is contained in trees T_0, T_1 and T_3 (Figure 13(a)). Continuing the example, the hash table shows that there are 8 unique bipartitions ($AB|CDEF$, $ABC|DEF$, $ABCD|EF$, $ABD|CEF$, $ABE|CDF$, $ABDE|CF$, $ABCF|DE$, and $ABCE|DF$) out of 18 total.

For weighted trees, we also store branch lengths in our hash table, in an array that corresponds with our list of tree identifiers. For example, bipartition $ABCD|EF$ which is contained in trees T_0, T_1 and T_3 has edge weights of 0.23, 0.33 and 0.57 respectively in Figure 10(b). They are stored along with the tree identifiers in our hash table at location $H[2.129]$ (Figure 13(b)). Since weighted trees have weights along the trivial bipartitions, they are collected along with the $n-3$ non-trivial bipartitions, yielding $2n-3$ total bipartitions collected per tree. Trivial bipartitions are not collected in the unweighted case as all the trees have them.

C. Using Hashing to Detect Unique Trees

Let k denote the number of unique bipartitions in our set of t trees. We can represent each tree T as a k-bit bitstring (I), where each position denotes a unique bipartition in a collection. Ordering of bipartitions do not matter, so long as the ordering is consistent for all trees. In the context of our algorithms, we enforce an ordering based on the number of ones in the bitstring, with ties broken lexicographically. If

[1]For simplicity of explanation, we will refer to hash table locations as $H[h_1.h_2]$.

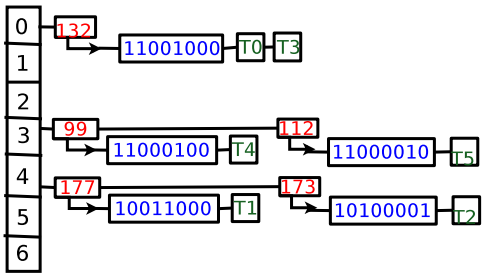


Fig. 14. An illustration of using hashing to detect the set of unique topologies contained in the tree collection from Figure 10.

the tree T contains bipartition j, then $I_j = 1$. Else, $I_j = 0$. We now generate two new sets of of k random numbers, $\mathcal{O}$ and $\mathcal{P}$, where o_j is the j^{th} random number in $\mathcal{O}$ and p_j is j^{th} random number in $\mathcal{P}$. We define a new hash function h_3 as

$$h_3(T) = \sum_{j=0}^{k} I_j \times o_j \mod m_3. \tag{4.3}$$

Here, h_3 corresponds to a hash table location in a new hash table, the length of which is m_3. Since there can only be t unique trees, m_3 is defined as the first prime greater than t. h_4 is used to generate a unique tree identifier (TID). It is defined as

$$h_4(T) = \sum_{j=0}^{k} I_j \times p_j \mod m_4. \tag{4.4}$$

Since there can be at most t identical trees, m_4 is the first prime greater than $t \times c$. Thus, we can once again bound the probability of double collision to be an arbitrarily small $O(\frac{1}{c})$ by choosing a large c value.

Figure 14 depicts the result of hashing the six trees in figure 10 through h_3 and h_4. In this example, $\mathcal{O} = (16, 11, 41, 29, 1, 32, 4, 24)$, $\mathcal{P} = (81, 12, 50, 57, 39, 6, 19, 42)$, $m_3 = 7$, and $m_4 = 601$. When we feed our six trees through these two new hash functions, both trees T_0 and T_3 hash to location $H[0.132]$, since their k-bitstring representations are identical. All the other trees hash to different locations. A linear

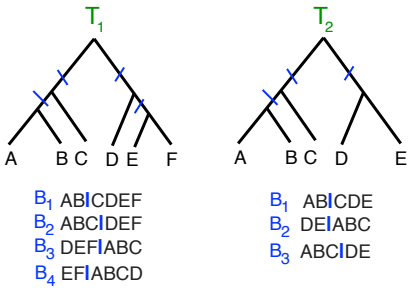


Fig. 15. Bipartitions in a heterogeneous collection of trees. How do we hash them?

traversal of this new hash table reveals that there are 5 unique trees, and T_0 and T_3 share the same topology.

D. Supporting Heterogeneous Trees

To extend the usefulness of hashing for representing relationships between trees, we need to look beyond homogeneous collections of trees (where all trees in the collection contain the same n taxa each) to heterogeneous collections. In a heterogeneous collection of trees, trees have different taxa. For example, the two trees in Figure 15 are heterogeneous. *How can we capture relationships contained in heterogeneous collections of trees?*

To detect similarity between heterogeneous trees, we introduce the concept of a partial bitstring. A partial bitstring is a bitstring representation of a bipartition in which the bitstring is just as long as the last (right-most) '1' bit. Thus for T_1 in Figure 15, bipartition $AB|CDEF$ will be stored as `11`, bipartition $ABC|DEF$ will be stored as `111`, bipartition $DEF|ABC$ will be stored as `000111`, and bipartition and $EF|ABCD$ will be stored as `000011`. For T_2 in Figure 15, bipartition $AB|CDE$ will be stored as `11` and bipartition $ABC|DE$ will be stored as `111`, and bipartition $DE|ABC$ will be stored as `00011`. For each of these cases, the bitstring representation

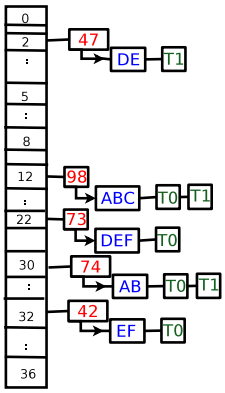


Fig. 16. An illustration of stored bipartitions in the hash table for the heterogeneous trees shown in Figure 15.

is only as long as the right-most '1'-bit.

However, note that both bipartition $AB|CDEF$ and $AB|CDE$ have the representation `11`. The reason may not be immediately intuitive in the normal definition of the bipartition. Interpreting this bitstring in the classical sense would be "A and B are on one side and C, D, E and F are on the other." With this interpretation, `110000` is not a valid representation for $AB|CDE$, since F is not grouped together with taxa C, D and E, as it does not exist in this tree. However if we interpret bitstring `11` as "A and B form a group/clade that C, D, E and F is not part of" the representation makes sense. For T_1, taxa A, B are in the tree, but not grouped with taxa C, D, E, and F. For T_2, A B, are in the tree, but not grouped with taxa C, D and E. Since F is not in T_2 it automatically follows that it is not grouped with taxa D and E, and therefore `11` is a valid representation for this relationship. We note that we treat all heterogeneous trees as being rooted. Thus, unrooted heterogeneous trees are treated as rooted trees in the context of these hashing functions.

Heterogeneous trees can now be hashed together using the hash functions h_1 and h_2. Assuming the same random numbers for $\mathcal{R}$ and $\mathcal{S}$ from before, Figure 16 shows the result when hashing using this scheme. Using this scheme, T_1 and T_2

both share relationships `11` (AB) and `111` (ABC). T_1 has the unique relationships `000011` (EF) and `000111` (DEF). T_2 has the unique relationship `00011` (DE). The ramifications of being able to hash heterogeneous trees are monumental. By being able to identify relationships between heterogeneous collections of trees, we can now perform operations quickly on these datasets. For example, we can compute the RF distance between trees T_1 and T_2. With two bipartition in common, the Robinson-Foulds distance between T_1 and T_2 is $D_{RF}(T_1, T_2) = \frac{2+1}{2} = 1.5$. In later chapters, we will show how this allows us to compare and compress heterogeneous trees efficiently.

E. Handling Hashing Collisions

A consequence of using hash tables is the potential for collisions. Type 0 collisions occur when multiple input trees share a common bipartition, which is a consequence of the input, and not considered an issue. For example, the six tree identifiers hashing to location $H[30.74]$ in Figure 13 is an example of type 0 collision, as is the four tree identifiers hashing to location $H[12.98]$. Type 1 collision occurs when multiple bipartitions hash to the same hash line (though have different BIDs), which is a consequence of hashing. The probability of type 1 collisions is $\frac{1}{m_1}$. In Figure 13(a), the bipartitions with BIDs 121 and 124 are an example of type 1 collisions, as they both reside on hash line 32.

Type 2 (double) collision occurs when two different bipartitions are assigned the same BID and hash to the same hash line. That is, $h_1(B_i) = h_1(B_j)$ and $h_2(B_i) = h_2(B_j)$ and $B_i \neq B_j$. This would be a serious error, and would mean that our results are incorrect. Under the Monte Carlo version of this algorithm, the probability of type 2 collision is $\frac{1}{m_1 \cdot m_2}$, which can be bounded to an arbitrarily small value of $O(\frac{1}{c})$ by choosing very large c values in setting the m_2 value.

The primary difference between a Monte Carlo and a Las Vegas randomized algorithm is that the latter guarantees 0% error, while the former arbitrarily bounds the probability of error close to 0%. A Las Vegas approach was not utilized in previous approaches such as HashRF and HashCS due to memory concerns. By choosing a very efficient way of storing bitstrings, the algorithms presented in this dissertation store an explicit representation of the bipartition in our hash table, with minimal overhead on performance. Thus, we can always check if $B_i = B_j$. This guarantees us 0% probability of error, and makes our hashing strategy a Las Vegas approach. To eliminate the possibility of non-termination, our code terminates and notifies the user to restart the approach if a Type 2 error is ever detected. In practice, this has never occurred.

F. Summary

In this chapter, we discussed our universal hashing functions for detecting unique evolutionary relationships between a group of trees. We then showed how these hashing functions have been extended to detect the set of unique trees in a collection. Representing bipartitions as *partial bitstrings* allows us to hash heterogeneous collections of trees. Lastly, we show how collisions are handled, and how storing an explicit representation of the bitstring in the hash table augments the previous Monte Carlo hashing strategy to a Las Vegas approach, guaranteeing 0% error.

Hashing allows us to quickly capture the shared relationships contained in a collection of trees. Once this information is uniquely captured, operations can be performed rapidly on this data. In the context of phylogenetic analysis, we can quickly compare trees to each other, or compute consensus trees. We can also quickly determine the number of unique trees in a collection. The ability to represent het-

erogeneous trees in this context means that operations designed for homogeneous datasets (e.g. tree comparison) can now be applied without additional overhead to heterogeneous collections of trees. To the best of our knowledge, there are no other algorithms that perform operations on heterogeneous collections of trees, underscoring the critical importance and novelty of our work.

The hashing functions discussed here form the basis of all the work discussed in this dissertation. In upcoming chapters, we will show how the hash table allows us to compare trees very quickly. We also show how an encoded representation of the hash table allows us to store tree collections in a very space-efficient textual format. The ability to detect unique trees plays a critical part in the structure of this format, and also forms the basis of our merge algorithm in our Noria system. Lastly, the ability for the hash table to handle heterogeneous trees opens up a whole new world of possibilities. In later chapters, we will also discuss how this ability allows us to compare heterogeneous collections of trees and store them efficiently.

CHAPTER V

EFFICIENT ALGORITHMS FOR COMPARING LARGE GROUPS OF PHYLOGENETIC TREES

In Chapter II we showed an example of how topological matrices can be used to elucidate information that would otherwise be hidden by a single consensus tree, such as determining if a phylogenetic search actually converged. In order for tree comparison to be used in the context of phylogenetic search and other applications, it must be performed in real-time. In this chapter, we present MrsRF [12], a fast parallel algorithm capable of computing large RF matrices in real-time. Furthermore, we present Phlash, a fast and novel algorithm that is capable of comparing trees using a variety of different methods. Together, these two approaches offer biologists a variety of new ways to study the information contained in their tree collections.

A. Motivation

To compare t trees, scientist commonly construct an RF distance matrix, where every cell (i, j) represents the RF distance between trees T_i and T_j. However, as the number of trees t grows very large, computing large $t \times t$ distance matrices becomes very difficult. If we attempt to store the entire $t \times t$ matrix in memory, the $O(t^2)$ memory footprint quickly becomes a bottleneck and restricts the maximum number of trees that can be computed. *How can we construct large topological distance matrices in real-time?*

Furthermore, in Chapter III we demonstrated that one of the limitations of RF distance is that the presence of a single rogue taxon can produce maximally distant trees. *Are there other metrics that can be used to compare trees?* Our solutions MrsRF and Phlash are the fastest in the field for comparing large collections of trees.

Furthermore, our Phlash algorithm is capable of comparing trees using a variety of distance and similarity methods. Our Phlash algorithm is also capable of comparing large groups of weighted phylogenies, something no other available method can do.

B. MrsRF: Using MapReduce for Comparing Trees with Robinson-Foulds Distance

MrsRF(MapReduce Speeds up RF) is a MapReduce based algorithm for computing the all-pairs Robinson-Foulds distance between t evolutionary trees on multi-core computing platforms. To get around the memory bottleneck, MrsRF divides the global $t \times t$ matrix into a series of sub-matrices which are then computed independently via MapReduce.

MapReduce is a paradigm popularized by Google for designing parallel applications. The *novelty* of our work centers around using MapReduce in a non-standard way. Typical uses of the MapReduce framework reduce the final output into a smaller representation than the initial input. One of the interesting aspects of the all-pairs RF distance problem is that the output size (a $t \times t$ RF matrix) is much larger than the input size (t phylogenetic trees). Under the all-pairs RF problem, we are significantly expanding the data.

In addition to enabling RF matrices to be computed in real-time, MrsRF shows the applicability of MapReduce for creating fast algorithms for phylogenetic analysis. MrsRF is free to download from `http://mrsrf.googlecode.com`. As of March 2012, MrsRF has been downloaded 95 times [79].

1. MapReduce

MapReduce [80] is an exciting new paradigm for designing parallel applications. It was popularized by Google to support the parallel and distributed execution of data

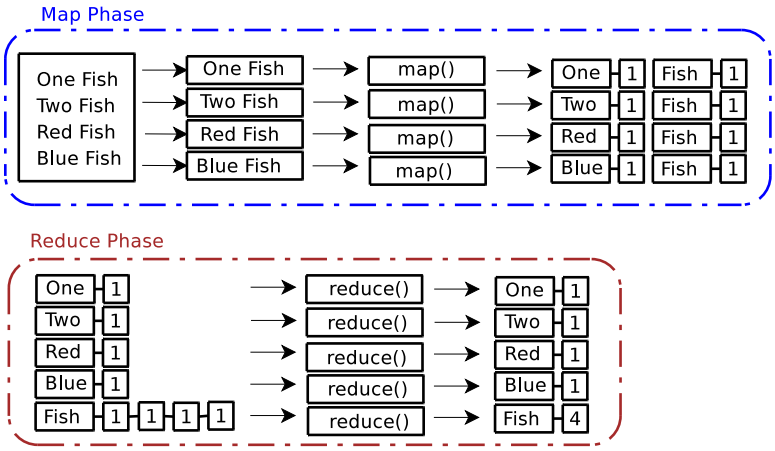


Fig. 17. Word count example using the MapReduce paradigm.

intensive applications. To process petabytes of data, Google executes thousands of MapReduce applications per day. The MapReduce model is used for a variety of applications, such as distributed sort and grep, Google web indexing, and data processing by large companies such as Amazon, Yahoo! and Facebook. Furthermore, there is interest within the bioinformatics community to harness the power of MapReduce to develop parallel applications to process large datasets of genomic data. For example, CloudBurst [81], a MapReduce application for sequence analysis, has recently been released.

The central features of the MapReduce framework are two functions: `map()` and `reduce()`. The `map()` function produces a set of intermediate key/value pairs. The `reduce()` function accepts an intermediate key and a set of values and merges them together. Both the `map()` and `reduce()` functions are written serially by the programmer. The underlying MapReduce framework takes care of scheduling these functions on the multi-core system.

Figure 17 shows an overview of how the MapReduce paradigm operates in order to count the number of words in a file. We have as input the title of a popular

children's book, with two words located on each line. The master splits the file line by line, assigning each instance of the map function (or *mapper*) one line of input. Each mapper takes its line of input and splits it into words. The mapper then outputs a (key, value) pair of the word and the value 1 (indicating that the word was found exactly once). Since all the lines are independent from each other, all mappers are run in parallel. If the number of lines were to exceed the number of designated mappers, then the master would delegate new lines to available mappers after each mapper has processed its given input, until all lines are exhausted. As each mapper outputs (key, value) pairs, these pairs are merged to form keys with associated lists of values. In the reduce phase, each instance of the reduce function (or *reducer*) takes as input a key and associated list of values.

For example, in Figure 17, our first reducer takes as input the key `one` and the associated list `1`. This list contains only a singleton, since the word "one" only occurred once in our input. In contrast, our fifth reducer takes in as input the key `fish` and the associated list `1-1-1-1`, as "fish" occurred four times in the input file. Each reducer takes its list of values, sums all of its member's values, and outputs this sum of values with the key. Thus, the output of reducer 5 will be `fish-4`.

a. Phoenix: a MapReduce library

The underlying MapReduce framework of MrsRF is based on Phoenix [82] v.1.0, a multi-core MapReduce approach. Phoenix is a threads-based implementation of MapReduce designed specifically for multi-core systems where all computing cores have access to shared memory. It dynamically schedules map and reduce tasks to available compute cores. Consider a $N \times c$ multi-core cluster configuration, where N represents the number of physical nodes (or machines) of the cluster and c is the number of computing cores per node. Each of the cores on a node share access

to memory. Phoenix works on a $1 \times c$ configuration. We augment Phoenix with OpenMPI in order to use distributed-memory clusters, where $N \geq 1$.

Hadoop [83] is the most popular framework for developing MapReduce applications. While we had developed MrsRF's implementation in both Phoenix and Hadoop, we discovered that we were able to achieve significantly better performance using Phoenix over Hadoop. There are other advantages for using Phoenix. Hadoop often requires a dedicated cluster, and for programs that are not data-intensive on the input, the overhead of writing to the HDFS greatly outweighs any potential benefits. Phoenix, on the other hand, is easy to install.

We hybridized Phoenix with Open-MPI for the sake of our application, allowing it to run on large clusters. This hybrid implementation also makes MrsRF portable, allowing the user to run the program locally on a multi-core environment before testing it out on a large cluster. Furthermore, the framework's ANSI C implementation and combination with OpenMPI allows it readily interface with most modern clusters, since the Pthreads library is ubiquitous to Unix-like systems, and MPI is a standard.

2. Algorithm Description

The design of MrsRF is motivated by the HashRF algorithm. Moreover, in MrsRF, bipartitions are analogous to words in the MapReduce word count example. MrsRF takes as input a tree file containing t trees and a $N \times c$ cluster configuration. The number of cluster nodes specifies the number of physical machines that executes the code. The number of cores is the number of CPUs within each node. For serial execution, $N = 1$ and $c = 1$. If instead one wanted to run MrsRF on 2 machines each containing 4 CPUs, the respective N and c values would be 2 and 4.

There are two main steps to our MrsRF algorithm. First, we organize the N nodes into a grid in order to partition the t input trees among the nodes. Phoenix,

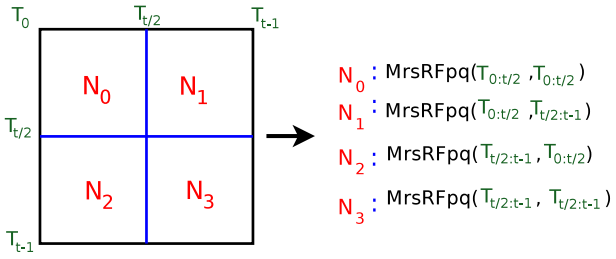


Fig. 18. Global partitioning scheme of the MrsRF algorithm. Here, $N = 4$. Each of these 4 sub-matrices will be calculated by a separate instance of MrsRF(p, q).

the underlying MapReduce library, automatically distributes the input for a single node amongst its c cores. That is, it works for $1 \times c$ cluster configurations. As a result, we manually partition the input among the N nodes. If N is a perfect square, then we assume the nodes are organized into a $\sqrt{N} \times \sqrt{N}$ grid. If N is not a perfect square, let $i = \lfloor \sqrt{N} \rfloor$. If $N \mod i = 0$, then we assume a $N/i \times i$ grid of nodes. Otherwise, we decrement i until it divides N evenly[1].

For $N = 4$, the N nodes are partitioned into a 2×2 grid (see Figure 18). If $N = 18$, we obtain a 6×3 grid. The size of the input tree file has no bearing on how the N nodes are organized into a grid. Secondly, once the nodes are organized in a grid using OpenMPI, the MrsRF(p, q) algorithm is executed on each node to compute a $p \times q$ sub-matrix. For example, consider node N_2 in Figure 18. Let $\mathcal{P}$ and $\mathcal{Q}$ represent the row and column trees, respectively, in the sub-matrix. For node N_2, $\mathcal{P} = \{T_{t/2}, \ldots, T_{t-1}\}$ and $\mathcal{Q} = \{T_0, \ldots, T_{t/2\ -\ 1}\}$. Hence, the size of node N_2's sub-matrix is $p \times q$, where $p = |\mathcal{P}|$ and $q = |\mathcal{Q}|$.

Once each node is finished computing its $p \times q$ sub-matrix, the final $t \times t$ RF

[1]Note that a suboptimal solution will be reached with prime numbers.

matrix is the concatenation of the N sub-matrices.

3. MrsRF(p, q): Computing a $p \times q$ RF sub-matrix

The heart of our MrsRF algorithm lies in the subprogram MrsRF(p, q), which runs independently on each of the N nodes. Each node has access to the input file containing the t trees and is responsible for retrieving the appropriate trees for its $\mathcal{P}$ and $\mathcal{Q}$ sets. A node knows which trees belongs to its $\mathcal{P}$ and $\mathcal{Q}$ sets based on its identifier within the node grid. In Figure 18, $\mathcal{P} = \{T_{t/2}, \ldots, T_{t-1}\}$ and $\mathcal{Q} = \{T_0, \ldots, T_{t/2\ -\ 1}\}$ on node N_2. If the number of computing cores on node N_2 is eight, then the trees associated with sets $\mathcal{P}$ and $\mathcal{Q}$ will each be split into four files, yielding a total of eight input files for the eight cores. Under MapReduce, these eight files will be automatically assigned to the cores on node N_2.

Under MrsRF(p, q) the trees in $\mathcal{P}$ are compared to the trees in $\mathcal{Q}$. If $\mathcal{P} = \mathcal{Q}$, all trees are compared to each other. Node N_i's sub-matrix is created in parallel using two MapReduce phases as described below.

a. Phase 1 of the MrsRF(p, q) algorithm

The first map stage. Similarly to HashRF, the first MapReduce phase is responsible for generating the global hash table. That is, every bipartition is given a unique identifier (key). Its values are the tree identities (TIDs) that contain that bipartition. The number of mappers corresponds to the number of cores utilized on a particular node. Each mapper sends its trees to HashBase to create a local hash table. HashBase is our name for a modification to HashRF that outputs a hash table from its input trees. Each line from the hash table that is provided to MrsRF(p, q) from HashBase consists of a bipartition B_i and the associated list of tree ids that were found to share it. In addition, all the bipartitions that are found are given a marker to denote which

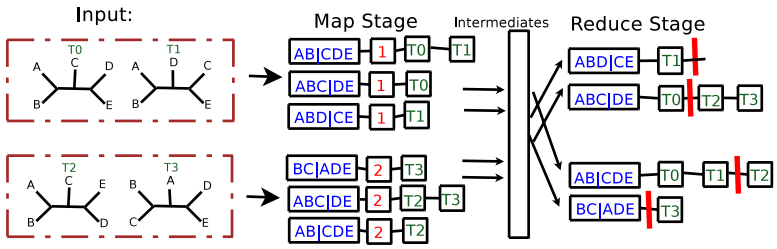


Fig. 19. Phase 1 of the MrsRF(p, q) algorithm. Two mappers and two reducers are used to process the input files, where $\mathcal{P} = \{T_0, T_1\}$ and $\mathcal{Q} = \{T_2, T_3\}$.

input file created it. This is to ensure that bipartitions shared within a tree file are not compared to each other. The bipartition and its list of tree ids form a (key, value) pair, which is emitted as an intermediate for the reduce stage.

In Figure 19, there are two input files, each containing two trees each. Trees in the first file are only compared to trees contained in the second file. In the figure, we assume there are only two mappers, where each mapper is responsible for handling one of the two input files. Each mapper creates a local hash table based on the trees that it receives by using a marker of "1" (for file 1) or "2" (for file 2) to keep track of trees and their bipartitions from the $\mathcal{P}$ and $\mathcal{Q}$ sets, respectively. Each mapper then emits its marked hash table to the reduce stage. For example, in Figure 19, the first mapper emits the following (key, value) pairs: $(AB|CDE,\ (1,\ T0,\ T1))$, $(ABC|DE,\ (1,\ T0))$, and $(ABD|CE,\ (1,\ T1))$. These (key, value) pairs are processed in an intermediate stage, where each reducer processes all of the values associated with a particular key.

The first reduce stage. Once the map stage completes, each of the r reducers takes as input a (key, list(value)) pair with bipartition B_i as the key and a list of tree id

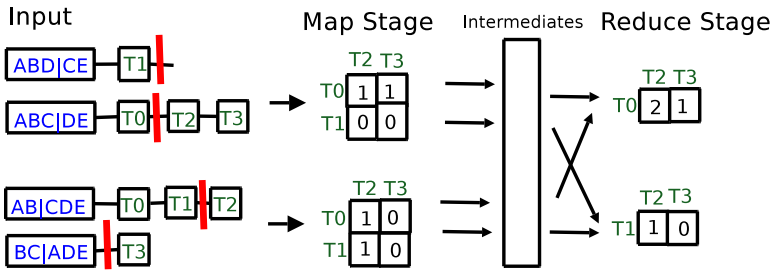


Fig. 20. Phase 2 of the MrsRF(p, q) algorithm. Once again, there are two mappers and two reducers. The horizontal bars between elements represent a partition that separates trees from one file from trees in the other. Bipartitions containing elements on only one side of the partition are discarded.

lists as the value. There will be at most m lists of tree ids for each bipartition. Each reducer then combines these $O(m)$ lists in a manner such that the trees from file 1 are separated from the trees from file 2 to form a single line in the global hash table. Each row of the global hash table represents a unique bipartition among the t trees. Continuing with our example from Figure 19, the first reducer processes the lists associated with keys $ABD|CE$, and $ABC|DE$. Thus, the first reducer receives the list of lists $(ABD|CE,\ (1,\ T1))$ and $(ABC|DE, (1,\ T0),\ (2,\ T2,\ T3))$ and outputs the final (key, value) pairs of $(ABD|CE,\ (T1\ ||))$, and $(ABC|DE,\ (T0\ ||\ T2, T3))$, respectively. The symbol $||$ denotes a partition that separates trees from the first input file with trees from the second input file.

b. Phase 2 of the MrsRF(p, q) algorithm

The second map stage. Each of the m mappers receives an equal portion of the global hash table based on the total number of comparisons required to process the (key,value) pairs. In Figure 20, key $(ABC|DE)$ has as its list of values $(T0||T2, T3)$.

Two total comparisons will be done since $T0$ is compared to $T2$ and $T3$. In general, if there are u tree ids on the left side of $||$ and v trees on the right side, then $u \times v$ total comparisons are required. Each mapper then computes a local similarity matrix from its portion of the hash table. In the second reduce stage, this similarity matrix will be converted into a RF (or dissimilarity) matrix.

Consider Figure 20. The first mapper has two rows of the global hash table assigned to it. Next, it computes a $p \times q$ similarity matrix from those hash table elements. In our example, the resulting similarity matrix is of size 2×2. To do this, it compares the tree ids in the first partition of a row to the tree ids located in the second partition of a row, and increments the local similarity matrix accordingly. For example, for the hash table row $(ABC|DE,\ (T0\ ||\ T2,\ T3))$, the first mapper increments by one the locations $(T0,\ T2)$ and $(T0,\ T3)$ in its local similarity matrix. Rows that do not contain elements in both of its partitions are discarded. Therefore, in Figure 20, the hash table row $(ABD|CE,\ (T1\ ||))$ is discarded. Each increment to entry (i, j) in the similarity matrix represents that the pair of trees Ti and Tj share a bipartition.

Once a mapper has finished processing its hash table, it emits its similarity matrix for processing in the reduce stage. The key is the row id and the value is the contents of that row id. Thus, in Figure 20, the first mapper emits the (key, value) pairs $(T0,\ (1,\ 1))$, and $(T1,\ (0,\ 0))$.

The second reduce stage. Here, the input is a similarity matrix row identifier, i, and a list of rows that contain the local similarity scores found by each of the mappers. For a particular key, the number of rows within the list of rows received by a reducer is equal to the number of mappers, m. In Figure 20, the first reducer receives the following (key, value) pair for similarity identifier, $T0$: $(T0, (1, 1), (1, 0))$. The reducer

sums up the columns of each of the lists to produce $(T0, (2, 1))$. To produce the RF distance for row T_i, each column in the final similarity matrix is subtracted from $n - 3$, the maximum possible RF distance where n represents the number of taxa in the t trees of interest. Together, the output from the reduce stage yields a final sub-matrix. For each node N_i, the resulting sub-matrix is written to a file. These files can then be combined to form a final RF matrix, or be kept in their partitioned form for easier handling.

4. Algorithm Analysis

MrsRF(p, q) is where all of the computation for the MrsRF algorithm lies. At least one node will require $O(t)$ time to obtain the trees for its $\mathcal{P}$ and $\mathcal{Q}$ sets. The first map phase of the MrsRF(p, q) algorithm, which is based on HashRF's first phase, requires $O(\frac{(n(p+q))}{m})$, where n is the number of taxa and m is the number of mappers. $O(n(p+q))$ is the total number of bipartitions that must be processed across the $p+q$ trees and inserted into the hash table. Suppose b unique bipartitions are found. In the worst case, a bipartition has a length of $p+q$, which reflects the fact that it appears in all $p+q$ trees. Hence, the complexity is $O(\frac{b}{r}(p+q))$ for the first reduce phase, where r is the number of reducers. For the second phase of the MrsRF(p, q) algorithm, in the worst case, each mapper requires $O(\frac{bpq}{m})$ to produce its local similarity matrix. Each reducer requires $O(\frac{mpq}{r})$ time. Hence, if p and q are large enough, phase 2 is more time-consuming than phase 1 in the MrsRF algorithm. Our analysis does not incorporate communication costs as there is not an explicit model of communication for the MapReduce framework.

5. Experimental Analysis

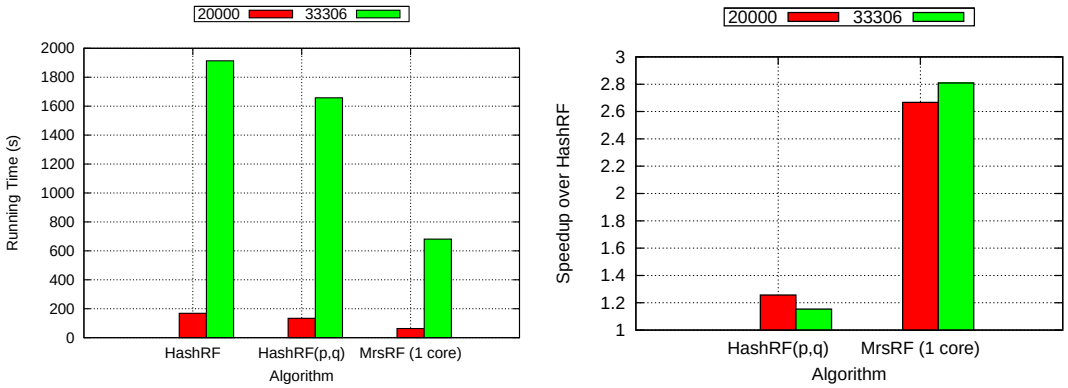
We ran our experiments on 20,000 and 33,306 biological tree collections consisting of 150 and 567 taxa, respectively. MrsRF was implemented using Phoenix [82], a MapReduce implementation for shared memory multi-core platforms, and Open-MPI [84]. Our results show that MrsRF is a promising methodology for parallelizing the all-pairs RF distance problem. In our experiments, MrsRF shows good overall speedup. On 8 cores, MrsRF is over 6 times faster than the best-performing sequential algorithm, which is also MrsRF run on a single core. For 32 cores, it is 18 times faster than the serial version of MrsRF. Speedup resulted from allowing the underlying MapReduce runtime system to schedule communication on the multi-core system, which greatly simplifies MrsRF's implementation.

a. Cluster configurations

To test how much a factor architecture is to speedup, we used different system configurations to measure performance. Let N denote the number of nodes used, and c denote the number of cores used on each node. For any $N \times c$ configuration, there are Nc total cores being utilized. Thus, for 8 total cores to be used, we run our algorithm using 1×8, 2×4, 4×2 and 8×1 configurations. Each curve denotes the number of cores utilized per node. Therefore, if $c = 4$ and the total number of cores is 8, then this data point reflects a 2×4 configuration. Likewise, if $c = 1$ and the total number of cores is 32, then the data point reflects a 32×1 configuration.

b. Establishing the fastest sequential algorithm

We evaluate the performance of MrsRF on our computational platform as we vary the number of cores, the number of nodes, and the problem size of interest. First,

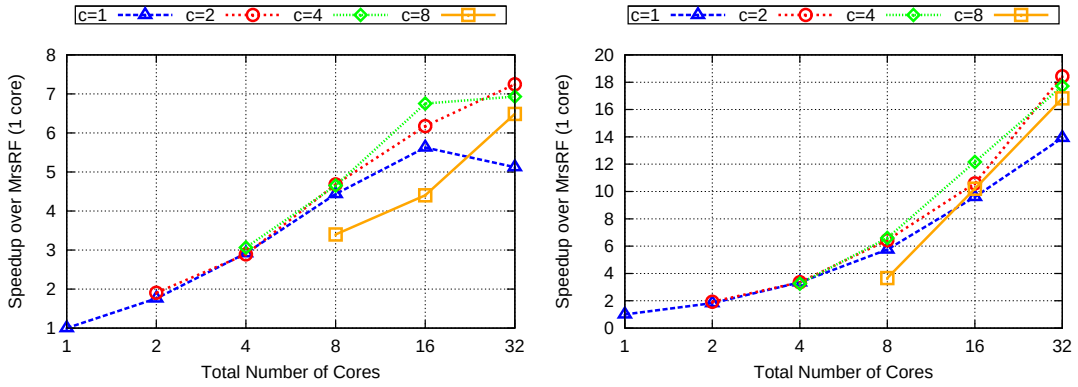


(a) Running Time (b) Speedup

Fig. 21. Sequential running time and speedup of HashRF, HashRF(p, q), and MrsRF (1 core) algorithms on $20,000$ and $33,306$ trees on 150 and 567 taxa, respectively.

we establish the fastest sequential algorithm in order to compute the speedup of our approach. Speedup is defined as $\frac{T_1}{T_{N\times c}}$ where, T_1 is the time required by the fastest sequential program and $T_{N\times c}$ is the time required by MrsRF run on N nodes and c cores. Previous experiments established HashRF and HashRF(p, q) as the fastest sequential algorithms for computing the RF matrix [14, 77].

Figure 21 compares the sequential running time of HashRF, HashRF(p, q), and MrsRF. Each data point represents the average of five runs of the algorithm for each dataset. Surprisingly, our experiments showed that our MrsRF algorithm using 1 core is up to 2.8 times faster than HashRF on larger tree sets. This corresponds to an average time of 680.9 seconds for MrsRF compared to a running time of 1913.22 seconds for HashRF, and a running time of 1657.75 seconds for HashRF(p, q). The difference in performance is due to language-specific implementation decisions. MrsRF is written in C to match the implementation language of Phoenix. HashRF and HashRF(p, q), on the other hand, are C++ implementations that employ the Standard Template Library (STL) and classes, which introduces extra overhead when



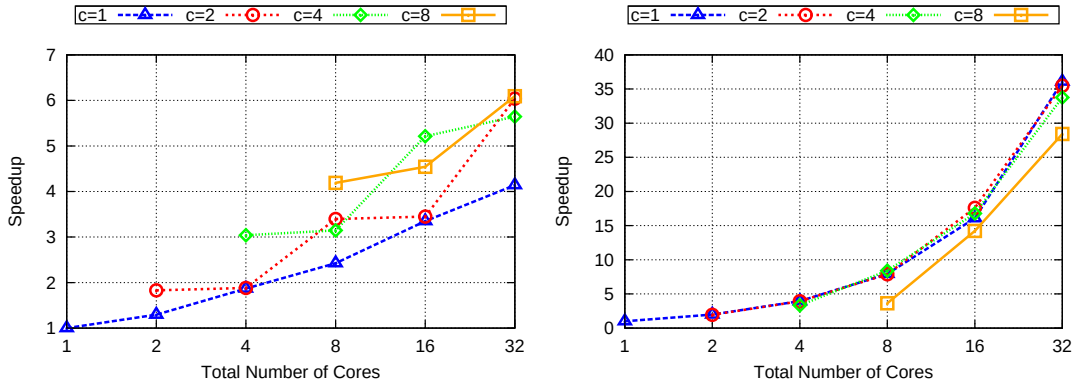
(a) 150 taxa (b) 567 taxa

Fig. 22. Speedup of MrsRF algorithm on various $N \times c$ multi-core cluster configuration.

compared to MrsRF's ANSI C implementation. Thus, all our speedup results are in relation to MrsRF run on a single core.

c. Multi-core performance on `freshwater` trees

Figure 22 shows that as the number of bipartitions increase, so does the performance of MrsRF. For our 150 taxa case, we attain perfect speedup on the 2 core case, and near-linear speedup through the 8 core case. At this point, we begin observing differences in the performance of MrsRF, given the base architecture. While the curves for $c = 2$ and $c = 4$ performs the best, the $c = 1$ and $c = 8$ configurations performs the poorest. Performance differences across various cluster configurations are underscored as the total number of cores increases. This is due to overhead in partitioning the data ($c = 1$) and inter-node resource contention for memory bandwidth ($c = 8$). Despite this, an increase in total cores results in an increase in performance, and we see our best performance when 16 total nodes is utilized, with a maximum speedup of 7.25 for this dataset. This corresponds to an average running time of 8.73 seconds for MrsRF on 32 cores.



(a) Phase 1 (b) Phase 2

Fig. 23. Performance of Phase 1 and Phase 2 of MrsRF(p, q) on 567 taxa and 33,306 trees.

d. Multi-core performance on `angiosperms` trees

We see a similar trend in architectural performance in the 567 taxa case, thus underscoring the importance of managing resource contention and communication overhead in relation to performance. However, with the increased bipartitions present in the 567 taxa case, we see a markedly increased amount of speedup, with a maximum amount of speedup of 18.4 attainable with 32 cores using a 16×2 cluster configuration. The maximum speedup corresponds to an average running time of 36.93 seconds. In comparison, it took the serial execution of MrsRF an average of 680.9 seconds to compute the RF matrix, while it took HashRF and HashRF(p, q) an average of 1913.22 and 1657.75 seconds respectively. Our results show that MrsRF is a very scalable approach for computing the all-to-all RF Matrix, with performance increasing with large problem sizes. Figure 23 shows that Phase 2 of the MrsRF(p, q) approach exhibits linear speedup. Overall speedup of MrsRF increases (decreases) when Phase 2 (Phase 1) of MrsRF(p, q) dominates the computation time. Once again, the differences in speedup that we observe with different $N \times c$ configurations suggest

that multiple cluster configurations should be run to achieve the maximum speedup.

C. Phlash: Advanced Topological Comparison

The *Phylogenetic Hashing (or Phlash)* algorithm is a fast serial algorithm for comparing large groups of phylogenetic trees. It is faster than MrsRF executed on one core, and removes the need to allocate a $t \times t$ matrix in memory. This is accomplished through the use of two novel data structures in tandem: the *compact table* and the *inverted index*. By using these data structures together, we can (a) reduce the number of comparisons needed to compare t trees, and (b) we can compute the $t \times t$ matrix one row at a time, reducing the $O(t^2)$ memory footprint to $O(t)$. This allows us to compute matrices of arbitrary size.

Furthermore, we explore new methods for assessing the topological diversity in a collection of trees by utilizing a feature-based bitstring representation of a phylogenetic tree. The presence or absence of these features (bipartitions) can be summarized using four numeric quantities: a, b, c and d. Any number of distance and similarity measures between trees can be written as a function of these four values. Phlash can quickly compute a distance or similarity matrix of any coefficient that is a function of a, b, c and d. In this section, we also explore the viability of 24 widely used coefficients for quantifying the topological richness in a collection of trees. Our results demonstrate the usefulness of these different coefficients, and suggest a motivation to look beyond traditional distance measures for analyzing collections of phylogenetic trees.

1. The ABCs of Topological Comparisons

We develop a general framework for designing similarity and distance measures for phylogenetic trees based on their bitstring representation. There is a vast literature of similarity and distance measures that are based on bitstring representations [85, 86, 87], which includes measures used in phylogenetics such as the Robinson-Foulds distance. Our Phlash algorithm leverages these measures by representing trees efficiently as bitstrings. Certainly, the idea of representing trees by their bitstring representation is not new. Each edge of a tree separates a subtree from the root, and this subtree contains a subset of the taxa. If we list all possible subsets of the n taxa and order them, we can create a bitstring for a tree. We would put a 0 for the subset if tree T_i contained that edge and a 1 if it did not contain that edge. Unfortunately, using this representation, the size of a bitstring for any given tree would be $O(2^{n-1}-n-1)$, where n is the number of taxa. For most trees with the number of taxa, n, of interest, such a bitstring representation is infeasible.

Our Phlash algorithm uses an m-bit bitstring representation of the trees, where m is equal to the number of unique bipartitions in a collection of interest. The intuition behind this representation is to only have bipartitions that appear in the collection of trees as features of the tree bitstrings. In practice, $m << 2^{n-1} - n - 1$ since the trees from a phylogenetic analysis share many bipartitions. For example, the 6 trees from Figure 10(a) contain 8 unique bipartitions and thus can be represented with a 8-bit bitstring.

a. Computing similarity and distance between bitstrings

Recent work [85, 86] provides us with over 75 similarity and distance measures (or coefficients) used over the last century for determining similarity and difference of

Table II. A contingency table showing the relationship of the variables a, b, c, d and m for two objects X and Y.

		Y		
		presence (or 1)	absence (or 0)	total
X	presence (or 1)	a	b	$a+b$
	absence (or 0)	c	d	$c+d$
	total	$a+c$	$b+d$	$m=a+b+c+d$

binary strings. These measures are each expressed as a function of five variables: a, b, c, d, and m. Table II shows the relationship of these variables to each other. In comparing any pair of bitstrings X and Y: a represents the number of features present in both X and Y, b is the number of features in X but not Y, c is the number of features in Y but not X, d is the number of features not present in either X or Y, and m is the total number of bits (i.e., $m = a + b + c + d$). Here, m represents the number of unique bipartitions in the input collection of trees. The minimum value that m can take on is the number of non-trivial bipartitions in an unrooted tree, or $n-3$, where n is the number of taxa in that tree. This occurs if all the trees in the collection are identical. If trivial bipartitions are of interest (as it is in the weighted case), the total number of bipartitions in a tree (and thus the minimum value of m) is $2(n-1)$.

Table III shows a sample of 24 similarity and distance measures expressed in their algebraic formulation. Consider the following bitstrings representations for trees T_0 (10010100) and T_1 (10000110). We observe here that $a = 2$, $b = 1$, $c = 1$, and $d = 4$. In general, for binary trees, $b = c$. Thus, the Dice similarity ($C3$ in Table III) between T_0 and T_1 is 0.67, and the RF distance ($C24$ in Table III) is 1. More specifically, in terms of bipartitions as features of trees T_0 and T_1: a corresponds to the shared bipartitions between trees T_0 and T_1, b represents the number of unique bipartitions in tree T_0 that are not in tree T_1, c is the number of unique bipartitions in T_1 that are

not in T_0, and d represents the number of unique bipartitions that are absent from both trees T_0 and T_1.

b. Weighted similarity and distance

The binary formulation for our a, b, c, and d values assumes that all our unique bipartitions are weighted equally. While this is a valid assumption for unweighted phylogenies, it no longer holds for weighted phylogenies, in which bipartitions in each tree can have a different weight associated with them. The formulation of a, b, c, and d values can be extended to account for real values.

Suppose we have two trees, X and Y, over some larger collection of t trees, in which there are m unique bipartitions. Each tree can be represented by an array $X = \{x_1 \dots x_m\}$ and $Y = \{y_1 \dots y_m\}$, where x_i and y_i denotes the weight of the i^{th} bipartition in trees X and Y respectively. Since each tree has at most $2n - 2$ bipartitions, $m \geq 2n-2$. Suppose bipartition j does not exist in tree X. In this case, x_j would be 0.0. Since a represents the number of bipartitions that are contained in both X and Y, it follows that $\sum_{i=1}^{m} x_i y_i$ is the weighted representation of a. Note that if either x_i or y_i is 0.0, the product for the i^{th} bipartition would be 0.0. $b + c$, which is thought of as the difference sum, is represented by the notation $\sum_{i=1}^{m} |x_i - y_i|$. It also follows that $\sum_{i=1}^{m} x_i$ is the weighted version of $a + b$ and $\sum_{i=1}^{m} y_i$ can be though of $a + c$. In certain measures, however, the weighted versions of $(a + b)$ and $(a + c)$ become $\sum_{i=1}^{m} (x_i)^2$ and $\sum_{i=1}^{m} (y_i)^2$ respectively. However, when dealing with binary presence or absence, then $\sum_{i=1}^{m} x_i = \sum_{i=}^{m} (x_i)^2$.

Weighted coefficients can thus be derived by combining these values together. Notably absent from this discussion is the handling of value d. Since d represents the quantity of bipartitions that are not contained in either X or Y, and when a bipartition is not contained in X or Y it has a value of 0.0, d in the weighted context

Table III. A sampling of similarity and distance coefficients that appear in the literature [85, 86, 87]. D or S represents whether the coefficient measures distance or similarity between two trees, respectively. Bipartition ranges are for non-trivial bipartitions.

Coeff	**S/D**	**Equation**	**Range**		**Year**
			General	*Bipartitions*	
$C1$	D	Hamming $= b + c$	$[0, m]$	$[0, 2(n-3)]$	1950
$C2$	S	Jaccard-Tanimoto $= \frac{a}{a+b+c}$	$[0, 1]$	$[0, 1]$	1901
$C3$	S	Dice $= \frac{2a}{2a+b+c}$	$[0, 1]$	$[0, 1]$	1945
$C4$	S	Russel-Rao $= \frac{a}{m}$	$[0, 1]$	$[0, 1]$	1940
$C5$	S	Sokal-Sneath $= \frac{a}{a+2b+2c}$	$[0, 1]$	$[0, 1]$	1963
$C6$	S	Hamann $= \frac{a+d-b-c}{m}$	$[-1, 1]$	$[-1, 1]$	1961
$C7$	S	Yule $= \frac{ad-bc}{ad+bc}$	$[-1, 1]$	$[-1, 1]$	1900
$C8$	S	Ochai $= \frac{a}{\sqrt{(a+b)(a+c)}}$	$[0, 1]$	$[0, 1]$	1957
$C9$	S	Anderberg $= \frac{\alpha-\beta}{2m}$, where $\alpha = \max(a, b) + \max(c, d) + \max(a, c) + \max(b, d)$ $\beta = \max(a + c, b + d) + \max(a + b, c + d)$	$[0, \frac{1}{2}]$	$[0, \frac{1}{2}]$	1973
$C10$	S	Forbes $= \frac{am}{(a+b)(a+c)}$	$[0, \infty)$	$[0, \infty)$	1907
$C11$	S	Mountford $= \frac{a}{\frac{1}{2}(ab+ac)+bc}$	$[0, 1]$	$[0, 1]$	1962
$C12$	S	Sorgenfrei $= \frac{a^2}{(a+b)(a+c)}$	$[0, 1]$	$[0, 1]$	1959
$C13$	S	Gilbert & Wells $= \log a - \log m - \log(\frac{a+b}{m}) - \log(\frac{a+c}{m})$	$(-\infty, \infty)$	$(-\infty, \infty)$	1966
$C14$	S	Tarwid $= \frac{ma-(a+b)(a+c)}{ma+(a+b)(a+c)}$	$[-1, 1]$	$[-1, 1]$	1960
$C15$	S	Johnson $= \frac{a}{a+b} + \frac{a}{a+c}$	$[0, 2]$	$[0, 2]$	1967
$C16$	S	Peirce $= \frac{ab+bc}{ab+2bc+cd}$	$[0, 1]$	$[0, 1]$	1884
$C17$	S	Driver-Kroeber $= \frac{a}{2}(\frac{1}{a+b} + \frac{1}{a+c})$	$[0, 1]$	$[0, 1]$	1932
$C18$	S	Fager-McGowan $= \frac{a}{\sqrt{(a+b)(a+c)}} - \frac{\max(a+b, a+c)}{2}$	$(-\infty, \frac{1}{2}]$	$(-\infty, \frac{1}{2}]$	1963
$C19$	D	Euclidean $= \sqrt{b+c}$	$[0, \sqrt{m}]$	$[0, \sqrt{2(n-3)}]$	300 BC
$C20$	S	Stiles $= \log \frac{m(\lvert ad-bc\rvert - \frac{m}{2})^2}{(a+b)(a+c)(b+d)(c+d)}$	$(-\infty, \infty)$	$(-\infty, \infty)$	1923
$C21$	S	Rogers-Tanimoto $= \frac{a+d}{a+2(b+c)+d}$	$[0, 1]$	$[0, 1]$	1960
$C22$	S	Eyraud $= \frac{m^2(ma-(a+b)(a+c))}{(a+b)(a+c)(b+d)(c+d)}$	$(-\infty, \infty)$	$\left[\frac{-m^2}{(m-(n-3))^2}, \frac{m^2}{(n-3)(m-(n-3))}\right]$	1936
$C23$	D	Lance-Williams $= \frac{b+c}{2a+b+c}$	$[0, 1]$	$[0, 1]$	1967
$C24$	D	Robinson-Foulds $= \frac{b+c}{2}$	$[0, \frac{m}{2}]$	$[0, n-3]$	1981

is always 0. For this reason, it is rare to find a weighted definition of for coefficients involving the quantity d. Of our sample of 24 coefficients, we were able to identify 10 weighted forms in the literature [88], and propose a derivation of an additional 5. These can be viewed in Table IV.

2. Algorithm Description

In this section, we discuss the intricacies of the Phlash algorithm. Phlash can handle both weighted and unweighted topologies. While the algorithms for both are largely the same, there are subtle differences that need to be discussed. In both cases, the bipartitions from each individual tree are extracted and inserted into a hash table, which is used to identify the set of unique bipartitions (see Chapter IV). The contents of the hash table are then transformed into an intermediary structure on which all critical operations are formed. We discuss this process in the subsections below, and discuss how the unweighted and weighted cases relate to each other.

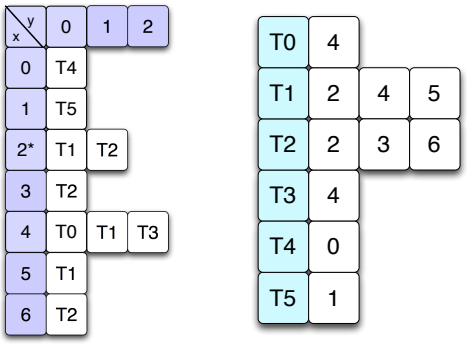
a. Step 1: Constructing the compact or shared table of interest

To construct the similarity or distance matrix of interest, we next take the contents of our hash table and convert it to an intermediary structure which we then use for all further computations. For the unweighted case, this intermediary structure is called the *compact table*. For the weighted case, we use a different structure called the *shared table*. We discuss these in turn in the paragraphs below.

The unweighted case: Constructing a compact table. After constructing the hash table for a collection of t trees, we convert it to a compact table to attain better running time and storage requirements. For every bipartition B, there is a set of trees that contain it, and a set of trees that do not. Our new compact representation

Table IV. Established and proposed weighted coefficients for our 24 distance measures. Measures that were not found from the literature [88], but derived independently are marked with the prefix P. Measures we were unable to derive or find in the literature are marked with 'n/a'.

Coeff	S/D	Name	Weighted Equation	Range
$C1$	D	Hamming	$\sum_{i=1}^{m} \lvert x_i - y_i \rvert$	$[0, \infty)$
$C2$	S	Jaccard-Tanimoto	$\frac{\sum_{i=1}^{m} x_i y_i}{\sum_{i=1}^{m}(x_i)^2+\sum_{i=1}^{m}(y_i)^2-\sum_{i=1}^{m} x_i y_i}$	$[-1/3, 1]$
$C3$	S	Dice	$\frac{2\sum_{i=1}^{m} x_i y_i}{\sum_{i=1}^{m}(x_i)^2+\sum_{i=1}^{m}(y_i)^2}$	$[0, 1]$
$C4$	S	Russel-Rao	$\frac{\sum_{i=1}^{m} x_i y_i}{m}$	$[0, \infty)$
$C5$	S	Sokal-Sneath	$\frac{\sum_{i=1}^{m} x_i y_i}{2\sum_{i=1}^{m}(x_i)^2+2\sum_{i=1}^{m}(y_i)^2-3\sum_{i=1}^{m} x_i y_i}$	$[0, 1]$
$C6$	S	Hamann	n/a	n/a
$C7$	S	Yule	n/a	n/a
$C8$	S	Ochai	$\frac{\sum_{i=1}^{m} x_i y_i}{\sqrt{\sum_{i=1}^{m}(x_i)^2 \times \sum_{i=1}^{m}(y_i)^2}}$	$[0, 1]$
$C9$	S	Anderberg	n/a	n/a
$C10$	S	Forbes	$\frac{m\sum_{i=1}^{m} x_i y_i}{\sum_{i=1}^{m}(x_i)^2 \times \sum_{i=1}^{m}(y_i)^2}$	$[0, \infty)$
$C11$	S	Mountford	n/a	n/a
$C12$	S	Sorgenfrei	$P = \frac{(\sum_{i=1}^{m} x_i y_i)^2}{\sum_{i=1}^{m}(x_i)^2 \times \sum_{i=1}^{m}(y_i)^2}$	$P = [0, \infty)$
$C13$	S	Gilbert & Wells	n/a	n/a
$C14$	S	Tarwid	$P = \frac{m\sum_{i=1}^{m} x_i y_i - \sum_{i=1}^{m}(x_i)^2 \times \sum_{i=1}^{m}(y_i)^2}{m\sum_{i=1}^{m} x_i y_i + \sum_{i=1}^{m}(x_i)^2 \times \sum_{i=1}^{m}(y_i)^2}$	$P = [-1, 1]$
$C15$	S	Johnson	$P = \frac{\sum_{i=1}^{m} x_i y_i}{\sum_{i=1}^{m}(x_i)^2} + \frac{\sum_{i=1}^{m} x_i y_i}{\sum_{i=1}^{m}(y_i)^2}$	$P = [0, 2]$
$C16$	S	Peirce	n/a	n/a
$C17$	S	Driver-Kroeber	$P = \frac{\sum_{i=1}^{m} x_i y_i}{2} \times (\frac{1}{\sum_{i=1}^{m}(x_i)^2} + \frac{1}{\sum_{i=1}^{m}(y_i)^2})$	$P = [0, 1]$
$C18$	S	Fager-McGowan	$P = \frac{\sum_{i=1}^{m} x_i y_i}{\sqrt{\sum_{i=1}^{m}(x_i)^2 \sum_{i=1}^{m}(y_i)^2}} - \frac{max(\sum_{i=1}^{m}(x_i)^2, \sum_{i=1}^{m}(y_i)^2)}{2}$	$P = (-\infty, \infty)$
$C19$	D	Euclidean	$\sqrt{\sum_{i=1}^{m} \lvert x_i - y_i \rvert^2}$	$[0, \infty)$
$C20$	D	Stiles	n/a	n/a
$C21$	S	Rogers-Tanimoto	n/a	n/a
$C22$	S	Eyraud	n/a	n/a
$C23$	D	Lance-Williams	$\frac{\sum_{i=1}^{m} \lvert x_i - y_i \rvert}{\sum_{i=1}^{m} x_i + \sum_{i=1}^{m} y_i}$	$[0, 1]$
$C24$	D	Robinson-Foulds	$\frac{\sum_{i=1}^{m} \lvert x_i - y_i \rvert}{2}$	$[0, \infty)$

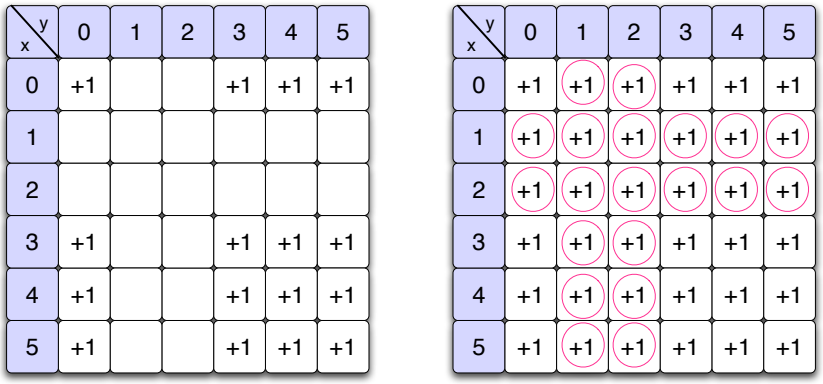


(a) compact table (b) compact inverted index

Fig. 24. Data structures used in Phlash for the phylogenetic trees shown in Figure 10. For the compact table (a), the `global` counter is 2 and `local` $[0 \ldots 5]$ is 0, 1, 1, 0, 0, 0. An $*$ denotes a compacted line. The inverted index in (b) is based directly on the compact table in (a).

stores the smaller of these two sets. Let $\mathcal{T}$ represent the set of input trees, where $|\mathcal{T}| = t$. Let $\mathcal{B}_{share}$ represents the set of trees that share bipartition B. $\mathcal{B}_{noshare} \equiv \mathcal{T} - \mathcal{B}_{share}$. If $|\mathcal{B}_{share}| < |\mathcal{B}_{noshare}|$, then bipartition B's list of TIDs is $\mathcal{B}_{share}$. Otherwise, it is $\mathcal{B}_{noshare}$. For example, the bipartition $ABC|DEF$ located in $H[12.98]$ in Figure 13 occurs in trees T_0, T_3, T_4 and T_5. Hence, there are four trees that share this bipartition and two trees ($T1$, T_2) that do not contain it. Thus, only T_1 and T_2 are associated with this bipartition in our compact representation (row 2 in Figure 24(a)).

Let $S(i, j)$ represent the number of bipartitions that are shared between trees T_i and T_j. $S(i, j)$ is a crucial quantity for computing the value of a. According to the hash table(Figure 13), bipartition $AB|CDEF$ is contained in all of the trees. Normally, all t^2 cells in the shared matrix S will be incremented by one. Instead of incrementing every $S(i, j)$ entry, Phlash instead increments a `global` counter to symbolize updating each entry $S(i, j)$ in the matrix. We note here that since $|\mathcal{B}_{noshare}|$ for the bipartition $(AB|CDEF)$ is zero, it does not need to be stored in the compact



(a) hash table (b) compact table

Fig. 25. An illustration of how to update the shared matrix S when trees T_0, T_3, T_4, and T_4 share bipartition $ABC|DEF$, which is depicted in Figure 13. (a) How the entries in the 6×6 matrix are normally updated using a hash table. (b) How the matrix is updated using a compact table.

table. A single increment to the `global` counter is sufficient to account for this bipartition.

Consider bipartition $ABC|DEF$ in Figure 13(a). Figure 25(a) shows how the shared matrix is normally updated after processing this bipartition. Row 2 of the compact table (see Figure 24(a)) stores the $\mathcal{B}_{noshare}$ set of this bipartition, and the `global` counter is incremented by one. However, cells $S(1,i), S(2,i), S(i,1)]$, and $S(i,2)$, where $0 \leq i \leq 5$, should not be updated (see Figure 25(b)), and are overvalued by the `global` counter. To handle this situation, we correct the shared matrix values by using a `local` vector, whose length is equal to t, the number of input trees. Thus, for this bipartition, `local`[1] and `local`[2] are incremented by one.

The weighted case: Constructing a shared table. If the set of input trees contain branch lengths, Phlash will calculate the weighted distance between them. For the calculation of weighted coefficients, three main quantities come into play: $(a + b)$,

$(a + c)$, and a. Since $(a + b)$ and $(a + c)$ equal the quantities $\sum_i^m x_i$ and $\sum_i^m y_i$ respectively, it suffices to simply sum together all the branches in each tree and store it in a array. In cases where the quantities $\sum_i^m (x_i)^2$ and $\sum_i^m (y_i)^2$ are needed, we store the sum of the squares of all the branches instead. What remains is to compute the value of a, which is $\sum_i^m x_i y_i$ in our similarity coefficients and $\sum_i^m |x_i - y_i|$ in our distance coefficients. For calculating a for our similarity coefficients, we used a shared table data structure in conjunction with a data structure called the shared inverted index. A shared table is simply a linearized version of our hash table, where each bipartition occupies single row. For each bipartition, we store the id of the tree that contains it, and its associated weight. Note that a compact table does not make sense here, since branch lengths are over an infinite domain, unlike our tree ids. The shared inverted index contains t rows, one for each tree. For each row, we note the bipartitions that the tree contains, along with the associated weights. Figure 26 illustrates the corresponding shared table (Figure 26(a)) and the shared inverted index (Figure 26(b)) for the weighted trees presented in Figure 10.

b. Step 2: Computing the a, b, c, and d values between trees

Once the appropriate intermediary data structures are populated, we perform operations on them directly to compute our a, b, c, and d values of interest. For the unweighted case, these operations are performed on the compact table and inverted index. For the weighted case, the shared table and shared inverted index are used for the computation. Details for the unweighted and weighted computation follow below.

Using a compact table to compute the a, b, c, and d values for two trees. Consider two trees T_i and T_j. Once the compact table is constructed and the `global` and

x \ y	0	1	2	3	4	5
0	T4 0.29					
1	T2 0.11					
2	T2 0.43					
3	T0 0.14	T1 0.12	T2 0.63	T3 0.23	T4 0.26	T5 0.27
4	T5 0.19					
5	T0 0.63	T1 0.82	T2 0.72	T3 0.69	T4 0.65	T5 0.14
6	T0 0.34	T3 0.34	T4 0.92	T5 0.54		
7	T0 0.81	T1 0.62	T2 0.22	T3 0.56	T4 0.30	T5 0.22
8	T0 0.40	T1 0.92	T2 0.33	T3 0.12	T4 0.11	T5 0.27
9	T1 0.31					
10	T0 0.12	T1 0.32	T2 0.24	T3 0.34	T4 0.43	T5 0.36
11	T0 0.91	T1 0.04	T2 0.01	T3 0.45	T4 0.03	T5 0.82
12	T0 0.32	T1 0.42	T2 0.31	T3 0.35	T4 0.14	T5 0.02
13	T0 0.23	T1 0.33	T3 0.57			

(a) shared table

T0	3	5	6	7	8	10	11	12	13
	0.14	0.63	0.34	0.81	0.40	0.12	0.91	0.32	0.23
T1	3	5	7	8	9	10	11	12	13
	0.12	0.82	0.62	0.92	0.31	0.32	0.04	0.42	0.33
T2	1	2	3	5	7	8	10	11	12
	0.11	0.43	0.63	0.72	0.22	0.33	0.24	0.01	0.31
T3	3	5	6	7	8	10	11	12	13
	0.23	0.68	0.92	0.56	0.12	0.34	0.45	0.35	0.57
T4	0	3	5	6	7	8	10	11	12
	0.29	0.26	0.65	0.92	0.30	0.11	0.43	0.03	0.14
T5	3	4	5	6	7	8	10	11	12
	0.27	0.19	0.14	0.54	0.22	0.27	0.36	0.82	0.02

(b) shared inverted index

Fig. 26. Data structures used in Phlash for the phylogenetic trees shown in Figure 10(b). Note that both the shared table (a) and the shared inverted index (b) have branch lengths associated with each tree id.

`local` vectors are updated, the value of a for T_i and T_j is:

$$a \;=\; S(i,j) + \texttt{global} \;- \texttt{local}[i] - \texttt{local}[j]. \tag{5.1}$$

Given the value of a, it is easy to get the values of the other variables. The maximum number of bipartitions in a binary tree is $n-3$. Hence, for binary trees, b and c are equal to $(n-3)-a$. However, if during step 1, if we detected that our trees are multifurcating, then we keep a separate vector that stores the number of bipartitions that a tree contains. Let u_i and u_j, where $0 < u_i, u_j \leq (n-3)$, represent the number of bipartitions in trees T_i and T_j, respectively. Then, $b = u_i - a$ and $c = u_j - a$. Once we have the values for a, b, c, and m, we can easily get the value for d, which is $m-(a+b+c)$. With these values, we can compute the distance between trees T_i and T_j using any of the measures listed or more generally, any measure that is a function of a, b, c, d and m.

Using a shared table to compute the a, b, and c values for two trees. Our shared table alone is sufficient to calculate the value of $\sum_{i=1}^{m} x_i y_i$. To compute $a = \sum_{i}^{m} x_i y_i$ for our similarity measures for a particular row in our matrix, we use the shared inverted index to to guide us to which rows in the shared table we should visit, and the associated branch length values that should be used in the calculation. For example, to calculate the first row of our matrix, we will visit shared table locations 3, 5, 6, 7, 8, 10, 11, 12 and 13 (Figure 26(b)) according to our shared inverted index, as those are the row locations that contain T_0 in our shared table. In the first access to our shared table, we visit row 3 in our shared table with value 0.14. The first row of the matrix then gets the values of [(0.14×0.14), (0.14×0.12), (0.14×0.63), (0.14×0.23), (0.14×0.26), (0.14×0.27)], or [0.0196, 0.0168, 0.0882, 0.0322, 0.0364, 0.0378]. The next access to our shared table, we visit row 5 with value 0.63. We

add this new series of multiplications to the vector of values that already exist. For example, we add (0.63×0.63) to 0.0196, the the current value residing in cell (0,0) to yield a updated value of 0.4165. In this manner, all the corresponding cells in our row of interest is updated, until we have all the weighted a values for the row.

For distance measures, we must account for both the presence and absence of any bipartition for any pair of trees. However, since a shared table does not tell our algorithm what bipartitions *are not* shared between two trees, we use the fully inverted index to calculate our values. The fully inverted index (not shown) is a $t \times m$ structure, where each row pertains to a tree, and each column to a unique bipartition in our collection. If a tree contains a bipartition, its weight is used. If the tree does not contain a bipartition, the corresponding value is 0.0. For any pair of trees X and Y, we visit rows X and Y in our inverted index and calculate $a = \sum_i^m |x_i - y_i|$ accordingly. Thus, to compute trees 1 and 2, we iterate over all the bipartitions in rows 1 and 2 of our inverted index. Once these a quantities are established, it is easy to compute any of our defined weighted coefficients.

A note about weighted measures for comparing trees in the literature. To the best of our knowledge, Phlash is currently the only program in the literature that can compute the weighted RF distance. We examined a number of phylogenetic software programs, such as WHashRF (unreleased), RAxML, PhyloNet, and Felsenstein's `treedist` program. While RAxML and PhyloNet claim to calculate the weighted RF distance, the value actually being calculated is the RF rate (or the normalized RF distance), which does not require branch lengths to be computed correctly. The `treedist` program does not implement weighted RF distance. However, it does implement a version of Euclidean distance that is implemented in Phlash as well ($C19$).

Lastly, WHashRF is an unpublished program for calculating weighted RF dis-

tances. While most of the code is correct, we found a few serious errors in the implementation that cause the weighted RF distance between two trees to be computed incorrectly. In addition to RF distance, we have also implemented the weighted coefficients for the 15 coefficients that we were able to derive from available literature and independently.

c. Step 3: Computing a $t \times t$ matrix of interest

Even with a reduction of the number of comparisons, memory requirements can prove to be prohibitive. Since it is impossible to determine which trees will be associated with what bipartition in a collection of trees, a $t \times t$ matrix is necessary allocated in memory to allow for the near-random increment of cells. However, the allocation of the $t \times t$ matrix quickly becomes a bottleneck for very large collections. Consider our `insects` collection of $150,000$ trees over 525 taxa. The corresponding matrix requires 60 GB of space, which is not possible to allocate in main memory on commercially available desktop machines.

To reduce memory requirements, Phlash updates the matrix S (and the corresponding similarity or distance matrix X) one row at a time. We use an inverted index to help us identify where each tree id is located in the compact table. Row i of the inverted index corresponds to tree T_i and the contents at this row refer to the locations in the compact table where T_i appears. For example, the inverted index in Figure 24(b) shows that tree T_1 is located at locations 2, 4 and 5 in the compact table.

Unweighted computation. To compute the $t \times t$ similarity or distance matrix of interest, we start with computing row 0 which corresponds to tree T_0. For each j in row 0 of the inverted index, we increment $S[0, j]$ by 1. Afterwards, we compute

the values of a, b, c, and d as described above to calculate $X[0, l]$, where $0 \leq l < t$. We continue the above process to compute each row i of the similarity or distance matrix, X.

Weighted computation. For the weighted computation, we compute the row of a values using the appropriate procedure described above, depending on whether or not a similarity or distance coefficient is being calculated. Once our array of a values is computed, we use the $(a+b)$ and $(a+c)$ as described above to calculate $X[0, l]$, where $0 \leq l < t$. As before, this process is repeated until each row of the similarity matrix is computed.

d. A note about comparing heterogeneous trees

Our hashing strategy in combination with the use of partial bitstrings allows Phlash to compare heterogeneous trees with no extra overhead. Our a, b, c and d values are calculated as per normal. Consider T_0 and T_1 from Figure 15. Both share relationship `11` (AB) and `111` (ABC). T_0 has the unique relationships `000111` (DEF) and `000011`(EF). T_1 has the unique relationship 00011 (DE). Therefore, $a = 2$, $b = 2$ $(4 - 2)$, and $c = 1$ $(3 - 2)$. The RF distance, represented by $\frac{b+c}{2}$, will be 1.5 for these two trees.

3. Algorithm Analysis

Like MrsRF, $O(nt)$ time and storage is needed for computing our hash table. The power of Phlash lies, however, in the reduction of comparisons and the memory footprint for the $t \times t$ matrix. We discuss each of these in turn. For unweighted trees, the performance of Phlash improves with the level of bipartition sharing, due to the use of the compact table. This guarantees that each line in our compact table has length

$O(\frac{t}{2})$, where t is the number of trees in our collection. Given that a tree has at most $O(n)$ bipartitions, in the worst case, each tree has each of its bipartitions on a line that is shared by $\frac{t}{2}$ trees. Thus, to compute one row of the matrix, $O(\frac{nt}{2})$ comparisons will be needed in the worst case. In the best case, all the trees are identical. In this case, no comparisons are necessary, as all the relevant values for computing a will be stored in our `global` counter.

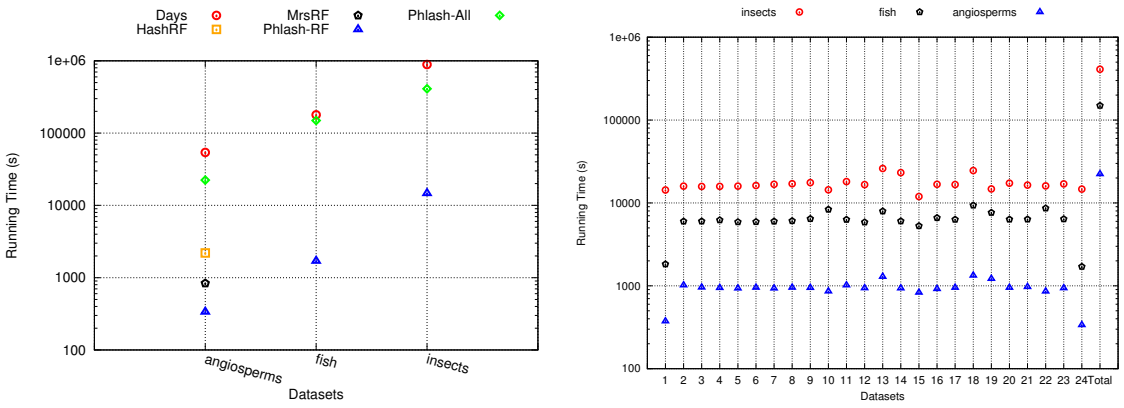
In contrast, our weighted similarity coefficients require a shared table, resulting in $O(nt)$ comparisons to compute a row of the matrix. For our weighted distance approaches, $O(mt)$ comparisons are needed on the inverted index, where m is the number of unique bipartitions in the collection. Since $m \geq n$, this explains why the computation of our weighted distance coefficients takes much longer than those of our weighted similarity coefficients. In all cases however, we can compute arbitrarily large comparison matrices. This is due to the reduction of the memory footprint. In MrsRF and previous approaches, a $t \times t$ matrix or sub-matrix must necessarily be allocated in memory. In Phlash, we only allocate one row of the matrix. This reduces the memory footprint from $O(t^2)$ to $O(t)$, where t is the number of trees.

4. Experimental Analysis

a. Summary of Phlash performance

We tested the performed of Phlash over three biological datasets: `angiosperms`, `fish` and `insects`. To measure performance, Phlash was run on each dataset and coefficient. The run-times shown represent the average of three runs.

Unweighted performance. Phlash is very efficient on unweighted phylogenies. Figure 27 shows the performance of Phlash on the RF coefficient (Figure 27(a)) and on our 24 implemented coefficients (Figure 27(b)). On our smallest dataset of `angiosperms`,



(a) RF Comparison (b) Phlash Coefficient Comparison

Fig. 27. Analysis of Phlash performance. (a) shows the running time of Phlash-RF compared to other RF algorithms. (b) shows the running time of Phlash on all our coefficient methods. Results for each dataset are shown.

Phlash-RF is roughly 2.5 times faster than MrsRF, 6.5 times faster than HashRF, and 160 times faster than Day's algorithm. Due to memory restrictions, MrsRF and HashRF were unable to calculate the RF matrices for the `fish` and `insects` datasets serially on our platform. Since Day's algorithm computes the RF distance between every pair of trees, we were able to use it as a basis of comparison on these larger datasets.

On the `fish` dataset, Phlash-RF is approximately 105 times faster than Day's algorithm. On our largest dataset (`insects`), Day's took over 10 days to compute the 60 GB RF matrix required for this dataset. Phlash-RF is 60 times faster by comparison, taking approximately 4 hours to compute and print out the matrix. We note that for each of our datasets, Phlash computes all the coefficients in significantly less time than it takes Day's algorithm to finish the computation of the RF matrix for the dataset of interest.

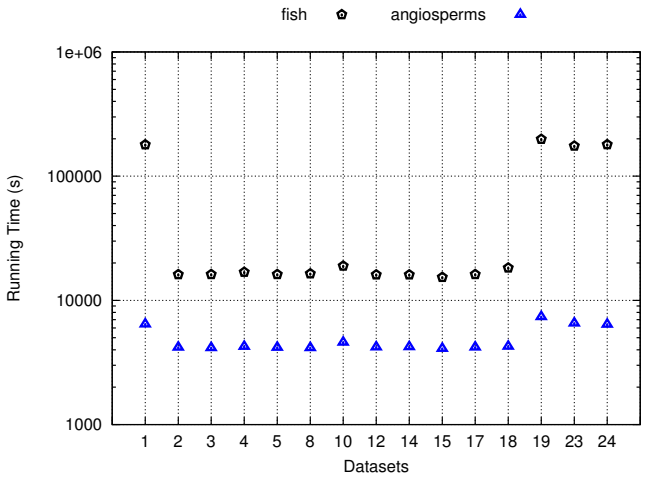


Fig. 28. Weighted run-time performance of our different coefficients on our weighted datasets of interest. Due to their complexity, distance measures perform worse than similarity measures.

Weighted performance. Due to additional complexity of branch lengths, the weighted coefficient calculation in Phlash is significantly slower than than the unweighted coefficient calculation. The fact that we can no longer take advantage of a compact table for our weighted similarity measures causes results in a slow-down of about 15. Figure 28 shows the performance of our weighted coefficients. Phlash computes our weighted distance coefficients, which requires only the inverted index for calculation, up to 100 times slower than the unweighted distance calculation. The weighted distance measures are also about 12 times slower to calculate than our weighted similarity measures. In contrast, the unweighted similarity measures were up to 3 times slower than the unweighted distance measures (namely RF), due to the extra overhead involved in these computations over something simple like RF (which uses simply the b or c value for binary trees). In the future, we plan on optimizing the performance of our weighted coefficients.

5. Identifying rogue taxa

Now that we have a number of ways to compare trees, our goal was to see if we could use any of them to address the rogue taxa problem. Recall the two trees shown in Figure 11. If taxon E were removed, both of these trees have the same unrooted topology. However, given the placement of taxon E in both trees, they have the maximum RF distance of 3. To test how robust the 24 coefficients are to the rogue taxa problem, we compared the two trees using each of our 24 coefficients. We found that all our distance coefficients yield the maximum distance for the respective ranges for the two trees. For the similarity coefficients, all but 4 yielded the minimum similarity value: Anderberg (returned 0.5, min is 0), Peirce (returned 0.5, min is 0), Fager-McGowan (returned -1.5, min is $-\infty$) and Stiles (returned 0.426, min is $-\infty$). This suggests that these four coefficients may be useful as alternatives for the rogue taxa problem. While the test was on a small topology, the results will hold for larger topologies where the RF distance is maximally distant, as the values of b, c, d, and m will increase in constant proportion to the increase in n, the number of taxa. a will continue to remain 0, so the relationships will not change.

6. Clustering of coefficients

Next, we wished to ascertain how our 24 coefficients compared to each other. By comparing the trees with the 24 similarity and distance coefficients, we will be able to analyze different perspectives of tree relationships across phylogenetic analyses. We use Ward's hierarchical clustering algorithm [89] to cluster the runs. Such a clustering produces a rooted dendrogram, and such dendrograms are non-trivial when clustering three or more objects. For us, this means that we need to consider datasets consisting of three or more runs. Hence, our focus on the `angiosperms` and `insects`

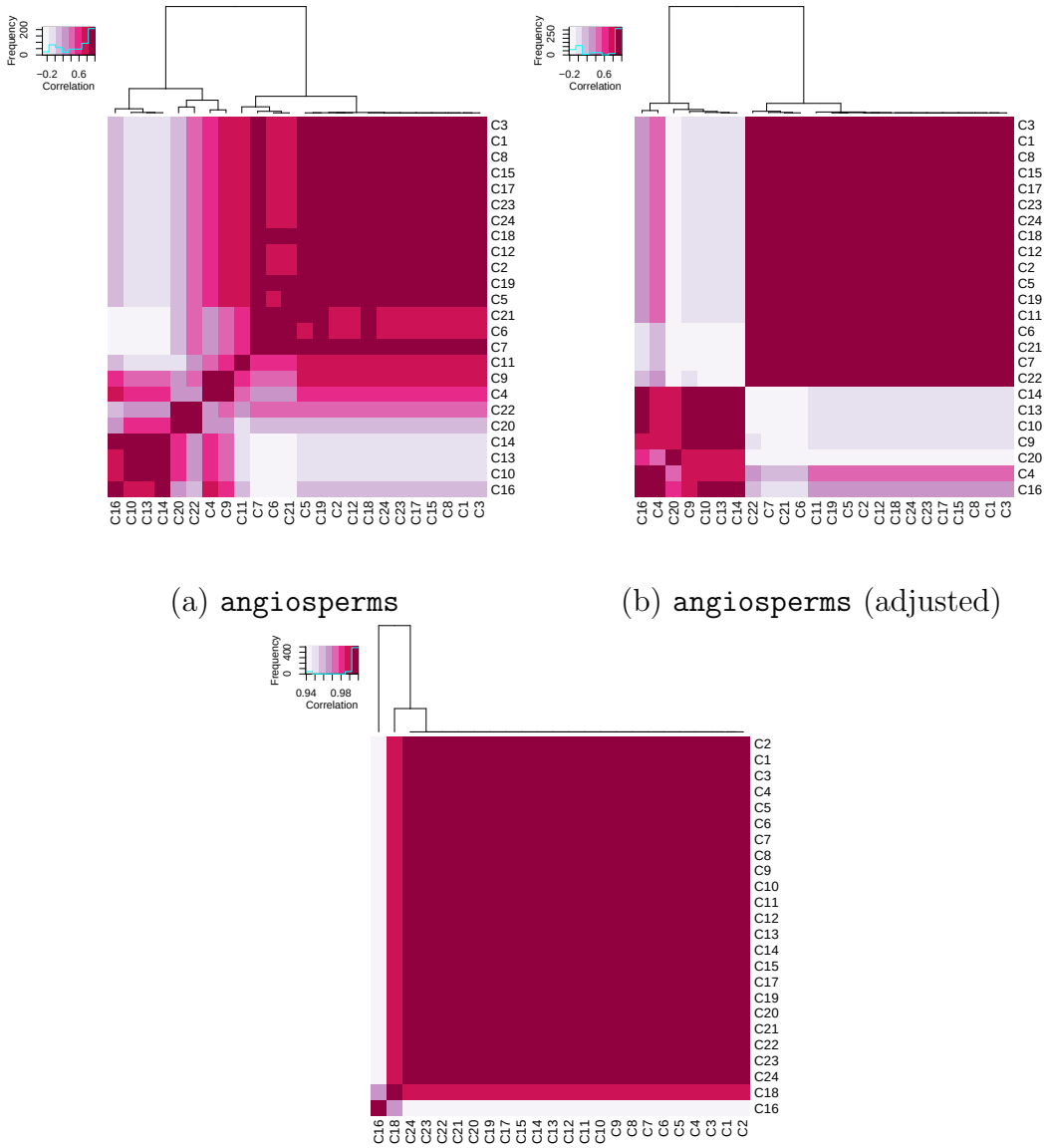
tree collections.

Next, we used the Mantel statistic [90] to compute the correlation between the matrices. The Mantel test examines the relationship between two square matrices (often distance matrices) X and Y. The values within each matrix ($X(i,j)$ or $Y(i,j)$) represent a relationship between points i and j. The basic Mantel statistic is simply the sum of the products of the corresponding elements of the matrices $Z = \sum_{i,j} X(i,j)Y(i,j)$, where $i \neq j$. Because Z can take on any value depending on the exact nature of X and Y, one usually uses a normalized Mantel coefficient, calculated as the correlation between the pair wise elements of X and Y. Like any product-moment coefficient, it ranges from -1 to 1.

The Mantel statistic as implemented in R requires all of the matrices to be dissimilarity matrices. Since we have a combination of similarity and distance matrices, we converted all of the matrices to Euclidean distance matrices, which is the default option, using the `dist` function in R. Figure 29 shows the result.

a. `angiosperms` dataset

Figure 29(a) shows the relationships between the 24 different coefficients on the `angiosperms` dataset. Clearly, the coefficients cluster in a distinct fashion. We observe two main clusters, with two large sub-clusters underneath. This suggests that our different coefficients tell different stories about how the trees are related, with a cluster representing a consensus on the types of stories being told. Of note is that Anderberg (C9), Peirce (C16) and Stiles (C20) appear in a distinct cluster from all the other coefficients. These are three of the four coefficients that were immune to the rogue taxa example that we showed in Figure 11. Figure 29(b) shows what happens when we adjust the data to include trivial bipartitions. Since all trees have the same n taxa in common, this inflates the similarity between all the trees by n. The result



(a) `angiosperms` (b) `angiosperms` (adjusted)

(c) `insects`

Fig. 29. Heatmaps depicting the relationship of the 24 similarity and distance coefficients. In (a) and (b) we show the regular and adjusted correlation matrices for the `angiosperms` dataset. In (c) we show the correlation matrix for the `insects` dataset.

is that the heatmap in Figure 29(b) looks more uniform than the one in Figure 29(a). However, it is still significant to note that all three coefficients still cluster separately from the large, main block of coefficients, of which RF distance (C24) and RF Rate (C23) belong to. This could be a possible reason why these three coefficients treat rogue taxa differently than RF.

b. `insects` dataset

The `insects` dataset is composed of five runs using the maximum parsimony optimization criterion. Each run contains $30,000$ trees. Figure 29(c) shows the same information regarding the correlation of the coefficients to each other. This dataset only has 573 unique bipartitions. Given that a binary tree for this dataset over 525 taxa would have 522 unique bipartitions, the fact that there are only 573 unique bipartitions across the $150,000$ trees implies high similarity across the tree collection. As a result, 22 of the 24 measures agreed on their clustering of the data. However, it is still interesting to note that Peirce and Fager-McGowan clustered together and separately from the large main cluster. Once again, these are two coefficients that were robust to our rogue taxa example.

What could be causing different coefficients to cluster together? We believe the clustering patterns that we have observed have to do with the combination of values used in the coefficient equations. For example, all the equations that are formed only with a combination of the variables a, b and c cluster together. Those that use the variable m in the numerator and denominator form other separate clusters. Finally, coefficients calculated with a combination of a, b, c and d with d in the numerator cluster together. As a result, we strongly believe that m and d can have a large impact on how similar trees appear and offer insights on treating the rogue taxa problem, as these coefficients incorporate broader aspects of the collection (i.e, the number of

unique bipartitions in the collection, or the number of bipartitions in the collection *not* shared between two trees).

D. Summary

To efficiently compare the topology between large groups of trees, fast algorithms are needed. However, comparing large groups of trees is both difficult and slow, with acting as the most common bottleneck and restriction for computing large topological comparison matrices. To allow for the real-time comparison of trees, we created MrsRF, the fastest parallel algorithm for computing the all-to-all RF matrix between trees. To get around the memory bottleneck, MrsRF divides the global $t \times t$ matrix into a series of sub-matrices which are then computed independently via MapReduce.

Furthermore our use of MapReduce is *novel*, as the output size (a $t \times t$ matrix) is significantly larger than the input of t trees. On 32 cores, MrsRF can compare $20,000$ trees in roughly 8 seconds, and $33,306$ trees in roughly 36 seconds. When run on one core, it is up to 2.8 times faster than HashRF, the previous fastest serial algorithm.

We then sought to look beyond Robinson-Foulds distance as a method for comparing trees. *What other methods are there to compare trees?* By representing trees as bitstrings, our algorithm Phlash is capable of comparing trees using 24 novel coefficients. Our results for clustering show that these coefficients tell different stories on how our trees are related to each other, thus motivating future study on the usefulness of these methods for comparing trees. Furthermore, Phlash employs two novel data structures that reduce the number of comparisons needed to compute a $t \times t$ matrix, and reduces the memory footprint from quadratic to linear space. This makes Phlash the fastest serial algorithm for comparing large groups of phylogenetic trees. Furthermore, it is up to 160 times faster than traditional methods like Day's

algorithm.

For biologists, these algorithms represent the fastest and most powerful methods for studying large collections of trees. Phlash is capable of comparing weighted, unweighted, rooted, unrooted, binary and multifurcating trees, making it the most comprehensive package for exploring tree collections today. Since it implements numerous methods for comparing trees, biologists can now look beyond Robinson-Foulds in their quest of understanding the information content of their large collections of trees. Furthermore, Phlash is capable of comparing heterogeneous trees, opening up avenues of tree comparison that were not previously available. The ability to leverage MapReduce to create topological distances matrices in parallel through MrsRF shows the potential of using such techniques in the context of other applications, such as phylogenetic search.

For computer scientists, the data structures and algorithms presented in this work can be utilized for comparing other types of tree data, or feature-rich datasets. Given that we are representing trees as bitstrings, it is feasible that other datasets which utilize bitstrings to denote the presence or absence of features can benefit from the high performance algorithms presented here, allowing for the fast construction of large, dense distance and similarity matrices.

CHAPTER VI

EFFICIENT COMPRESSION OF PHYLOGENETIC TREES

In order to study large collections of trees, efficient methods are needed to store tree collections compactly on disk. Furthermore, successful version control systems must effectively compress the information that is contained in its database of files (or repository), Hence, compression also plays a critical role in its success. *How do we compress large collections of trees effectively?* Our solution is TreeZip [15, 16], a novel algorithm that compactly represents large tree collections. TreeZip is capable of handling binary or multifurcating phylogenies, whether weighted or unweighted. It is currently the most efficient method for compressing and storing large collections of phylogenetic trees.

A. Motivation

Given that many of the evolutionary relationships in a collection of phylogenetic trees are shared, the novelty of the TreeZip approach is storing such relationships only once in the compressed representation. TreeZip compresses a Newick file based on the *semantic* representation (i.e., tree bipartitions and unique topologies). The Newick format [69] is the most widely used file format to represent phylogenetic trees. In this format, matching pairs of parentheses symbolize internal nodes in the evolutionary tree.

However, the Newick representation of a tree is not unique. For a particular tree with n taxa, there are $O(2^{n-1})$ different (but equivalent) Newick strings to represent its topology. For example, there are 32 different, but equivalent Newick string representations for the phylogenetic tree shown in Figure 1. General-purpose data compression techniques (e.g., `gzip`, `bzip,` and `7zip`) cannot recognize (or leverage)

domain-specific information regarding the Newick file. To continue our example, a general purpose compressor identifies a file containing the 32 different, but equivalent Newick representations as 32 distinct objects.

In contrast, semantic (or domain-based) compression approaches leverages knowledge of the underlying data to achieve high compression quality. Popular examples include the PNG format for images, H264 for video, and MP3 for audio files. Thus, there is great potential for obtaining good compression by utilizing the semantic information in a Newick file describing large collections of evolutionary trees.

B. TreeZip Algorithm

TreeZip is an efficient compression algorithm that is robust to the multiple Newick string representations a tree can have. The TreeZip algorithm is composed of two main parts: compression and decompression. We discuss each of these in turn in the subsections that follow. Section 1 discusses the process of compression, in which an inputted Newick file is transformed into the TreeZip compressed format, or TRZ file. Section 2 discusses decompression, in which a TRZ file is used to reconstruct the desired set of phylogenetic trees in Newick format. We note here that since any phylogenetic tree with n taxa has $O(2^{n-1})$ equivalent Newick string representations, any one of these equivalent Newick string representations can be used as the decompressed version.

Section 3 describes the algorithm behind the TreeZip set operations. Unlike the compression and decompression functions, the TreeZip set operations take as input two TRZ files, and outputs a single TRZ file. In this manner, set operations are performed in the context of a TRZ file, without any loss of space savings. The latest implementation of TreeZip can be downloaded from `http://treezip.googlecode.com`.

As of March 2012, the software has been downloaded 74 times [91].

1. Compression

In the Newick input file, each string i, which represents tree T_i, is read and stored in a tree data structure. Bipartitions are extracted via depth-first search traversal and inserted into the hash table data structure using the h_1 and h_2 functions described in Chapter IV. Taxa are ordered lexicographically, where b_0 represents the first bit and the first taxon name in the ordering, b_1 is the second bit representing the second taxon in the ordering, etc. The resulting hash table is then encoded to form the TRZ file.

The first three lines of the compressed TRZ file represent the taxa names, the number of trees in the file, and the number of unique bipartitions. Afterwards, we process each hash table row which will represent a line in the compressed file. There are three components (bitstrings, tree ids, branch lengths) to a TRZ line. We also note here that bipartitions stored in the TRZ file are stored in sorted order according to the number of ones they contain. Ties are broken lexicographically. This guarantees that if two tree collections have equivalent one-to-one corresponding sets of trees, the TRZ files of the two collections will be *identical* despite differences in the Newick string representations. Below, we describe how TreeZip encodes each of these components.

a. Encoding bipartition bitstrings

Once all of the bipartitions are organized in the hash table, we begin the process of writing the TRZ compressed file, which is a plain text file. We run-length encode our bitstrings. Run-length encoding is a form of data compression in which runs of data (i.e, sequences in which the same data value occurs in many consecutive data elements) are stored as a single data value and count, rather than as the original run.

For the bipartition $AB|CDEF$ (bitstring: 110000) in Figure 13, we would have a run-length encoding of `1:2␣0:4`, where each $x : y$ element represents the data value (x) and the number of repetitions (y). The ␣ character denotes a space. Since bitstrings can either contain runs of 1s or 0s, we introduce two new symbols. `1:` is encoded as `K`, while `0:` encoded as `L` (we use characters A through J for compressing our list of tree ids described shortly). To maximize compression and to ensure support of heterogeneous collections of trees, we use an encoded partial bitstring representation, where the bitstring is only as long as its left-most 1. Hence, we encode the bitstring 110000 as `K2`. In our experiments, we considered taking every group of 7 bits in our bitstring and translating it to an ASCII character. However, we were able to get better compression by using run-length encoding, which showed significant benefits on our biological tree collections consisting of hundreds of taxa.

b. Identifying and encoding the set of unique tree ids

Let $\mathcal{T}$ represent the set of evolutionary trees of interest, where $|\mathcal{T}| = t$. For a bipartition B, $\mathcal{B}_{share}$ represents the set of the trees in $\mathcal{T}$ that share that bipartition. $\mathcal{B}_{noshare}$ is the set of trees that do not share bipartition B. Since these sets are complements, their union comprises the set $\mathcal{T}$. To minimize the amount of information present in our TRZ output, we print out the contents of the smaller of these two sets. If $|\mathcal{B}_{share}| \leq |\mathcal{B}_{noshare}|$, then we output $\mathcal{B}_{share}$. Otherwise, $\mathcal{B}_{noshare}$ is outputted. In our TRZ file, we denote $\mathcal{B}_{share}$ and $\mathcal{B}_{noshare}$ lines with the `'+'`and `'-'` symbol, respectively. This is essentially an encoded representation of the compact table discussed in Chapter V.

Even with use of the smaller of the $\mathcal{B}_{share}$ or $\mathcal{B}_{noshare}$ sets, the list of tree ids can get very large. This is due to the fact that as t grows large, the number of bytes necessary to store a single id also grows. We note first that a tree T can be represented as a k-bit bitstring, where k is the number of bipartitions discovered

in the collection. Feeding these k-bit bitstring representations into the h_3 and h_4 hash functions described in Chapter IV yields the set of unique trees, U, where $|U| = u$. This set of unique trees is given the corresponding tree ids of $0 \ldots u-1$, and will represent the total set of trees in consideration with any bipartition. Duplicate information is encoded and stored at the end of the TRZ file.

Since the trees are inserted into the hash table in their order of appearance in the Newick file, our lists of tree ids will be in increasing order. As a result, we store the differences between adjacent elements in our tree id list. These differences are then run-length encoded. To eliminate the need for spaces between the run-length encoded differences, the first digit of every element is encoded as a character, with $0 \ldots 9$ represented by $\mathrm{A} \ldots \mathrm{J}$. Consider bipartition $ABE|CDF$ (bitstring 110010), which is in row 5 (its h_1 value) in our hash table shown in Figure 13 and has an h_2 value of 90. Tree T_2 is the only tree containing this bipartition. Thus, the $\mathcal{B}_{share}$ set will be used for this bipartition, and its run-length encoded differences will be `2`, which will be encoded as `C`. Given the large number of shared bipartitions in a collection of trees that result from a phylogenetic search, there will be many more unique trees than unique bipartitions. Hence, encoding the differences in the tree ids leverages the sharing among the trees—especially since 1 is the most common difference between adjacent elements in the tree id lists.

c. Encoding branch lengths

The last item to process on a hash line are the branch lengths associated with a unique bipartition. Branch lengths take the form $x.y$, where x is the integral and y the mantissa. For this domain, branch lengths tend to be very small ($x = 0$). Hence, we use this property to our advantage by only encoding the integral in special cases ($x > 0$). For these special cases, we store the integral separately along with its

related tree id. On the datasets studied here, at least 99.6% of the branch lengths begin with 0. The mantissa corresponds to a fixed number, d, of digits. For our tree collections, $d = 6$.

To encode the mantissa, we take two digits at a time (since we can guarantee this value fits into a byte) and translate it into a readable ASCII character. For example if we have a value of 99 as input, we add 33 to create the corresponding Extended ASCII readable character ä. It is necessary to add 33 to any inputted value since the first 32 characters in ASCII are non-printable, control characters. We tried different universal integer encodings (e.g., variable byte encoding) [92], but given the range of integers represented by d digits, the various integer encodings did not result in a smaller compressed file. This is due to the fact that when $d = 6$, universal codes become less effective than straight binary as the size of the integers themselves increase [92]. Furthermore, we achieve better compression by feeding the resulting TRZ file to a general-purpose compressor such as `7zip`. However, when using variable byte encoding for the branch lengths, `7zip`'s algorithm could not reduce the file size any further and resulted in a much larger compressed file.

Lastly, we note that branch lengths are not compacted like tree ids since the branch lengths originate from an infinite set of real numbers. Tree ids, on the other hand, are drawn from a finite set of t tree ids ranging from 0 to $t - 1$.

d. A note on compressing heterogeneous trees

For heterogeneous trees, every bipartition from the tree is collected via depth-first search and hashed according to the procedure described in Chapter IV. Trivial bipartitions are also hashed, even in the case of unweighted trees. This is crucial for the proper storage and retrieval of heterogeneous trees, since every tree can have a unique set of taxa. Aside from this change, the rest of TreeZip compression works as

described above.

2. Decompression

The two major steps of the decompression in TreeZip are decoding the contents in the TRZ file and rebuilding the collection of t trees. Decoding reconstructs the original hash table information which consists of bipartition bitstrings and the tree ids that contain them. When the TRZ file is decoded, each line of the file is processed sequentially. First, the taxa information is fed into TreeZip. Next, the number of trees is read. Each bipartition is then read sequentially.

a. Decompression data structures

To assist in bipartition collection, we maintain two data structures M and N, both which are $t \times u$ matrices, where $u = 2n$ is the maximum number of bipartitions for a phylogenetic tree. The length of each matrix corresponds to the number of trees specified in the TRZ file. Each row i in matrix M corresponds to the bipartitions required to rebuild tree T_i. The corresponding row in matrix N is the list of associated branch lengths.

For example, in Figure 13(b), the bipartition $AB|CDEF$ at location $H[30.74]$ is shared among all the trees. It is therefore added to every row in M. To N, we add the value 0.32 to $N[0]$, 0.42 to $N[1]$, 0.31 to $N[2]$, 0.35 to $N[3]$, 0.14 to $N[4]$ and 0.02 to $N[5]$, signifying that these are the associated branch lengths for the corresponding bipartition in M. On the other hand, the bipartitions $ABE|CDF$ and $ABCE|DF$ are contained only in trees T_2 and T_5 respectively, and therefore will be added to $M[2]$ and $M[5]$. Thus, we also add branch lengths 0.43 and 0.19 to $N[2]$ and $N[5]$. Since each bipartition is processed in order in our TRZ file, we are able to guarantee a one to one correspondence between the values in M and N. We also maintain a separate

data structure that stores duplicate tree information to assist in the construction of M and N.

b. Flexible decompression

The decoded bitstrings are the basic units for building trees. Once the bitstrings, associated tree ids and branch lengths are decoded, we can build the original trees one by one. In order to build tree T_i, the tree building function receives as input matrix row $M[i]$ which contains the bipartitions encoded as bitstrings for tree T_i, and matrix row $N[i]$ which contain the associated branch lengths for each bipartition in $M[i]$. Each of the t trees is built starting from tree T_0 and ending with tree T_{t-1}, whose bipartitions (branch lengths) are stored in $M[0]$ ($N[0]$) and $M[t-1]$ ($N[t-1]$), respectively. The trees are reconstructed in the same order that they were in the original Newick file. However, given $O(2^{n-1})$ possible Newick strings for a tree T_i, the Newick representation that TreeZip outputs for tree T_i will probably differ from the Newick string in the original file. This is not a problem semantically since the different strings represent the same tree.

To build tree T_i, it is initially represented as a star tree on n taxa. A star tree is a bitstring representation consisting of all 1's. In the TRZ file, bipartitions are stored in decreasing order of their bitstrings. This means the when it is time to rebuild trees, the bipartitions that group together the most taxa appear first. The bipartition that groups together the fewest taxa appears last in the sorted list of '1' bit counts. For each bipartition i, a new internal node in tree T_i is created using the bitstring in $M[i]$, and the associated weight is added using the value in $N[i]$. Hence, the taxa indicated by the '1' bits become children of the new internal node. The above process repeats until all bipartitions are added to tree T_i.

The decompresser can also output sub-collections of trees of interest to the user.

For example, if the user was interested in the set of unique trees in the collection (rather than the entire collection), TreeZip can return this set of trees of interest to the user. In addition, TreeZip has built-in functionality to return the strict and majority-rule consensus trees of an encoded collection of trees in a couple of seconds to the user. The strict and majority-rule consensus trees are especially of interest to biologists, since this is the summary tree structure that commonly appears in publications.

Furthermore, these subcollections of trees can be produced directly from the TRZ file, without a need to decompress the original collection. In other words, operations can be performed directly on the TRZ file without requiring a loss of space savings. This is not the case with standard compression approaches which produce unreadable binary output. In these cases, the original file must always be fully decompressed in order for any operations to be performed, resulting in zero space savings.

c. A note on decompressing heterogeneous trees

For heterogeneous trees, the star bipartition can be unique for each tree. This is not an issue for homogeneous collections of trees, since the set of taxa are the same across all the trees. Thus, the star bipartition for homogeneous is an n-bitstring consisting of all 1s. For a heterogeneous trees with n' taxa, where $n' \leq n$, the tree is represented by a n-bitstring consisting of exactly n' 1s, where a 1 denotes the presence of a particular taxa in the tree, and a 0 denotes its absence. Thus, an addition initial step is needed to ensure that the star bipartition for each tree is properly computed.

To do this, we initially populate an n-bitstring of all 0s for each tree. By performing an `OR` operation over all the bipartitions contained in the tree of interest, one gets the appropriate star bipartition associated with the tree. One pass over the entire hash table is sufficient to generate the star bipartitions of all our trees in

Table V. Two sample tree files of weighted trees.

File 1
1. (((A:0.1,B:0.1):0.1,C:0.1):0.1,(D:0.1,(E:0.1,F:0.1):0.2):0.2);
2. (((A:0.1,B:0.3):0.2,D:0.2):0.2,(C:0.2,(E:0.2,F:0.2):0.2):0.2);
3. (((A:0.2,B:0.1):0.3,E:0.3):0.1,(D:0.3,(C:0.3,F:0.3):0.3):0.3);
File 2
4. (((E:0.1,F:0.1):0.2,D:0.1):0.2,(C:0.1,(A:0.1,B:0.1):0.1):0.1);
5. (((A:0.3,B:0.2):0.2,C:0.5):0.2,(F:0.4,(E:0.1,D:0.5):0.2):0.3);
6. (((A:0.1,B;0.4):0.2,C:0.2):0.2,(E:0.1,(D:0.2,F:0.3):0.5):0.4);

our collection. Once the star bipartitions are properly computed, the decompression procedure in TreeZip proceeds as described above.

3. Set Operations

One of our goals is to show that the TRZ file represents a viable alternative archive format to the Newick file for representing large collections of trees. If the same set of operations can be performed on a TRZ file that can be done on a Newick file, then we can argue that the two file types are equivalent. In order to accomplish this goal, we implemented a series of set operations that exploits the textual structure of the TRZ file to produce sets of trees of semantic interest.

The set operation functions in TreeZip takes as input two TRZ files and outputs a single TRZ file that represents the results of a particular set operation. Here, we implement three set operations in total: *union*, *intersection*, and *set difference*. The *union* between two collections of trees is defined as the set of unique trees that exist over both collections. The *intersection* between two collections is defined as as the set of unique trees that exist in the first collection *and* the second collection. The *set difference* between two collections is defined as the set of unique trees that exist in the first collection and *not* in the second.

For example, consider the collections stored in files 1 and 2 in Table V. In this

example, each file contains three trees. For simplicity, we label these $1 \ldots 6$. We note that the first tree in both files (trees 1 and 4) are identical; they just have a different Newick string representations. The rest of the trees are distinct from each other. The union between files 1 and 2 consists of of 5 trees (trees 1, 2, 3, 5 and 6). The intersection consists of 1 tree (tree 1), and the set difference consists of 2 trees (trees 2 and 3).

To perform set operations between files 1 and 2, TreeZip first computes a union operation on the bipartitions contained in the two files to determine the set of unique bipartitions contained over the two collections, or k'. Once k' is determined, a k'-bit bitstring representation can be created for each tree, based on a set permutation of the k' bipartitions. For a particular tree, its i^{th} location will receive a value of '1' if it contains bipartition i Else, it receives the value '0'. These tree bitstrings are then be fed through the h_3 and h_4 hashing functions, yielding a hash table representation showing the relationships between trees in our two files. Thus, trees 1 and 4 in Table V reside at the same $H[h_3.h_4]$ position, while the other trees reside on different positions. Set operations are computed directly on this structure.

TreeZip is able to perform these set operations (and other operations) quickly since the set of unique bipartitions and trees are *known* and already encoded into the TRZ file. Note that no special processing of heterogeneous trees are needed here. TreeZip then uses this encoded information to create a new TRZ file with the set of desired trees without needing to rebuild the tree structures. On the other hand, if one were to attempt to perform these operations on a Newick file, the bipartitions from each tree will have to be extracted and the relationships between the set of trees will have to be discovered every single time. As tree collections grow large, this can pose a significant overhead.

Lastly, we stress again that these set operations can all be performed on the input

TRZ files without any loss of space savings. This is of critical interest, as it shows the viability of using the TRZ file as an alternative format for storing trees. With standard compression methods, the resulting binary file must always be decompressed in order for any type of manipulation on the data to be performed. As a result, these could not be considered as alternative formats to the Newick file. The TRZ file on the other hand is a viable format because set operations can be performed on it. Furthermore, since there is no loss of space savings, the TRZ file is a more efficient way of storing collections of trees.

C. Experimental Analysis

We ran our experiments on fourteen sets of biological trees which are described in Table VI. The type is indicated by a two-letter code, where U indicated "unweighted", W indicates "weighted", B indicates "binary" and M indicates "multifurcating" trees, respectively. Our tree collections are very diverse containing between $2,505$–$150,000$ trees over 16–$2,594$ taxa. The size of the Newick tree files range from 2.6 MB for our smallest tree collection (`mammals`) to 434 MB on our largest collection (`insects`). Experiments were conducted on a 2.5 Ghz Intel Core 2 quad-core machine with 4 GB of RAM running Ubuntu Linux 8.10.Our implementation of TreeZip used in the following experiments can be found at `http://treezip.googlecode.com`.

1. Measuring Performance

We compare TreeZip to the `gzip` `bzip` and `7zip` compression algorithms. We measure the performance of our TreeZip algorithm in two primary ways: space savings and by using different, but equivalent Newick strings. Please note that `7zip` here represents a Newick file compressed with the `7zip` compression scheme.

Table VI. Characteristics of our biological tree files. The `mammals`, `freshwater`, `angiosperms`, `fish`, and `insects` datasets were given to us by biologists. The remaining tree collections are the same ones used by Boyer et al. to evaluate their TASPI approach. Size is shown in megabytes (MB).

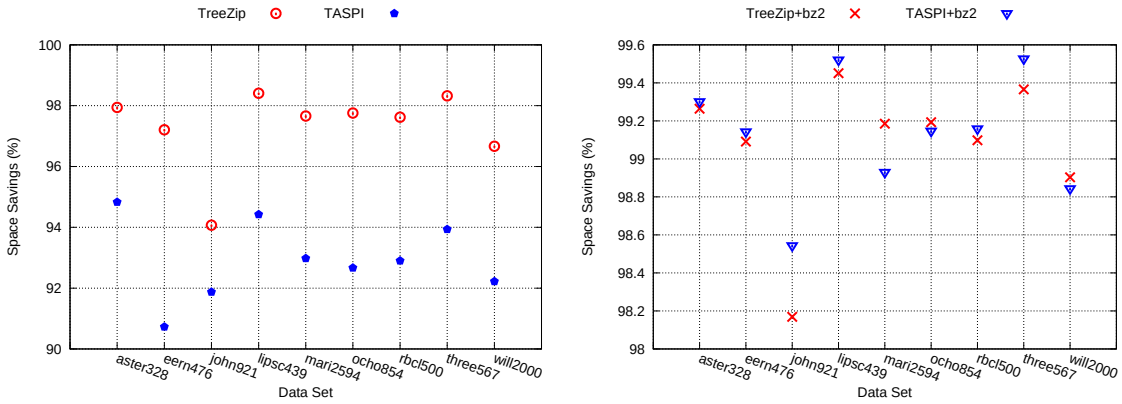
	Datasets	Description	Taxa	Trees	Size	Bipartitions	Type
1	`mammals`	Mammalian trees [93]	16	8,000	0.6	13	WB
2	`freshwater`	Organisms from freshwater, marine, and oil habitats [47]	150	20,000	67.0	1,168	WB
3	`angiosperms`	Flowering plants [37]	567	33,306	429.0	2,444	WB
4	`fish`	Fish trees (unpublished collection from M. Glasner's lab at Texas A&M)	264	90,002	533.0	12,115	WB
5	`insects`	Insect trees [48]	525	150,000	434.0	574	UM
6	`aster328`	Tree Collection 3 from Boyer et al. [72]	328	2,505	5.3	788	UB
7	`eern476`		476	2,505	7.7	3,019	UB
8	`john921`		921	2,505	16.0	15,448	UB
9	`lipsc439`		439	2,505	7.1	903	UB
10	`mari2594`		2,594	2,505	47.0	8,628	UB
11	`ocho854`		854	2,505	15.0	3,232	UB
12	`rbcl500`		500	2,505	8.2 (8.1 in [72])	1,579	UB
13	`three567`		567	2,505	9.3	1,588	UB
14	`will2000`		2,000	2,505	36.0	13,257	UB

Space savings and running time. We use the *space savings* measure to evaluate the performance of TreeZip in comparison to general-purpose compression algorithms. The space savings S is calculated as $S = 1 - \frac{\text{compressed file size}}{\text{original file size}} \times 100$. A higher space savings percentage denotes better compression of the original file. The goal is to get the level of space savings as close to 100% as possible. A value of 0% indicates no difference from the uncompressed, original Newick file. We also use running time to calculate how long each algorithm requires to compress and decompress a file. Time is shown in seconds.

Different, but equivalent Newick representations. As mentioned previously, for any given tree of n taxa, there are $O(2^{n-1})$ Newick string representations associated with it. Since general purpose compression methods such as `7zip` compress tree files by looking for redundancy at the Newick string level, they are unable to efficiently compress trees when there is a lack of redundancy in the Newick string representations. To illustrate this, we created a different, but equivalent Newick file for each dataset. For a Newick file containing t trees, each tree receives a different, but equivalent Newick representation. We note that using different, but equivalent Newick representations does not change the size of the resulting Newick file. For example, our `fish` dataset consisting of 90,002 trees over 264 taxa requires 533 MB of storage space. The Newick file containing different, but equivalent Newick strings still occupies 533 MB of disk space.

2. TreeZip vs. TASPI

To the best of our knowledge, The Texas Analysis of Symbolic Phylogenetic Information (TASPI) [71, 72] is the only described approached for compressing evolutionary trees. It is written in the ACL2 language. At a high-level, trees are represented



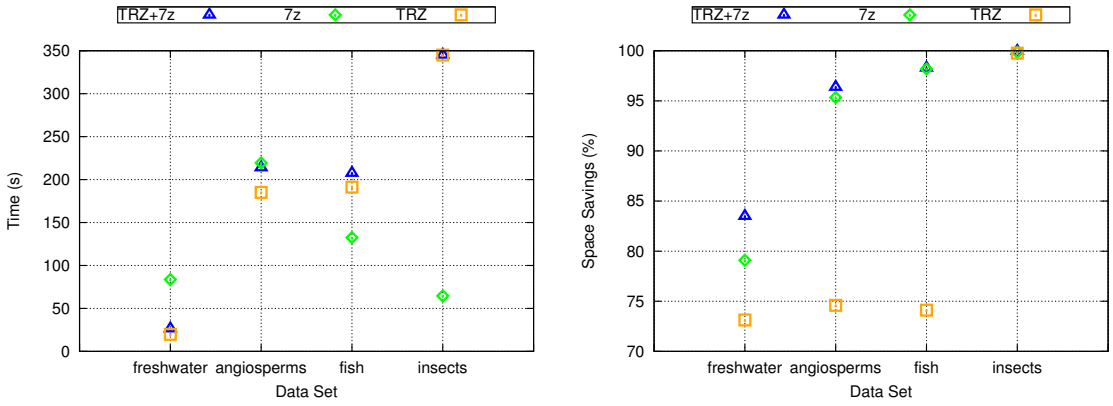
(a) TASPI vs. TreeZip (b) TASPI+`bzip` vs. TreeZip+`bzip`

Fig. 30. Space savings for various algorithms on Newick string representations of evolutionary trees. TASPI and TASPI+bz2 numbers come from [71].

as Lisp [94] lists, similar in appearance to Newick, but without commas and semicolons. Since an implementation of TASPI is not publicly available, we were unable to compare the running times of the algorithms.

The TASPI authors test the performance of their approach in two ways: (1) as a stand-alone TASPI compressed file and (2) applying the `bzip` algorithm to the TASPI file. We do the same for comparison. That is, we show results for TreeZip and TreeZip+`bzip`. In Figure 30, TASPI and TASPI+`bzip` are shown in blue, and TreeZip and TreeZip+`bzip` are shown in red. We compare the space savings achieved by the TreeZip and the TASPI approaches on nine trees used in Collection 3 of [72]. Since an implementation of TASPI is not available publicly, the space savings numbers for TASPI were calculated directly from [72].

When we compare TreeZip with TASPI in Figure 30(a), TreeZip achieves a better (higher) space savings than TASPI across all the listed datasets. For example, on the `lipsc439` dataset, the plain text TRZ file is 98.4% smaller than the input Newick file, while TASPI is only 94.4% smaller. This corresponds to file sizes of 116



(a) Compression Time (b) Space savings

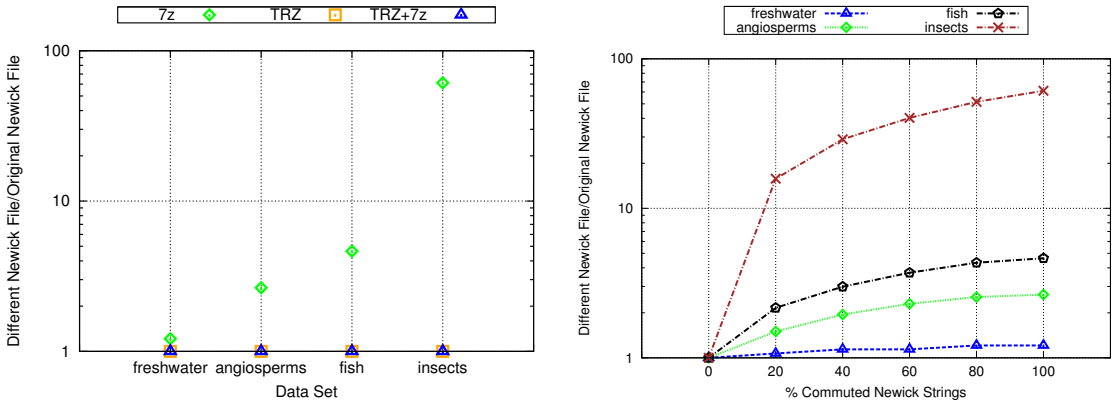
Fig. 31. Compression performance for our biological datasets. In this figure, (a) shows running time of compression approaches, while (b) shows space savings.

kilobytes and 406 kilobytes respectively. However, when coupled with `bzip`, TASPI achieves slightly better space savings than TreeZip+`bzip` on most of the datasets (Figure 30(b)). Since the TRZ format is plain-text, tree operations can be performed on it directly, without the need for decompression. This feature will help Noria perform real-time operations on compressed files.

3. TreeZip vs. Standard Compression Methods

a. Compression performance

Figure 31 shows the performance of TreeZip's compression algorithm. Figure 31(a) shows run-time information, and Figure 31(b) shows space savings results. On the `freshwater` and `angiosperms` datasets, we note that TreeZip is actually faster than 7zip. However, as the number of trees under consideration increases in size, so does the amount of time needed for compression. In terms of size, the TRZ file by itself is larger than the `7zip` file. However, we obtain an average of 75% space savings on our weighted collections, and about 99% space savings on our unweighted collection. The



(a) 100% Commuted Newick Strings (b) p% Commuted Newick Strings

Fig. 32. Compression performance for our biological datasets using different, but equivalent Newick strings. (a) 7zip experiences an increase in compressed file size when different, but equivalent Newick strings are introduced (100% commuted). (b) A closer look at how the percent of different, but equivalent Newick strings affect the increase in file size of 7zip (p% commuted).

discrepancy in space savings between the weighted and unweighted cases underlines the complexity of compressing branch lengths. However, we note that when the TRZ file is combined with `7zip`, the TRZ+`7zip` file has space savings on average of about 96%. `7zip` by itself, on the other hand, averages about 93% space savings. We also note that the higher the level of redundancy in a dataset (e.g. high bipartition sharing and duplicate trees), the better the compression. This is a key advantage, as tree collections returned from a phylogenetic search tend to contain very similar trees.

Effect of branch rotations on compression performance. *How robust are the different compression techniques under study to branch rotations?* To measure the effects of branch rotations on our datasets, we took each set of trees and gave them a random, but equivalent Newick string representation. We refer to this process as *commuting* the Newick representation. Figure 32 shows the performance of the various

compression schemes on different, but equivalent Newick string representations. The TRZ and TRZ+`7zip` files did not increase in file size. `7zip` took up to 4.4 times longer on this new file.

Figure 32(a) shows the change in space savings of the different compression schemes between the equivalent Newick files and the original files. Here, 100% of the Newick strings in the file have been commuted. A value of 1 signifies no change in file size. The space savings achieved by TreeZip and TreeZip+`7zip` does not change, despite the use of different, but equivalent Newick strings. This highlights TreeZip's robustness to branch rotations. This is not the case for `7zip`. On our weighted sets, the size of the `7zip` compressed file became almost 4 times larger. On the unweighted (`insects`) set, the `7zip` compressed file becomes 61 times bigger. This is equivalent of an increase of the size of the `7zip` compressed Newick file from 696 KB to 38 MB.

Figure 32(b) highlights the effect of these different, but equivalent Newick string representations on the efficacy of `7zip` to compress the file. The x-axis indicates the percent of the original file that received commuted Newick string representations. For each percentile, p percent of the trees in the file contain a different, but equivalent Newick string representation. The 0% mark is the original Newick file. All the datasets have a universal value of 1 at this point, since there is no change in the compression quality. The 100% mark is equivalent to the files that were used in Figure 32(a).

As one can see, as an increasing amount of the file is "randomized" at the surface with different, but equivalent Newick string representations, the performance of 7zip becomes increasingly worse. Thus, while TreeZip's compression approach is a little slower than `7zip`, its advantages in terms of robustness to branch rotations and great levels of compression gives it serious advantages. This is especially true when considering the fact that operations can be performed directly on the TRZ file, without

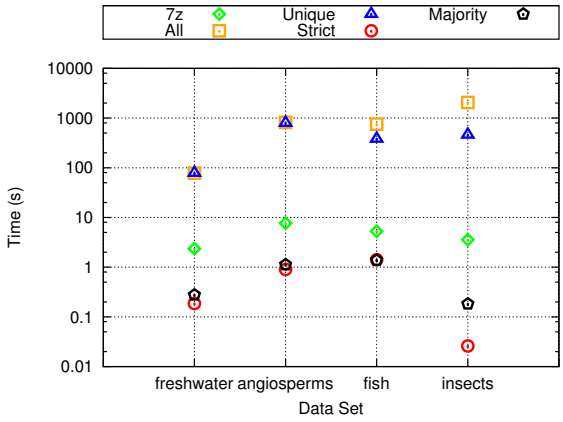


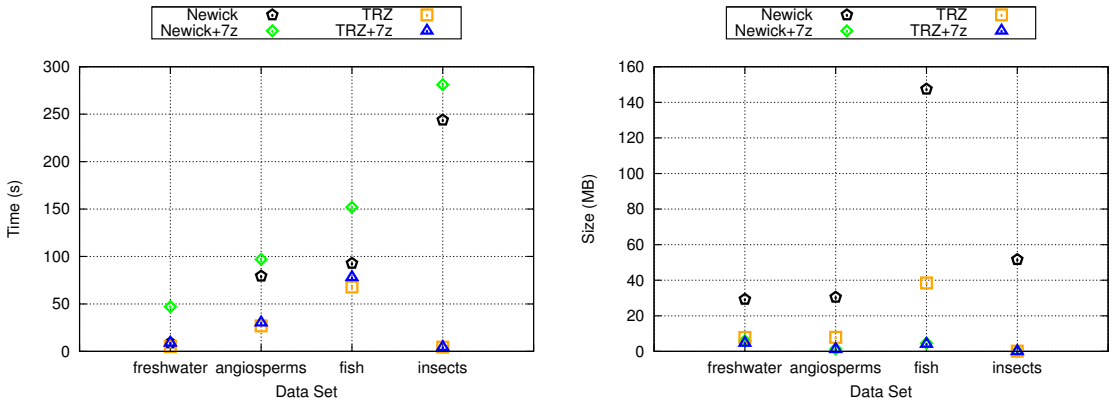
Fig. 33. Decompression performance for our biological datasets.

any loss of space savings.

b. Decompression performance

Figure 33 shows the decompression performance of `7zip` and TreeZip-based decompressors. `7zip` is a faster decompresser than the TreeZip-based approaches. For our two largest datasets (`angiosperms` and `fish`), `7zip` is two orders of magnitude faster than TreeZip and TreeZip+`7zip` at decompression. This is mainly due to the fact that TreeZip decompresses trees in a serial manner.

That said, the TreeZip is capable of decompressing subsets of trees of semantic interest to the user. The less the number of trees to be returned, the faster the procedure. For example, we can return the set of unique trees in a particular file, the strict consensus tree, or the majority consensus tree. Of particular interest is the fact that the strict and majority consensus trees on our datasets can be produced in less than second on average. This is something that no other algorithm has been able to accomplish so far on these large datasets. TreeZip's consensus operation is 150 times faster than HashCS [78], the fastest algorithm for computing consensus trees. We note that HashCS has been found to be 100 times faster than the consensus procedure in



(a) Time (b) Space Savings

Fig. 34. Performance of set operations on our biological datasets. (a) The running time of a random collection of set operations run on different file formats. (b) The amount of disk space required by the result of the set operations.

MrBayes [78].

We would also like to note that none of these operations can easily be performed on a `7zip` file. Since the `7zip` file is binary, all the trees would necessarily have to be decompressed before trees of semantic interest can be extracted, which can result in significant overhead. We explore this in greater detail in our set operations experiments, shown in Subsection c.

c. Set operations performance

Figure 34 shows our performance results of set operations performed on Newick files, Newick files compressed with `7zip`, TRZ files, and TRZ+`7zip` files. Figure 34(a) shows run-time results. On weighted trees, it is up to 3 times faster to perform set operations on TRZ files over Newick files. On the unweighted case, it is about 55 times faster. While there is little overhead in combining the TRZ file with `7zip`, there is more significant overhead when combining Newick files with `7zip`. While it is only at most 10 seconds slower to combine `7zip` with a TRZ file for set operations,

the overhead of combining `7zip` with a Newick file is as much as a minute. As a result, the speedup results are more significant when comparing set operations on a TRZ+`7zip` file versus a Newick+`7zip` file. On weighted trees, the speed up is at most 5.25. On unweighted trees, the speedup is up to 62.4. We note that the vast differences in speedup between the unweighted and weighted cases has to do with the fact that TreeZip has to do a lot of extra processing in order to handle branch lengths. Since `7zip` does not view branch lengths as being any different from raw text, there is no additional processing needed.

Figure 34(b) shows the average space savings of storing the results of set operations in a TRZ file versus a Newick file, TRZ+`7zip` file and a `7zip` file. In terms of size, the results of set operations are more efficiently stored in TRZ files than Newick files. On weighted trees, the TRZ file storing the results of the set operations is 74.1% smaller than the Newick file. On unweighted case, it is up to 99.7% smaller. This is very consistent with the general space savings of using a TRZ file over a Newick file on weighted and unweighted trees respectively. TRZ+`7zip` files have at most a 21% improvement in space savings over Newick+`7zip` files in the weighted case. On the unweighted case, the TRZ+`7zip` takes up 78.75% less space than the Newick+`7zip` file.

The TRZ+`7zip` file produced up to 5 times faster than the Newick file compressed with `7zip`, takes up less space. There are no space savings using a Newick file, and we reiterate that the files compressed with `7zip` have to be decompressed entirely before set operations can be performed on them. This is another reason why set operations on files compressed with `7zip` are slower than those compressed with TreeZip+`7zip`. Together, these results underline the benefit of using the TRZ file for set operations versus using a Newick file or a file compressed with `7zip`.

D. Summary

There is a critical need for phylogenetic compression techniques that reduce the space requirements of large tree collections. Phylogenetic searches typically produce tens to hundreds of thousands of candidate evolutionary trees. Furthermore, when these techniques are used to reconstruct larger trees, even larger number of candidate trees are produced. Our results strongly suggest that TreeZip has the potential for playing a vital role in how biologists manage their phylogenetic tree collections.

The TreeZip algorithm is valuable for several reasons. First, allows for the compression of binary and multifurcating trees, including those that contain (or do not contain) branch lengths. Second, we offer an extensible decompresser which allows for filtering and extraction of sets of trees of interest. Lastly, we offer functionality that allows for extremely fast set operations directed on the TRZ file without a loss of space savings.

By itself, TreeZip compresses a Newick file into a plain text compressed representation that is at least 73% smaller than the original file on the weighted case, and over 98% smaller on the unweighted case. When combined with `7zip`, the TreeZip+`7zip` file achieves an average space savings of 92% on the weighted case, and space savings of over 99% on the unweighted case. Another key advantage is that the quality of compression improves with the level of redundancy in the collection of trees. This makes it the best method to compress large collections of phylogenetic trees.

Furthermore, in cases of high degrees of bipartition sharing, our TreeZip algorithm is able to compress a collection of trees faster than `7zip`. Lastly, TreeZip is robust to different representations of a phylogenetic tree allowing the TreeZip and TreeZip+`7zip` compressed formats to achieve consistently good space savings. The space savings achieved by `7zip`, on the other hand, decreases as the number of dif-

ferent Newick representations for the same phylogenetic tree increases.

TreeZip's most powerful advantage arises from the TRZ file's textual format. This allows the TRZ file to be used as input to phylogenetic applications without a need to rebuild all the trees into Newick format. Given the flexibility of the TRZ format, we can easily design extensible decompressers that extract the relevant phylogenetic tree information of interest. Our results illustrate two decompresser applications—removing duplicate trees and creating consensus trees—that can be created to extract information from the TRZ file. We show that due to the textual form of the TRZ file, set operations can be performed up to five times faster than on a Newick file and store results in as much as 99% less space. This is not functionality that standard compression methods such as `7zip` can achieve since they cannot leverage the phylogenetic information contained within the Newick representation of a phylogenetic tree.

For the biological community, TreeZip and the TRZ format is an efficient way to store trees. Since TRZ files can with little overhead be further compressed by standard compression methods, large datasets can now be more easily shared. For the large datasets studied in this section, the corresponding TreeZip+`7zip` files are small enough to e-mail. For biologists interested in leveraging the most out of their phylogenetic tree collections, the textual TRZ format allows operations to be performed directly and quickly, allowing for rapid information gain. With Newick files, bipartition information will have to be re-discovered every time the data is analyzed. The $1:1$ relationship between a collection of trees and its TRZ file also means that it is now easy to compare collections of trees with each other.

The success of TreeZip and the TRZ compressed format also highlights the ability of hashing to eliminate redundancy in tree-like data. A textual encoding of such a representation allows for both efficient storage and fast retrieval of information of

interest. Thus, we strongly believe that the work here will be of interest for storing other tree-like data, such as HTML and XML files.

Lastly, since the TRZ format is a $1:1$ representation that decomposes trees into the smallest set of discrete units (unique trees and bipartitions) it can be thought of as an *object store* for tree collections. In this context, the object store represents a database of topological units that represents tree data. This makes the TRZ format an ideal candidate for representing trees in the context of a version control system, the speed and efficacy of which is partially dependent on its ability to maintain the smallest set of objects needed to represent the data. In the chapter that follows, we discuss how TreeZip and the TRZ format plays a vital role in the construction of Noria, a version control system designed specifically for phylogenetic trees.

CHAPTER VII

A NEW SYSTEM FOR ANALYZING AND STORING TREES

Throughout our work, we have demonstrated the power and versatility of the hash table for accommodating the burgeoning growth of phylogenetic data. With MrsRF and Phlash, we showed how universal hashing can be used to compare large collections of evolutionary trees. Our work with TreeZip shows that a natural consequence for hashing trees is removing redundancy in a collection of interest, thus allowing relevant data to be encoded in a compressed manner. Our research question is, *how can we design new approaches that enable biologists to seamlessly manage their large tree collections and share their results with the phylogenetic community in order to build more robust evolutionary trees?* In this chapter, we present Noria, a version control system designed for phylogenetic data in which the TreeZip algorithm plays a major role in the effectiveness of the system. A noria is a type of waterwheel used for irrigation purposes. Like its physical counterpart, we hope that Noria becomes a vital mechanism in the (biological) community, and helps nourish the field with new knowledge about collections of trees.

A. Motivation

The ultimate goal of computational phylogenetics is the reconstruction of a phylogenetic tree of interest. To infer phylogenies, biologists use fast heuristics that tend to return tens to hundreds of thousands of trees. In Chapter II, we demonstrated that these collections are only growing larger. A *version control system* (VCS) allows users to share their code with each other and track the evolution of a software project. A version control system that tracks and manages changes to tree collections would be very useful, since it would give scientists a record of how their tree collections evolved

over time, and allow for greater reproducibility of results.

Central to a VCS is the *repository*, which is the database that retains and manages the history of the project [95]. In recent years, *decentralized distributed version control systems* (DVCSs) like Mercurial [17] and Git [19] have been gaining momentum and popularity. In a DVCS, revisions can be made locally and independently to a local repository without needing to be connected to a remote server. Users can share their changes with others by placing them in a public repository. Since multiple people have a copy of the public repository, a DVCS has natural protections against data loss. We believe that a system designed to handle phylogenetic data that leverages the features of existing distributed version control systems would be widely used in the biological community.

B. Version Control and Source Control Management Systems

It is impossible to discuss Noria, a version control system for phylogenetic trees, without discussing version control systems in general. Version control (or revision control) systems offer a way to track and manage changes to files. In the software engineering world, the use of version control systems are essential for tracking and managing changes to source code, and provide a collaborative way for multiple users to make changes to code.

Figure 35 shows the family history of several popular version control systems. Source Code Control System (SCCS) [96], written in 1972, was one of the earliest version control systems used on UNIX systems. Its use was largely replaced by Revision Control System (RCS) [97] in the 1980s. However, RCS and SCCS only work on single files. Along with one of RCS's successors (PRCS) [98], these early version control systems depended on a *local* repository on the user's computer. All of these systems

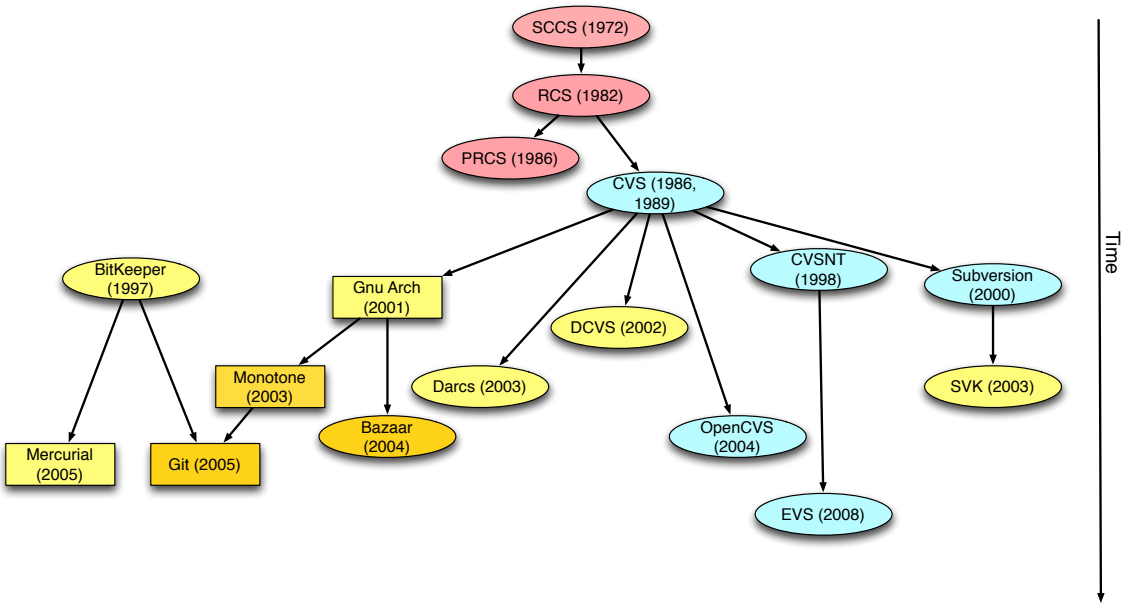


Fig. 35. A family history of popular version control systems. Local repository systems are colored in red, centralized collaborative repository systems are colored blue, and distributed, de-centralized collaborative repository systems are colored yellow. Patch-based systems are a light shade. DAG-based systems are a dark shade. Lastly, systems which use cryptographic hashing are square, while those that don't are round.

were designed for single users to maintain changes to their own projects. However, most large projects require the involvement of multiple developers and multiple files. In these cases, a local repository system is not a workable solution.

1. Collaborative Version Control Systems

Given the need for maintaining large projects in which files are maintained by multiple users, it is not surprising that the natural progression in the evolution of software version control systems was the creation of collaborative (distributed) repository systems. These fall into two categories: centralized and decentralized. We discuss each of these in turn.

a. Centralized version control systems.

Centralized repository systems store the main repository on a server. Perhaps the most famous of the central repository systems is the Concurrent Versions System (CVS) [99]. These systems use a "client-server" model, in which the repository is stored on a centralized server, which each developer (client) can access. To get the latest version of the project, a developer connects to the server and "checks out" the latest version. Once they are done with their changes, they re-connect to the server, and "push" their changes.

Practically all modern centralized version control systems, such as Subversion [18] and OpenCVS [100] are either derived from, or designed to replace the CVS system. Concurrent Versions System is also notable for introducing the concept of *branching* to version control systems, which appears in CVS documentation dating to 1990. In short, branching allows for the duplication of revision objects, so that parallel development can occur.

Today, Subversion is the dominant centralized version control system in use.

Unlike CVS, commits in Subversion are *atomic* (small, quick, and guaranteed). Lastly, instead of tracking entire files like CVS, Subversion tracks changes to the files, also referred to as *patches*.

b. Decentralized version control systems.

The use of a centralized repository system is not without its faults. Perhaps the largest risk to a centralized repository system is the existence of one "master" repository. This represents a single point of failure. In other words, if the project goes off-line for a few hours, no one can commit during that time. If the hard disk is corrupted, all the data for the project would be lost. In a distributed decentralized repository system, every developer has their own copy of the repository. Thus, if any particular repository were to go off-line or be lost, the contents can be recreated from the other repositories in existence. Also, since each user has his/her own copy of the repository, they can continue making commits to their *local repository* while they are off-line. These changes can then later be shared with everyone else when they push to their *public repository*, which anyone can *pull*, or receive changes, from.

Gnu Arch [101] is one of the earliest open source distributed, decentralized version control systems. Like Subversion, Gnu Arch makes use of atomic commits, and track changes using patches. However, Gnu Arch also has some important features not found in its centralized predecessors. Most notable is the use of cryptographic hashes to safely index and store objects, preventing corruption. This feature is inherited by some of Gnu Arch's most notable successors, such as Monotone [102] and Git.

Monotone, developed in 2003, uses cryptographic hashing like Gnu Arch, but differs fundamentally in the way it track changes. Instead of using a patch-based model to track changes to a file, Monotone tracks changes to a file using a directed acyclic graph (DAG). A DAG-based system of tracking changes essentially takes a

snapshot of a file system as it changes from commit to commit and stores these snapshots internally in a meaningful way that they can easily be accessed. Common systems that used a DAG-based approach include the `cp -r` UNIX command and the Time Machine application in OS X.

It is difficult to discuss the history of modern open source version control systems without mentioning BitKeeper [103]. BitKeeper is a proprietary version control system designed by BitMover Inc. that, prior to 2005, used to host a series of open source projects, most notably the Linux kernel. In 2005, however, BitMover changed its pricing model and announced that it would no longer provide a free version to open source project developers. This action spurred Linus Torvalds, the developer of the Linux kernel, to develop Git. Today, Git is arguably the most popular open source version control system in use. Unlike other version control systems, Git tracks *content*, not *files*. We use Git as the basis for our Noria version control system for phylogenetic data.

2. Customizing Version Control Systems For the Needs of a Community

The version control systems discussed so far were designed for managing text files and source code. However, given the intricacies of different kinds of data, these version control systems are not suitable for managing *all* kinds of data. The natural solution is to customize revision control software to reflect the needs of different communities.

One of the most popular examples is the Microsoft Track Changes feature, which is essentially is a revision control system Microsoft built into its popular Microsoft Word program. The Track Changes option, when turned on, implicitly tracks changes in a user's document. Users can view or hide these changes by using different options in the Track Changes menu. When a user shares his or her document with a collaborator, this collaborator can also view the changes made to the document. In this

manner, authors have implicit version control, allowing them to track changes to their Word documents (which are binary), and revert their documents to previous states as necessary. It is not possible to track changes in Microsoft Word documents in most of the version control systems mentioned in Section B, as these systems were not designed to track changes within binary files, or understand the proprietary format of Microsoft Word.

Another example comes out of the designer community. Unlike coders and developers who deal primarily with text files that contain source code, designers care about making and tracking changes to images, which are binary files. Again, the standard version control systems mentioned in Section B are not directly applicable, due to the need to track differences in binary files. However, version control systems such as Git and Mercurial are helpful to designers — text files can be used for brainstorming, and for a team of designers to share their ideas with each other. As a result, members of the design community chose to *extend* the functionality of popular version control systems to suit their needs.

Some prominent examples include Adobe Cue [104] and PixelNovel Timeline [105]. Adobe Cue has a GUI interface that guides users through the setup of a remote repository, tracking changes, and viewing previous versions. PixelNovel Timeline is a front-end to Subversion that is integrated into Photoshop. One major advantage is that it comes with a hosted online repository on PixelNovel that a user can use immediately, without any need of setting up their own. Kaleidoscope [106] is an OS X tool for helping users spot the difference between text and image files. Using a single screen interface for maximum readability, it uses colors to highlight changes between different versions of text. It also allows users to spot differences between images, making it ideal for designers.

These examples highlight the need of version control systems to adapt to the

needs of a community, rather than the community adapt to a version control system. Tracking the provenance of data is a fundamental part of most work-flows and a fundamental right — there is no reason for users to change their work-flows in order to gain access to these features. This is the primary motivation behind our creating a customized version control system for managing phylogenetic analyses.

3. Focus on MrBayes

Our examples in this dissertation and our initial version of Noria focuses on Bayesian phylogenetic analysis. We focus on the MrBayes implementation of Bayesian analysis since (as we showed in Chapter II) Bayesian analysis produces the greatest number of trees in a phylogenetic analysis, and MrBayes currently the most popular software used for Bayesian analysis. As input, MrBayes takes in a NEXUS file that contains input data (stored in a `DATA`) block and parameter data (stored in a `MRBAYES`) block, such as the number of runs and type of model to use. It outputs a series of NEXUS files, with trees (stored in `TREES`) blocks stored separately by run.

We will also show how Noria alleviates some of the problems with running and managing Bayesian analyses. For example, Bayesian analysis associates input and output of files through file extensions. If an input data set was labeled `input.nex`, and we ran two runs of Bayesian analysis, two tree files will be produced: `input.nex.run1.t` and `input.nex.run2.t`. If a user forgets to save the contents of the analysis before making changes to the model, data, or run parameters, the proper associations can be lost. Thus, even if a standard version control system is used to help a user manage their Bayesian analyses, the user must remember to commit and save their analyses at every turn. For scientists who are typically unfamiliar with version control, this can be a tall order. On the other hand, Noria performs implicit saves, thus putting very little imposition on biologists. We also note that

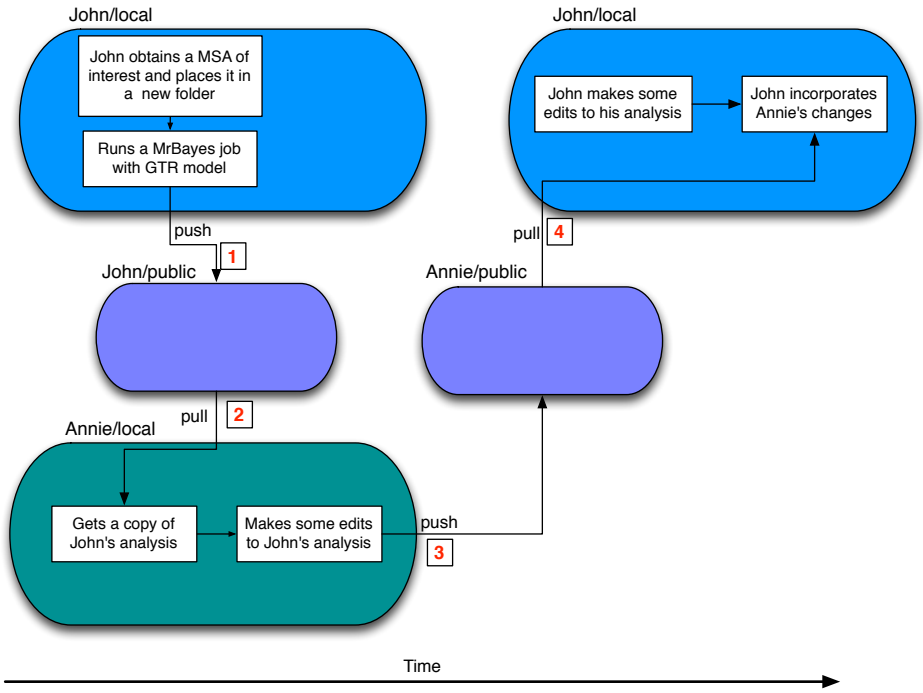


Fig. 36. Two users sharing their phylogenetic analyses.

while MrBayes and Bayesian inference is our focus for this initial version of Noria, the Noria system can easily be augmented to perform other types of phylogenetic search applications, such as TNT and RAxML.

C. A Collaborative Scenario

In Figure 36 we show a scenario in which two biologists are sharing trees. In these examples, John and Annie are two researchers who are collaborating on the same project. Each researcher maintains their own repository on their computer (shown as name/local). Thus, Annie's local repository is referred to as Annie/local. In addition, each person has access to a public repository in which other members can can access (shown as name/public). Thus, Annie's public repository is Annie/public. Public repositories are placed in a shared, common location that all users can access. A *push*

denotes a user sharing their analyses by placing it on the public repository. A *pull* denotes a user receiving analyses from a repository. The arrows denote the direction of movement of the phylogenetic data, which we label to denote the progression of sharing. The direction of time flows from left to right in this example, with later operations being shown on the right of earlier operations. Each collaborative step is denoted with a number. For example, step 1 is referred to as "1" in Figure 36, and denotes the act when John shares his analysis by pushing it to his public repository.

In this example, John is learning how to run a phylogenetic analysis. He runs a MrBayes analysis on the data and saves the results of his analysis in the repository. In step 1, he shares his results, by pushing them to his public repository. In step 2, Annie obtains a copy of John's analysis and takes a look at it. Realizing that he has a number of sub-optimal trees, she removes a number of trees from the beginning of his analysis (a process referred to as "accounting for 'burn-in'"). After saving her changes to her local repository, she pushes her changes (step 3) into her public repository. Meanwhile, John continues to generate additional trees, and saves them to his local repository. After doing a quick check on Annie's repository, he realizes that she has made some updates, and decides to pull and combine (merge) her changes with the work that he had been doing (step 4).

1. Standard Version Control Systems

Why are existing version control systems unsuitable for tree collections and phylogenetic analyses? Version control systems such as CVS, Subversion, Mercurial and Git were created to manage revisions to source code. As a result, they deal with data in the system as uninterpreted text. Figure 37 shows John and Annie sharing their phylogenetic analyses using a standard distributed version control system like Git. For simplicity, we focus on a very small collection of trees. Additions to the file are

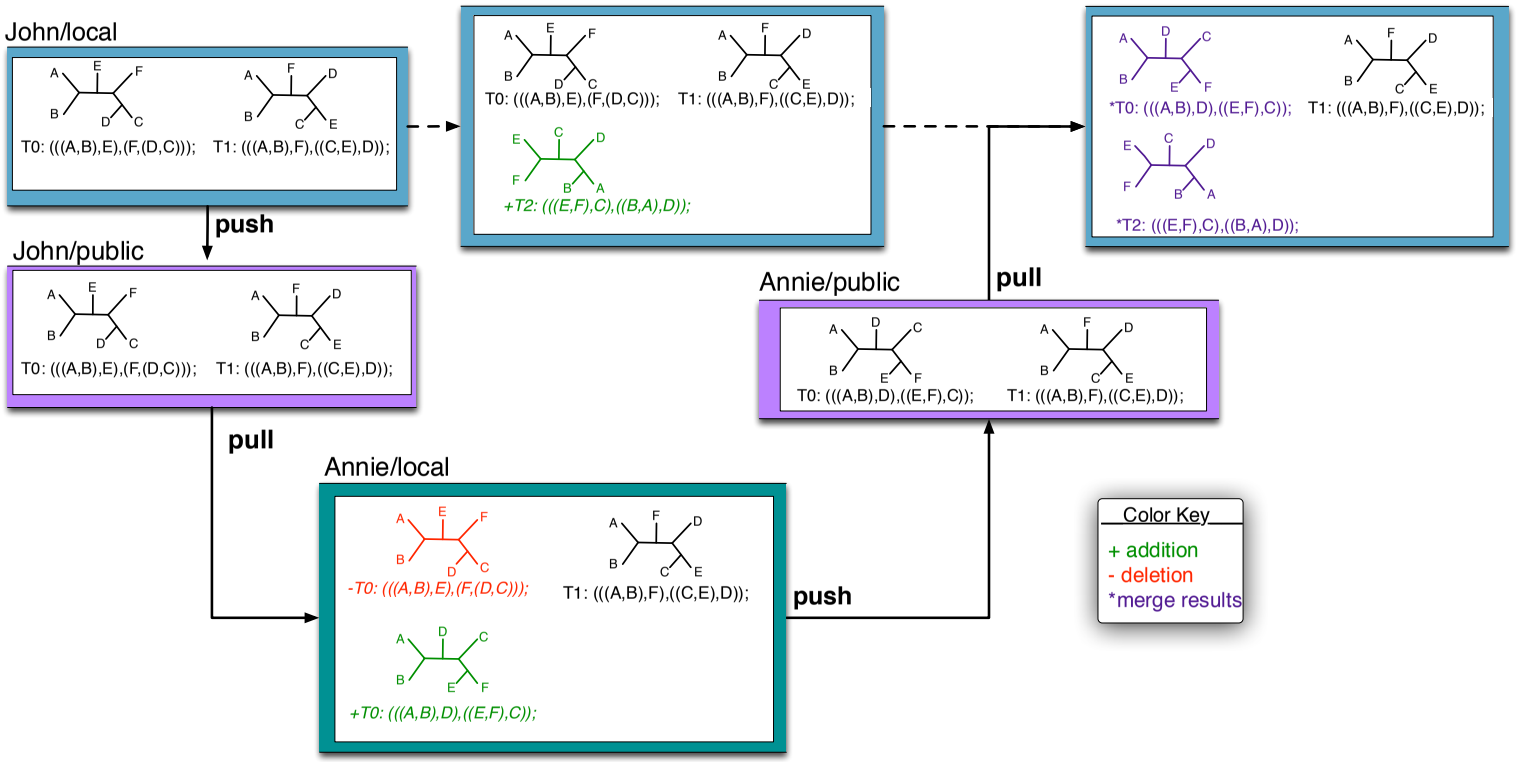


Fig. 37. How our two users from Figure 36 share tree collections using a standard version control system like Git. For the sake of simplicity, phylogenetic analyses are shown as tree collections. All trees are across the taxa set $A \ldots F$.

marked with a '+' and is shown in green. Deletions to the file are marked with '-' and are shown in red. The trees affected by the merge are marked with '*' and are shown in purple.

After John generates a set of trees, he shares them with Annie by pushing the contents of his local repository into his public repository. Next, Annie pulls the contents of John's public repository, and replaces the first tree with a different tree. She then shares her results by pushing the contents of her local repository to her public repository. At this point in time, there are only two trees in Annie's public repository: the tree that she replaced, and the tree that John originally produced. Meanwhile, John has added a new tree to his collection. Unknown to him, however, this new tree is equivalent to the tree Annie had added! Since John's tree has a different Newick string representation than Annie's trees (and they are located in different locations), a standard version control system would identify these trees as being different and would store them both. This would result in duplicate trees being added to the system when John pulls from Annie's public repository.

2. Noria

Next, we demonstrate how Noria handles the same situation. In Figure 38, John and Annie are collaboratively using Noria. Noria uses the TRZ file format as its base, which stores trees uniquely as a set of bipartitions, *not* strings. The set of bipartitions are shown as a list of (key, values) pairs. The key is the bipartition, while the values are the set trees that contain that particular bipartition.

When John pushes his trees to his public repository using the Noria system, he is sharing the bipartitions contained in the two trees he found, T_0 and T_1. When Annie pulls from John's public repository and replaces the first tree with a new one, this affects a set of bipartitions. Notice that since bipartitions $AB|CDEF$ is contained in

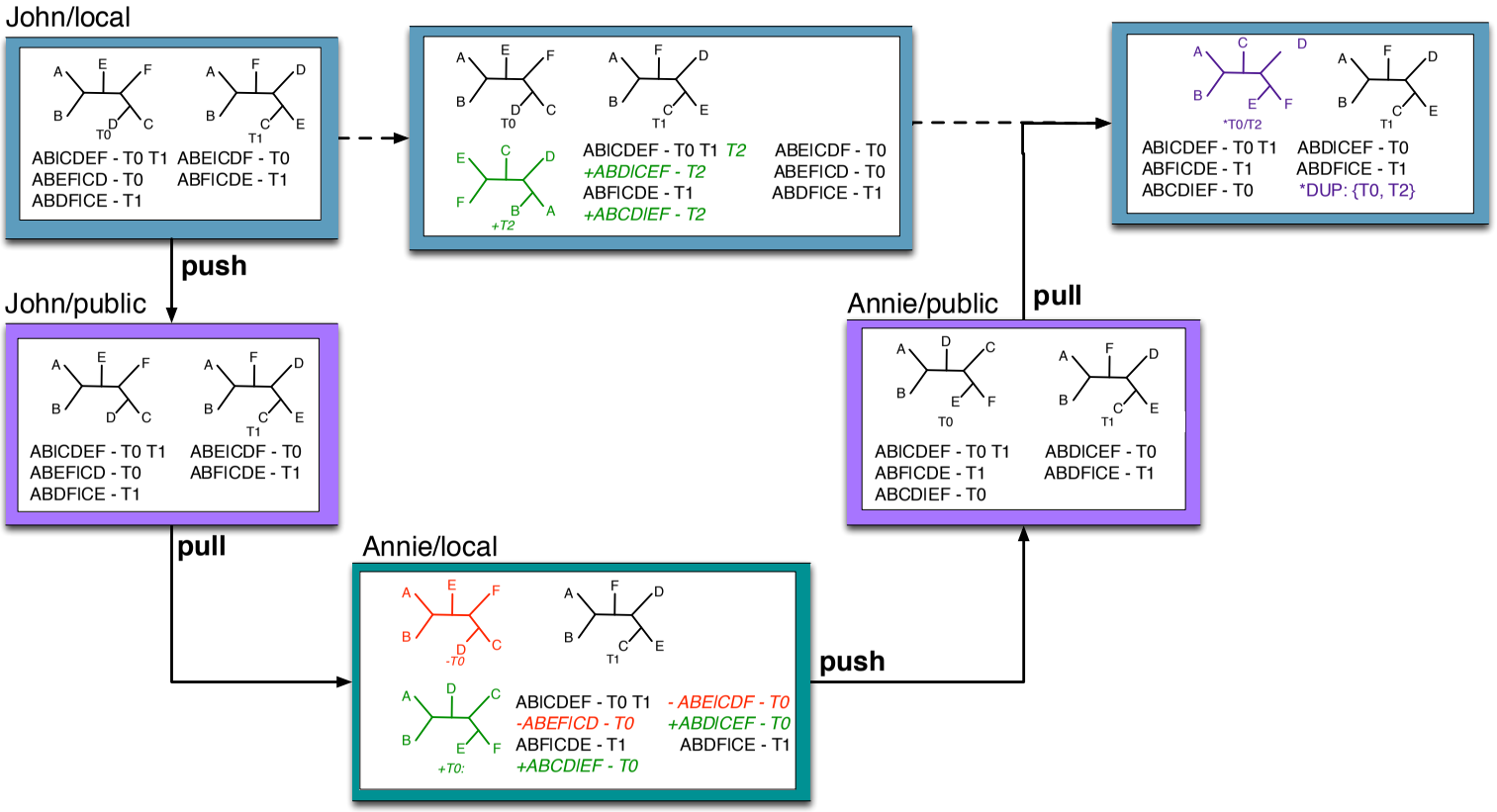


Fig. 38. How our two users from Figure 36 share tree collections using Noria. For the sake of simplicity, phylogenetic analyses are shown as tree collections. All trees are across the taxa set $A \ldots F$.

both the old and new T_0 it is not affected by Annie's change. Annie then shares her trees by pushing them to her public repository. Likewise, when John adds a new tree, he is essentially adding new bipartition information. So what happens when John pulls from Annie's public repository? Unlike the situation in Figure 37, Noria is able to correctly detect that only one new tree was added to the collection. As a result, John's repository contains only the bipartitions from Annie's trees. Since the TRZ file keeps track of unique trees as well as unique bipartitions, a pointer is added in the file that indicate that trees T_0 and T_2 are identical. Since the TRZ file's basic unit is the bipartition, Noria is able to keep track of a smaller set of objects and make statements about relationships between those objects. Again, this is not possible with a standard version control system. John now shares the work he did with Annie by pushing his changes to his public repository, which Annie can then access. In this manner, these two scientists can always see the work that they and their research partners have done, no matter where they are. This system also succeeds in giving each biologist a record of how their analyses evolved.

Why can't we just use Git or another standard version control system to track TRZ files? Since the TRZ file forms the core of the Noria system, one may ask why a standard version control system could not be used to simply track TRZ files. The reason for this is two-fold. First, current phylogenetic applications deal largely with Newick-formatted tree files. To ask biologists to operate entirely with a format that they are unfamiliar with is unreasonable, and will prevent rapid adoption of the system. While Noria depends on the TRZ format, it is hidden from the user's view. Second, Git and other standard version control systems will treat the TRZ file as unstructured text. Thus, it will be impossible to properly perform a *merge* on two TRZ files. One of the key features of Noria is the presence of a merge algorithm that is designed to properly merge TRZ files. We discuss this in length in the next section.

D. The Noria System in Depth

Given the success of version control systems to help source code developers manage their large projects, we wish to apply the principles of revision control systems to help scientists manage their phylogenetic analyses. Such a system will allow scientists to record how their analyses evolved over time, and allow for greater reproducibility of results. To this end, we've developed Noria, a customized DVCS built on top of Git to help scientists manage their large tree collections. Since Noria uses the TRZ file as its basis for objects, tree collections are stored more efficiently than in a standard version control system. Furthermore, Noria allows the user to compare their experiments, allowing biologists to fully leverage the knowledge contained in their trees, and understand the ramifications of their experimental parameters. In other words, Noria does not simply track the content: it actively aids in scientific analysis.

1. Architecture of the Noria System

Noria was implemented in C++, and utilizes several of the functions in the TreeZip program. Built on top of Git, Noria also utilizes LibGit2 [107], a novel library that allows users to incorporate Git functionality directly into their code. Currently, LibGit2 is still in its early stages of development. As the library continues to mature, we plan to further integrate the library's functionalities with Noria. Our system also has functionality to allow integration with the NEXUS Class Library (NCL) [108], which was designed to simplify the process by which a piece of software would interpret a NEXUS file. We had originally explored NCL as a way to normalize NEXUS files that users would input — unfortunately, we found a couple of incompatibilities with the normalized NEXUS files and the current version of MrBayes. NCL integration

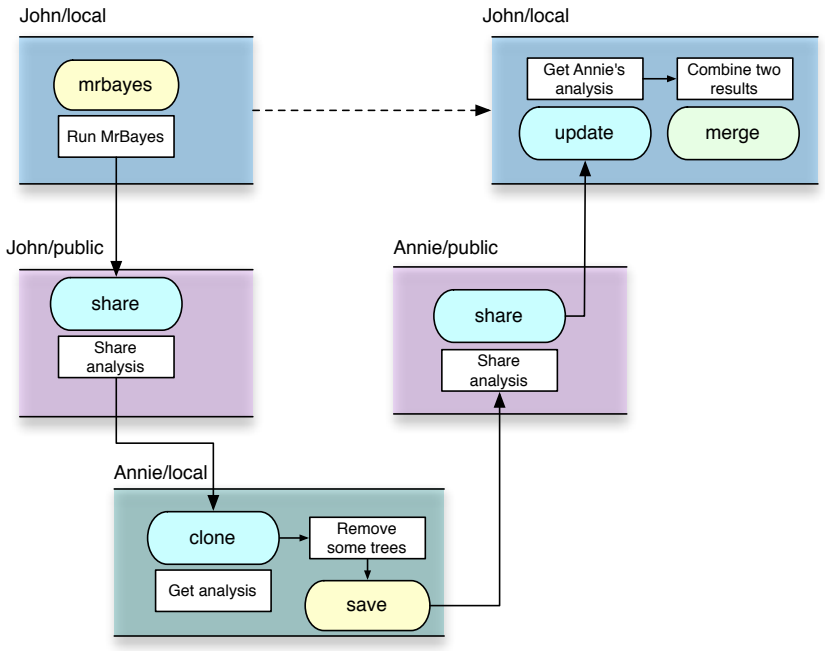


Fig. 39. Commands our two users use in the context of the Noria system to perform the collaboration shown in Figure 36. Local commands are shown in yellow, while distributed commands are shown in blue.

will be available in MrBayes 4.0, which is currently in development. Thus, we have kept our code (albeit inactive) for NCL integration as part of Noria. By itself, Noria contains over 7,000 lines of code. The prototype of the system is available as a virtual machine from the author.

Figure 39 gives a glimpse of some of the internal commands used by Noria to allow scientists to run their phylogenetic analyses in the context of the system, share their results with each other, as well as compare and combine their analyses with each other. While the commands shown in Figure 39 represent just a subset of the currently implemented 18 commands one can perform in the system, it does show the most important commands and how they interact with each other in the context of our collaboration scenario shown in Figure 36. Commands that interact only within a

local repository are shown in yellow. Commands that are used to transfer information between repositories are shown in blue. Merge, which can be used both locally and also to combine work done remotely, is shown in green.

To run his MrBayes analysis, John uses the `mrbayes` command in Noria. This function acts as a wrapper around the MrBayes program. When John runs MrBayes in the context of the Noria system, his results are automatically saved to his local repository once the analysis is finished. This means, if John runs an analysis that runs for several days, he does not need to remember to explicitly save his results, as he would need to if he were using a standard version control system like Git. In this manner, John does not need to modify his work-flow — Noria takes care of basics of the version control component implicitly and in the background.

To share his analysis with Annie by pushing his his results to the public repository, John uses the `share` command. Annie then *clones* a copy of John's repository, creating her local repository. The `clone` command essentially creates a Noria repository, links it to the John's public repository, and copies all the information contained in the John's public repository into Annie's local repository. A Noria repository can be created separately by running the `noria` command. Similarly, a user can host their local repository as a public repository using the `host` command, or attach their local repository to an existing public repository using the `attach` command.

After Annie modifies John's analysis, she then uses the `save` command to explicitly capture the current state of her local repository. Since a biologist running a Bayesian analysis often does a variety of other things aside from just running MrBayes (such as creating figures), an explicit save command is very useful. Once Annie is done with her adjustments, she shares her changes using the `share` command. John then updates his repository with the `update` command. This essentially creates a copy of Annie's analysis and places it on a separate branch on John's repository. To

combine his work with Annie's changes, he runs the `merge` command.

2. Augmenting the TRZ format

As we mentioned, the basis of the Noria system is the TRZ format, which we discuss in length in Chapter VI. However, the TRZ file that forms the basis of TreeZip was designed to compress tree collections. *How do we augment the TRZ format from simply compressing tree collections to compressing phylogenetic analyses?* The answer lies in extending the TRZ file format to handle NEXUS files.

As mentioned previously in Chapter III, NEXUS files store the information pertinent to a phylogenetic analysis in a series of blocks. MrBayes deals exclusively in NEXUS files: they form the basis for both the input and the output of MrBayes program. Thus, we augmented TreeZip and the TRZ format to support these blocks. Tree collections are now stored in the context of a `TREES` block. In the updated TRZ format, blocks appear in a normalized order. This helps ensure the one to one correspondence between a NEXUS file and its corresponding TRZ file.

To ensure that the input of a phylogenetic analysis is always associated with its output, we also associate every tree collection produced by a Bayesian analysis with the input blocks used to generate it. Thus, every tree collection in Noria is guaranteed to have model information and input data information associated with it. This is not a guarantee that can be made with a standard version control system, as MrBayes loosely associates input with output by using file names. If one of those files were accidentally overwritten before a user were to perform an explicit commit, valuable information would be lost, and the analysis may need to be rerun. When the `mrbayes` command is run inside of Noria, every file containing trees is automatically associated with the data, models, and parameters used to produce those trees. Thus, even if the file is renamed, the associations are not lost.

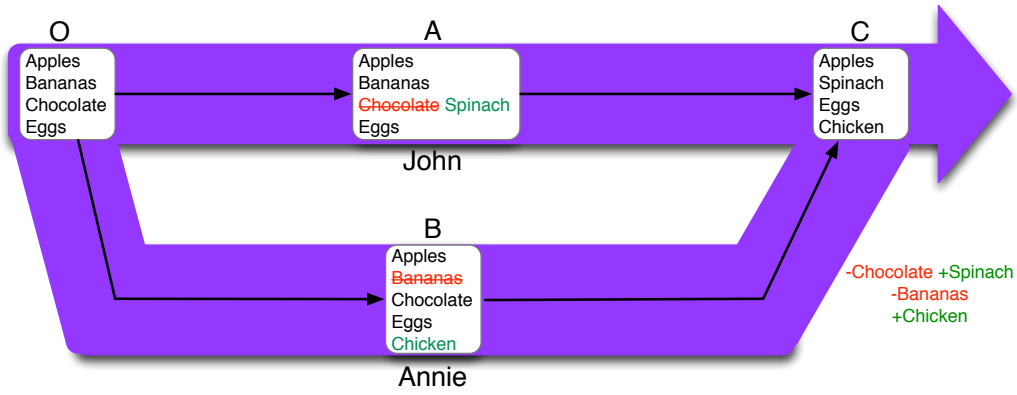


Fig. 40. Example of a three-way merge on a grocery list of items.

3. Merging Trees in Noria

One of the most challenging aspects of designing a version control system customized for phylogenetic analyses is dealing with merges. A *merge* takes two versions A and B, of a single file and tries to incorporate the changes made in both versions in a new file, C. The goal of any merge procedure is to combine two versions of a file in a manner that makes sense to the user, with as little manual intervention as possible. This can be a very difficult problem, and different version control systems have attempted to tackle file merging in a variety of ways. The most popular of these merge techniques is the three-way merge, a sophisticated variation of which is implemented in Git.

a. The three-way merge

To illustrate how a three-way merge works, we start with a simple example of John and Annie now working collaboratively on a grocery list (Figure 40), using a standard version control system like Git. The three-way merge uses *three* versions of a file, A, B, and O, in order to construct a new version of the file C. States A and B are two different versions of the file located on different branches. O is the version of the file

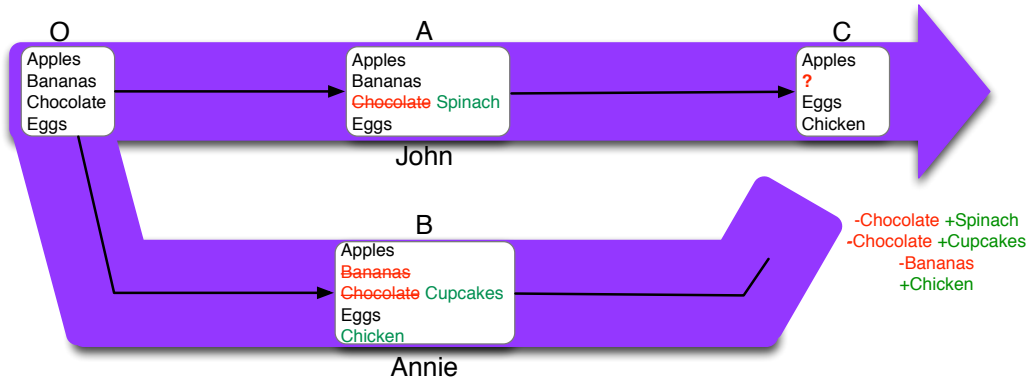


Fig. 41. Example of a three-way merge conflict on a grocery list of items.

that is the most recent common ancestor between A and B. To create C, the set of changes between O and A are computed, and the set of changes between O and B are computed. These changes are then applied to O to form file C.

In this example, O is the original list (and thus the most recent common ancestor), consisting of the four items of apples, bananas, chocolate and eggs. In step 1, John takes his version of the list (A) and decides that since he doesn't like chocolate, he should replace it with spinach. John's version of the list now contains apples, bananas, spinach, and eggs. Meanwhile in step 2, Annie decides she doesn't like bananas, and so she removes that item from the list. She also decides she wants to buy chicken, and so she appends that item to the end of the list. Annie's version of the file (B) now contains apples, chocolate, eggs, and chicken.

When John merges Annie's changes into his repository, Git computes the set of differences between the original list and John's version of the list (a deletion of chocolate and an addition of spinach), and the set of differences between the original list and Annie's version of the list (a deletion of bananas and an addition of chicken). The final version of the list (C) has replaced chocolate with spinach, deleted bananas and added chicken, to yield a final list of apples, spinach, eggs and chicken.

Sometimes a merge cannot be performed cleanly. In Figure 40 we were able to combine John and Annie's versions of the grocery list since changes were made to independent parts of the file. Consider the modified example shown in Figure 41. In this example, Annie also decides to delete chocolate from the grocery list and replace it with cupcakes. When Git computes the change set between O and A, it gets a deletion of chocolate followed immediately by an addition of spinach. When it computes the change set between O and B it sees a deletion of chocolate and an addition of cupcakes. Since both John and Annie made a change to the same item in the list, the 3-way merge algorithm will not be able to merge these two sets of changes. What should the second item be? Spinach or cupcakes? Since both choices are equally valid, this results in a *merge conflict*. Merge conflicts occur when independent changes occur to the same line in a file. When a merge conflict occurs, it is up to the user to manually *resolve*, or choose the version of the file that is correct and save the changes. For every version control system, the goal is to reduce conflicts, and thus manual intervention, whenever possible.

b. Merging trees

How does merging work in the context of trees? To perform a merge on a collection of trees, we can think of our trees as items in a list, where each tree is on a separate line. This is the structure of the PHYLIP tree format, and how trees are stored in NEXUS files. Let's revisit the merge scenario from Figure 38. John decides to merge the results of the analysis he received from Annie. In this case, John's original version of the trees serves as the most recent common ancestor (O), upon which John's version of the file (A) is used to merge in the changes from Annie's version of the file (B). To merge Annie's changes into his repository, the differences need to be computed between versions O and A and O and B. It is pretty clear that T_0 was replaced and

that the same topology was added to the end of the file by John. Thus, version C contains two topologies, with a pointer to the first. But, how does this exactly work?

c. Noria's merge algorithm

As previously mentioned, tree collections are stored internally in Noria as sets of TRZ files. It is not sufficient to use a standard version control system to track TRZ files, as a single change to a tree can affect multiple lines in the TRZ file. If the TRZ is interpreted purely as text, the standard merge drivers in standard version control systems will not be able to properly merge TRZ files. Thus, a custom merge driver is needed to ensure the proper merge of TRZ files.

Our custom merge driver for trees follows two rules that were largely inspired by those set for text in standard version control systems, and a third that accounts for robustness against Newick strings:

1. If two different sets of trees are independently modified in A and B, then those modifications should be reflected without conflict in C.

2. If the the same tree is modified differently by both A and B, then this would produce a conflict in C.

3. If a tree is modified in a way does that does not change its semantic meaning, then no change is recorded for this tree.

In order to apply these rules to TRZ files, it is necessary to be able to represent trees from the TRZ file uniquely and on a single line. We accomplish this by using our set operations and representing trees as bitstrings. In contrast to the set operations performed within TreeZip and discussed in Chapter VI, the Noria merge algorithm performs a set operation between *three* TRZ files (TreeZip performs set operations

between two TRZ files). To accomplish this, our custom merge driver first takes the union of all the bipartitions in O, A and B. This set of k bipartitions is then sorted, and a bitstring representation for each tree is created, where the i^{th} bit represents the i^{th} bipartition. We use the inverted index to assist in this process. For a particular tree, a bit will receive the value of '1' if the tree contains the bipartition, and '0' if it does not. This is almost identical to the process used by TreeZip's set operations. In the context of Noria, we run-length encode the bitstrings to yield unique string representations for the trees in O, A and B. In the last step, the three-way merge algorithm is applied to this set of unique representations to create file C. The TRZ file can then be easily rebuilt by transforming the unique string representations back into the encoded hash table form.

For the example in Figure 38, the set of seven unique bipartitions in the union of O, A, and B is $\{AB|CDEF$, $ABE|CDF$, $ABD|CEF$, $ABEF|CD$, $ABF|CDE$, $ABDF|CE$, $ABCD|EF\}$. Let each bipartition in this set be represented as by a number B_i, where B_i is the i^{th} bipartition in this set for $i = 0 \ldots 6$. Thus, B_1 is $ABE|CDF$. We can then represent our trees as follows in Figure 42. The trees that were modified in John's repository is shown first, followed by the trees modified in Annie's repository. The right-most column shows the unique, run-length encoded (RLE) string representation for each tree.

For example, since T_0 in John's repository contains bipartitions $AB|CDEF$, $ABE|CDF$ and $ABEF|CD$, the locations associated with these bipartitions get a value of 1, and the run-length encoded unique string for this tree is K2AB. By looking at the trees in RLE form, we can easily see that the trees John and Annie added are identical. This transformation allows us to easily utilize the three-way merge algorithm on trees (shown in Figure 43). The set of differences computed between O and A is the addition of the tree represented by `BABL3B`. The set of differences

		ABICDEF	ABEICDF	ABDICEF	ABEFICD	ABFICDE	ABDFICE	ABCDIEF	RLE
John's repository	T0	1	1	0	1	0	0	0	K2AB
	T1	1	0	0	0	1	1	0	BL3K2
	+T2	1	0	1	0	0	0	1	BABL3B
Annie's repository	-T3	1	1	0	1	0	0	0	K2AB
	+T4	1	0	1	0	0	0	1	BABL3B
	T5	1	0	0	0	1	1	0	BL3K2

Fig. 42. Identifying the unique trees in John and Annie's repository

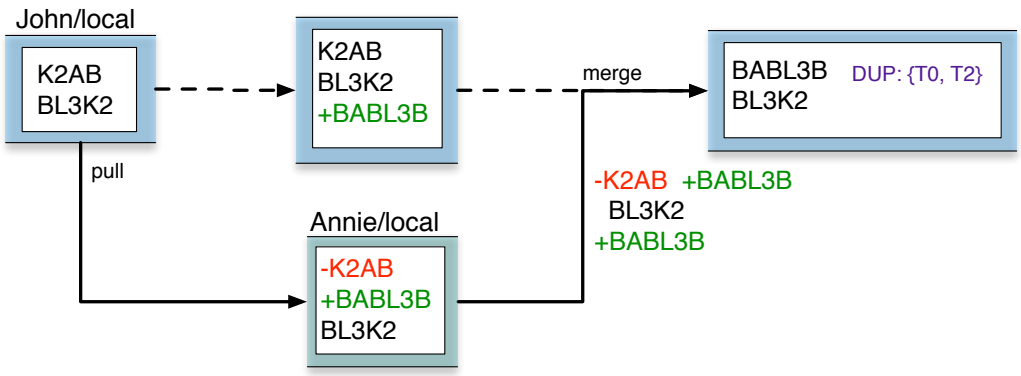


Fig. 43. Using unique representations of trees to perform a three-way merge.

computed between O and B is the deletion of the tree represented by `K2AB` and the addition of the tree represented by `BABL3B`. Since these changes do not conflict, we can produce a set of trees that do not conflict, and the hashing step we took prevents the storage of duplicates.

E. Implications of Noria

In addition to helping biologists manage and share their tree collections with each other, Noria allows them to directly analyze their data in the context of the system. Leveraging the algorithms described in previous chapters, users can compare tree collections and compute consensus trees. Since the underlying format of Noria is the TRZ file, these operations can be performed at speeds much quicker than if they were run on Newick-based input files.

The ramifications of such a system is huge. Through Noria, all the electronic data involved in a phylogenetic analysis are now accessible in one place. Since the Noria system leverages Git to be a fully distributed version control system, scientists can now share their analyses with other members of the biological community. As a result, Noria increases the experimental reproducibility of phylogenetic analysis. Users who have access to a biologist's Noria repository can now independently verify and explore the results of a phylogenetic analysis. While this is useful for error detection within the biological community, it also useful for helping novices familiarize themselves with a experimental process of a fellow scientist. This is especially useful in research labs, where a significant amount of time is usually needed to familiarize new students and researchers with the work being conducted. By allowing all the data and environment to be at the user's fingertips, Noria expedites this process.

Biologists also now have the opportunity to ask themselves questions about their

data and how it originated. For example, suppose an analysis produced several runs of trees. Did those runs actually coverage, topologically speaking? Noria allows scientist to compare their trees to each other, allowing for answers to these questions. Based on their results, a biologist may consider rerunning their analysis. Another potential application of comparing trees under Noria is to understand the impact a particular parameter has on the trees produced. For a particular dataset, what degree of difference does the model choice make? How does performing a small correction on the multiple sequence alignment impact the quality (or structure) of trees produced? By gaining a greater understanding of their data and the algorithms used to analyze them, biologists can fine tune how they perform phylogenetic analysis, potentially saving time in the process and producing more robust phylogenies.

For computer scientists, Noria illustrates the ability to create domain-specific version control systems for specific scientific communities. Noria's strength lies in its ability to leverage tree-like data. By utilizing TreeZip, Noria eliminates the redundancy that is contained in phylogenetic tree collections, and creates a smaller, queryable form that is tracked using version control. Noria then shows how a custom three-way merge algorithm can be created to uniquely identify each tree in our collection and combine the results appropriately. In a broader sense, these algorithms and concepts can be extended to other tree-like data beyond the realm of phylogenetics. For example, we believe that this work can be used to help create approaches for better managing Web 2.0 data such as XML files. Since we believe that version control is a fundamental right (especially for researchers), we hope Noria inspires others to create domain-specific systems that can take advantage of the knowledge unique to their particular communities.

F. Summary

In this section, we presented Noria, a new version control system that is designed for phylogenetic analysis. Noria allows users to keep track of their phylogenetic analyses locally on their system, and also share the results of their analyses with others. The *novelty* of Noria centers on how we use a modified version of the TRZ file to uniquely track trees and their bipartitions. Unlike standard version control systems, Noria is able to differentiate tree collections from unstructured text, and store trees uniquely. This reduces the potential for storing duplicate information, and removes un-necessary conflicts. A custom merge driver specifically for TRZ files allows users to perform merges. Noria also goes beyond a standard version control system, as it allows users to actively mine new knowledge from their data in the context of the system. This allows users to learn more about their data and the algorithms that produced them.

For biologists, Noria is a system that allows them to seamlessly manage and share their phylogenetic analyses with each other without interrupting their main workflow. Next, since Noria allows scientists to easily save their tree collections, there will be a significant impact on the ability of other biologists to verify the results of a phylogenetic analyses. Noria contains all the information for conducting a phylogenetic analysis — the data, the model, the trees, and all post-analysis work.

The fast algorithms implemented in the context of Noria allow biologists who have access to an author's public repository to quickly verify computational results, and explore the experimental process of the author. This will help lend greater credence to certain analyses over others, and quicken the detection of errors. As a positive side effect for computational biologists, the greater availability of large tree collections will allow for the generation of better algorithms to process large collections of trees, thus perpetuating the aims of this work.

For computer scientists, Noria highlights the need and potential to create custom version control systems tailored to the need of specific scientific communities. Furthermore, since analysis is a central part of computational science, systems should be created that allow scientist to study their data in the context of the system. Reproducible research is a hot-button issue today [109, 110, 111, 112], and the creation of systems like Noria will help increase the level of reproducibility within computational science.

Future versions of Noria will be more feature-rich, and interact with a central repository hub (similar to GitHub) that will allow scientists to open their analyses up to the general public, discuss results, and collaborate with greater ease. The code for Noria will be open sourced, to expedite the process of development and help make the system more robust. We strongly believe that Noria will help increase the reproducibility of modern phylogenetic analysis, increase the level of collaboration among biologists, shed light on the behavior of the algorithms involved in the process, and ultimately help in the construction of more robust phylogenies.

CHAPTER VIII

CONCLUSION AND FUTURE WORK

The goal of this research is to design new approaches that enable biologists to seamlessly manage their large tree collections and share their results with the phylogenetic community. Currently, these large collections are often discarded and are rarely shared, making it difficult to reproduce results. The central assumption is that the information discarded has less value than the information contained in a single consensus tree, which is the most popular approach to summarize large tree collections.

We have shown this assumption to be false. We illustrate how a Robinson-Foulds (RF) distance matrix is capable of detecting underlying diversity in a set of equally scoring trees[10, 11]. This suggests that score alone is not sufficient to evaluate a collection, and that topological methods should be used. Diversity in a collection of trees also imply greater heuristic value, as a slower heuristic may be able to detect different, but equally valuable trees in the process of search. In two case studies [12], we also show how RF matrices can be used for convergence detection. This suggests there is great value in retaining tree collections. Through an extensive literature search, we also demonstrated that the number of trees produced by a phylogenetic analysis is rapidly increasing, thus leading to massive information loss. Thus, novel methods are needed to allow biologists to easily retain and leverage their large tree collections.

We developed four novel algorithms that utilize universal hashing to efficiently compare, store, and share large collections of trees. Universal hashing enables us to capture the set of unique bipartitions contained in a collection of trees in $O(nt)$ time. We augmented a previously established Monte Carlo hashing algorithm to be a Las Vegas approach. Next, we introduced a new data structure, the compact table, to

efficiently represent tree information and allow for fast operations. We also presented an efficient way to represent trees as bitstrings, allowing trees to be identified uniquely and compared in novel ways. Lastly, we demonstrate how hashing can be applied to detect relationships in heterogeneous collections of trees. This opens up a whole new range of possibilities for exploring heterogeneous data.

Our algorithms for comparing large collections of evolutionary trees are the fastest approaches available today. MrsRF highlights how we can leverage the MapReduce paradigm to create large RF matrices in real-time [12]. When run in parallel, MrsRF is capable of comparing $33,306$ trees in approximately 35 seconds. Our Phlash algorithm allows us to look beyond Robinson-Foulds distance and compare trees in a multitude of ways.

Our TreeZip algorithm [15, 16] leverages universal hashing to yield the most efficient representation for a collection of phylogenetic trees. The TreeZip compressed (TRZ) file occupies at least 94% (74%) less space than a Newick file for unweighted (weighted) collections of trees. The TRZ file's textual format allow for tree operations to be performed directly, while retaining very high space savings. We have extended our TreeZip algorithm to compress heterogeneous collections of trees, allowing for universal applicability to tree collections.

Motivated by our our fast hash-based algorithms, we designed Noria to help life scientists manage their large phylogenetic analyses. Unlike standard version control system like Subversion [18] and Git [19], Noria is designed specifically for phylogenetic trees. Standard version control systems will not detect when a tree has changed leading to greater storage requirements. The TRZ file forms the basis of Noria, and allows us to store our tree collections uniquely and efficiently. A central part of the Noria system is the presence of a novel three-way merge algorithm that allows users to merge TRZ files.

Our work allows biologists to share large their tree collections, and to produce a record of the scientific progress. This in turn promotes a greater degree of collaboration in the scientific community, allows for greater reproducibility, and ultimately helps scientist create more robust phylogenies. Our work makes tree collections valuable and easy to analyze and share. Furthermore, all of our work is open-source and free to download by the community, allowing for rapid adoption.

Our work also makes numerous contributions to the computer science community. The fast and efficient algorithms presented in this work represent the state of the art for computational phylogenetics. Furthermore, our algorithms illustrate how universal hashing can be used for domain-based compression of scientific datasets. We also show how universal hashing can be used for set operations, and the construction of three-way merge algorithms for distributed version control systems. Our work spans several domains, from parallel computing and experimental algorithmics, to compression and distributed version control. Lastly, our work can be leveraged to exploit other tree-like data, and can be used for the creation of collaborative systems in other scientific domains.

Future work will concentrate on developing Noria into a more feature-rich system that can be released for general use in the biological community. Key to this is further improving the time to operate on weighted phylogenetic trees. We also plan on exploring new ways to improve the quality of branch length compression in TreeZip. Lastly, we want to explore how the data structures and algorithms described in this work can be leveraged to compare, store and share other types of tree-like data, such as XML files.

REFERENCES

[1] B. W. Davis, G. Li, and W. J. Murphy, "Supermatrix and species tree methods resolve phylogenetic relationships within the big cats, panthera (carnivora: Felidae)," *Molecular Phylogenetics and Evolution*, vol. 56, no. 1, pp. 64–76, 2010.

[2] D. I. Scaduto, J. M. Brown, W. C. Haaland, D. J. Zwickl, D. M. Hillis, and M. L. Metzker, "Source identification in two criminal cases using phylogenetic analysis of HIV-1 DNA sequences," *Proceedings of the National Academy of the Sciences (PNAS)*, vol. 107, no. 50, pp. 21 242–21 247.

[3] M. L. Metzker, D. P. Mindell, X.-M. Liu, R. G. Ptak, R. A. Gibbs, and D. M. Hillis, "Molecular evidence of HIV-1 transmission in a criminal case," *Proceedings of the National Academy of the Sciences (PNAS)*, vol. 99, no. 22, pp. 14 292–14 297, 2002.

[4] C. J. Birch, R. F. McCaw, D. M. Bulach, P. A. Revill, J. T. Carter, J. Tomnay, B. Hatch, T. V. Middleton, D. Chibo, M. G. Catton, J. L. Pankhurst, A. M. Breschkin, S. A. Locarnini, and S. Bowden, "Molecular analysis of human immunodeficiency virus strains associated with a case of criminal transmission of the virus," *The Journal of Infectious Diseases*, vol. 182, no. 3, pp. 941–944, 2000.

[5] C.-Y. Ou, C. A. Ciesielski, G. Myers, C. I. Bandea, C.-C. Luo, B. T. M. Korber, J. I. Mullins, G. Schochetman, R. L. Berkelman, A. N. Economou, J. J. Witte, L. J. Furman, G. A. Satten, K. A. MacInnes, J. W. Curran, H. W. Jaffe, L. I. Group, and E. I. Group, "Molecular epidemiology of HIV transmission in a dental practice," *Science*, vol. 256, no. 5060, pp. 1165–1171, 1992.

[6] P. G. Szekeres, A. I. Muir, L. D. Spinage, J. E. Miller, S. I. Butler, A. Smith, G. I. Rennie, P. R. Murdock, L. R. Fitzgerald, H. ling Wu, L. J. McMillan, S. Guerrera, L. Vawter, N. A. Elshourbagy, M. J. L., B. D. J., S. Wilson, and C. J. K., "Neuromedin U is a potent agonist at the orphan G protein-coupled receptor FM3," *Journal of Biological Chemistry*, vol. 275, no. 27, pp. 20 247–20 250, 2000.

[7] J. K. Chambers, L. E. Macdonald, H. M. Sarau, R. S. Ames, K. Freeman, J. J. Foley, Y. Zhu, M. M. McLaughlin, P. Murdock, L. McMillan, J. Trill, A. Swift, N. Aiyar, P. Taylor, L. Vawter, S. Naheed, P. Szekeres, G. Hervieu, C. Scott, J. M. Watson, A. J. Murphy, E. Duzic, C. Klein, D. J. Bergsma, S. Wilson, and G. P. Livi, "AG protein-coupled receptor for UDP-glucose," *Journal of Biological Chemistry*, vol. 275, no. 15, pp. 10 767–10 771, 2000.

[8] M. Spencer, E. A. Davidson, A. C. Barbrook, and C. J. Howe, "Phylogenetics of artificial manuscripts," *Journal of Theoretical Biology*, vol. 227, no. 4, pp. 503–511, 2004.

[9] A. C. Barbrook, C. J. Howe, N. Blake, and P. Robinson, "The phylogeny of the Canterbury Tales," *Nature*, vol. 394, p. 839, 1998.

[10] S.-J. Sul, S. J. Matthews, and T. L. Williams, "Using tree diversity to compare phylogenetic heuristics," *BMC Bioinformatics*, vol. 10, no. Suppl 4, p. S3, 2009.

[11] S.-J. Sul, S. Matthews, and T. L. Williams, "New approaches to compare phylogenetic search heuristics," in *IEEE International Conference on Bioinformatics and Biomedicine (BIBM'08)*, 2008, pp. 239–245.

[12] S. J. Matthews and T. L. Williams, "MrsRF: an efficient mapreduce algorithm

for analyzing large collections of evolutionary trees," *BMC Bioinformatics*, vol. 11, no. Suppl 1, p. S15, 2010.

[13] J. P. Huelsenbeck and F. Ronquist, "MRBAYES: Bayesian inference of phylogenetic trees," *Bioinformatics*, vol. 17, no. 8, pp. 754–755, 2001.

[14] S.-J. Sul and T. L. Williams, "An experimental analysis of robinson-foulds distance matrix algorithms," in *European Symposium of Algorithms (ESA'08)*, ser. Lecture Notes in Computer Science, vol. 5193. Springer-Verlag, 2008, pp. 793–804.

[15] S. J. Matthews and T. L. Williams, "An efficient and extensible approach for compressing phylogenetic trees," *BMC Bioinformatics*, vol. 12, no. Suppl 10, p. S16, 2011.

[16] S. J. Matthews, S.-J. Sul, and T. L. Williams, "A novel approach for compressing phylogenetic trees," in *Bioinformatics Research and Applications*, ser. Lecture Notes in Computer Science, vol. 6053. Springer-Verlag, 2010, pp. 113–124.

[17] M. Mackall, "Mercurial SCM," Internet Website, last accessed, July 2010, http://mercurial.selenic.com/.

[18] B. Collins-Sussman, B. Fitzpatrick, and C. Pilato, *Version control with subversion*. O'Reilly Media, Inc., 2004.

[19] L. Torvalds and J. C. Hamano, "Git - fast version control system," Internet Website, last accessed, July 2010, http://git-scm.com/.

[20] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, "Extensible markup language (XML)," *World Wide Web Journal*, vol. 2, no. 4, pp.

27–66, 1997.

[21] M. J. McInerney, C. G. Struchtemeyer, J. Sieber, H. Mouttaki, A. J. Stams, B. Schink, L. Rohlin, and R. P. Gunsalus, "Physiology, ecology, phylogeny, and genomics of microorganisms capable of syntrophic metabolism," *Annals of the New York Academy of Sciences*, vol. 1125, no. 1, pp. 58–72, 2008.

[22] D. R. Brooks and D. A. McLennan, *Phylogeny, ecology, and behavior: a research program in comparative biology.* University of Chicago press, 1991.

[23] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and D. L. Wheeler, "GenBank," *Nucleic Acids Research*, vol. 33, no. suppl 1, pp. D34–D38, 2005.

[24] H. Miller, C. Norton, and I. Sarkar, "GenBank and PubMed: How connected are they?" *BMC Research Notes*, vol. 2, no. 1, p. 101, 2009.

[25] R. Chenna, H. Sugawara, T. Koike, R. Lopez, T. J. Gibson, D. G. Higgins, and J. D. Thompson, "Multiple sequence alignment with Clustal series of programs," *Nucleic Acids Research*, vol. 31, no. 13, pp. 3497–3500, 2003.

[26] A. Varón, L. Vinh, and W. Wheeler, "POY version 4: Phylogenetic analysis using dynamic homologies," *Cladistics*, vol. 26, no. 1, pp. 72–85, 2010.

[27] K. Liu, S. Raghavan, S. Nelesen, C. R. Linder, and T. Warnow, "Rapid and accurate large-scale coestimation of sequence alignments and phylogenetic trees," *Science*, vol. 324, no. 5934, pp. 1561–1564, June 2009.

[28] D. L. Swofford, *PAUP*: Phylogenetic Analysis Using Parsimony (and Other Methods).* Sinauer Associates, 2002.

[29] P. A. Goloboff, J. S. Farris, and K. C. Nixon, "TNT, a free program for phylogenetic analysis," *Cladistics*, vol. 24, no. 5, pp. 774–786, 2008.

[30] M. Salemi and A.-M. Vandamme, *The Phylogenetic Handbook : A Practical Approach to DNA and Protein Phylogeny.* Cambridge University Press, 2003.

[31] S. Guindon and O. Gascuel, "A simple, fast, and accurate algorithm to estimate large phylogenies by maximum likelihood," *Systematic Biology*, vol. 52, no. 5, pp. 696–704, 2003.

[32] A. Stamatakis, T. Ludwig, and H. Meier, "RAxML: A fast program for maximum likelihood-based inference of large phylogenetic trees," *Bioinformatics*, vol. 1, no. 1, pp. 1–8, 2004.

[33] W. Piel, M. Donoghue, and M. Sanderson, "TreeBASE: A database of phylogenetic information," in *Proceedings of the 2nd International Workshop of Species*, 2000, pp. 41–47.

[34] U. Roshan, B. M. E. Moret, T. L. Williams, and T. Warnow, "Rec-I-DCM3: a fast algorithmic techniques for reconstructing large phylogenetic trees," in *Proceedings of the IEEE Computer Society Bioinformatics Conference (CSB'04).* IEEE Press, 2004, pp. 98–109.

[35] D. Sikes and D. O. Lewis, "PAUPRat: PAUP implementation of the parsimony ratchet," Internet Website, last accessed, January 2012, http://users.iab.uaf.edu/∼derek_sikes/software2.htm.

[36] K. Rice, M. Donoghue, and R. Olmstead, "Analyzing large datasets: rbcL 500 revisited," *Systematic Biology*, vol. 46, no. 3, pp. 554–563, 1997.

[37] D. E. Soltis, M. A. Gitzendanner, and P. S. Soltis, "A 567-taxon data set for angiosperms: The challenges posed by bayesian analyses of large data sets," *International Journal of Plant Sciences*, vol. 168, no. 2, pp. 137–157, 2007.

[38] M. Sanderson, B. Baldwin, G. Bharathan, C. Campbell, D. Ferguson, J. Porter, C. V. Dohlen, M. Wojciechowski, and M. Donoghue, "The growth of phylogenetic information and the need for a phylogenetic database," *Systematic Biology*, vol. 42, no. 4, pp. 562–568, 1993.

[39] M. J. Sanderson, M. J. Donoghue, W. Piel, and T. Eriksson, "TreeBASE: A prototype database of phylogenetic analyses and an interactive tool for browsing the phylogeny of life," *American Journal of Botany*, vol. 81, no. 6, p. 183, 1994.

[40] W. Piel, "TreeBASE: A database of phylogenetic knowledge," Internet Website, last accessed, January 2012, http://www.treebase.org/.

[41] B. Wrobel, "Statistical measures of uncertainty for branches in phylogenetic trees inferred from molecular sequences by using model-based methods," *Journal of Applied Genetics*, vol. 49, no. 1, pp. 49–67, 2008.

[42] B. Kolaczkowski and J. W. Thornton, "Effects of branch length uncertainty on bayesian posterior probabilities for phylogenetic hypotheses," *Molecular Biology and Evolution*, vol. 24, no. 9, pp. 2108–2118, July 2007.

[43] Z. Yang and B. Rannala, "Branch-length prior influences bayesian posterior probability of phylogeny," *Systematic Biology*, vol. 54, no. 3, pp. 455–470, June 2005.

[44] P. A. Goloboff, "Character optimization and calculation of tree lengths," *Cladistics*, vol. 9, pp. 433–436, 1993.

[45] P. Goloboff, "Methods for faster parsimony analysis," *Cladistics*, vol. 12, no. 3, pp. 199–220, 1996.

[46] A. J. Drummond, "BEAST: Bayesian evolutionary analysis by sampling trees," *BMC Evolutionary Biology*, vol. 7, no. 1, p. 214, November 2007.

[47] L. A. Lewis and P. O. Lewis, "Unearthing the molecular phylodiversity of desert soil green algae (chlorophyta)," *Systematic Biology*, vol. 54, no. 6, pp. 936–947, 2005.

[48] A. D. Molin, S. Matthews, S.-J. Sul, J. Munro, J. B. Woolley, J. M. Heraty, and T. L. Williams, "Large data sets, large sets of trees, and how many brains? — Visualization and comparison of phylogenetic hypotheses inferred from rdna in chalcidoidea (hymenoptera)," Entomological Society of America (ESA) Annual Meeting: Student Competition for the Presidents Prize (poster), December 2009.

[49] D. M. Hillis, T. A. Heath, and K. S. John, "Analysis and visualization of tree space," *Systematic Biology*, vol. 54, no. 3, pp. 471–482, 2005.

[50] M. A. Steel and D. Penny, "Distributions of tree comparision metrics — some new results," *Systematic Biology*, vol. 42, no. 2, pp. 126–141, 1993.

[51] M. Stissing, T. Mailund, C. N. S. Pedersen, G. S. Brodal, and R. Fagerberg, "Computing the all-pairs quartet distance on a set of evolutionary trees," in *The Fifth Asia Pacific Bioinformatics Conference (APBC'07)*, 2007, pp. 91–100.

[52] G. S. Brodal, R. Fagerberg, and C. N. S. Pedersen, "Computing the quartet distance between evolutionary trees in time $O(n \log^2 n)$," *Algorithms and Computation*, vol. 2223, pp. 731–742, 2001.

[53] W.-K. Hon, M.-Y. Kao, and T.-W. Lam, "Improved phylogeny comparisons: Non-shared edges, nearest neighbor interchanges, and subtree transfers," in

Proceedings of the 11th International Conference on Algorithms and Computation (ISAAC'00). Springer-Verlag, 2000, pp. 527–538.

[54] B. DasGupta, X. He, T. Jiang, M. Li, J. Tromp, and L. Zhang, "On computing the nearest neighbor interchange distance," in *Proceedings of the DIMACS Workshop on Discrete Problems with Medical Applications*. American Mathematical Society, 1997, pp. 125–143.

[55] M. Bourque, "Arbres de steiner et reseaux dont certains sommets sont a localisation variable," Ph.D. dissertation, Université de Montréal, 1978.

[56] D. F. Robinson and L. R. Foulds, "Comparison of phylogenetic trees," *Mathematical Biosciences*, vol. 53, no. 1-2, pp. 131–147, 1981.

[57] W. H. E. Day, "Optimal algorithms for comparing trees with labeled leaves," *Journal Of Classification*, vol. 2, no. 1, pp. 7–28, 1985.

[58] J. Felsenstein, "Phylogenetic inference package (PHYLIP), version 3.2," *Cladistics*, vol. 5, pp. 164–166, 1989.

[59] N. Pattengale, E. Gottlieb, and B. Moret, "Efficiently computing the Robinson-Foulds metric," *Journal of Computational Biology*, vol. 14, no. 6, pp. 724–735, 2007.

[60] C. Than, D. Ruths, and L. Nakhleh, "PhyloNet: a software package for analyzing and reconstructing reticulate evolutionary relationships," *BMC Bioinformatics*, vol. 9, no. 1, p. 322, 2008.

[61] S. Sul and T. L. Williams, "A randomized algorithm for comparing sets of phylogenetic trees," in *Proceedings of the Fifth Asia Pacific Bioinformatics Conference (APBC'07)*, 2007, pp. 121–130.

[62] S.-J. Sul and T. L. Williams, "HashRF: a fast algorithm for computing the Robinson-Foulds distance matrix," Department of Computer Science, Texas A&M University, Tech. Rep. TR-CS-2008-6-1, 2008.

[63] M. Kuhner and J. Felsenstein, "A simulation comparison of phylogeny algorithms under equal and unequal evolutionary rates." *Molecular Biology and Evolution*, vol. 11, no. 3, pp. 459–468, 1994.

[64] J. Felsenstein, "TreeDist," Internet Website, last accessed, December 2011, http://computing.bio.cam.ac.uk/local/doc/phylip/treedist.html.

[65] C. Stockham, L. S. Wang, and T. Warnow, "Statistically based postprocessing of phylogenetic analysis by clustering," in *Proceedings of the 10th International Conference on Intelligent Systems for Molecular Biology (ISMB'02)*, 2002, pp. 285–293.

[66] W. Maddison and D. Maddison, "Mesquite: a modular system for evolutionary analyses, version 2.75," Internet Website, last accessed, December 2011, http://mesquiteproject.org.

[67] M. Han and C. Zmasek, "phyloXML: XML for evolutionary biology and comparative genomics," *BMC Bioinformatics*, vol. 10, no. 1, p. 356, 2009.

[68] M. Settings, "NEXML - phylogenetic data as XML," Internet Website, last accessed, July 2010, http://www.nexml.org/.

[69] J. Felsenstein, "The Newick format," Internet Website, last accessed, September 2009, http://evolution.genetics.washington.edu/phylip/newicktree.html.

[70] D. R. Maddison, D. L. Swofford, and W. P. Maddison, "NEXUS: an extensible file format for systematic information," *Systematic Biology*, vol. 46, no. 4, p.

590, 1997.

[71] R. S. Boyer, W. A. Hunt Jr., and S. Nelesen, "A compressed format for collections of phylogenetic trees and improved consensus performance," in *Workshop on Algorithms in Bioinformatics (WABI'05)*, ser. Lecture Notes in Computer Science, vol. 3692. Springer-Verlag, 2005, pp. 353–364.

[72] R. Boyer, W. A. Hunt Jr., and S. Nelesen, "A compressed format for collections of phylogenetic trees and improved consensus performance," Department of Computer Sciences, The University of Texas at Austin, Tech. Rep. TR-05-12, 2005.

[73] E. Goto, "Monocopy and associative algorithms in extended lisp," University of Tokyo, Tech. Rep. TR 74-03, 1974.

[74] DropBox, Inc., "DropBox — simplify your life," Internet Website, last accessed, December 2011, http://www.dropbox.com/.

[75] H. C. White, S. Carrier, A. Thompson, J. Greenberg, and R. Scherle, "The Dryad data repository: a Singapore framework metadata architecture in a DSpace environment," in *Proceedings of the 2008 International Conference on Dublin Core and Metadata Applications (DCMI'08)*, 2008, pp. 157–162.

[76] N. Amenta, F. Clarke, and K. S. John, "A linear-time majority tree algorithm," in *Workshop on Algorithms in Bioinformatics (WABI'03)*, ser. Lecture Notes in Computer Science, vol. 2168. Springer-Verlag, 2003, pp. 216–227.

[77] S.-J. Sul, G. Brammer, and T. L. Williams, "Efficiently computing arbitrarily-sized robinson-foulds distance matrices," in *Workshop on Algorithms in Bioinformatics (WABI'08)*, ser. Lecture Notes in Computer Science, vol. 5251.

Springer-Verlag, 2008, pp. 123–134.

[78] S.-J. Sul and T. L. Williams, "An experimental analysis of consensus tree algorithms for large-scale tree collections," in *Proceedings of the 5th International Symposium on Bioinformatics Research and Applications (ISBRA'09)*. Springer-Verlag, 2009, pp. 100–111.

[79] S. J. Matthews and T. L. Williams, "MrsRF: Project website," Internet Website, last accessed, March 2012, http://mrsrf.googlecode.com.

[80] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the Association of Computing Machinery*, vol. 51, no. 1, pp. 107–113, January 2008.

[81] M. C. Schatz, "Cloudburst: Highly sensitive read mapping with mapreduce," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, July 2009.

[82] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *IEEE 13th International Symposium on High Performance Computer Architecture (HPCA'07)*, Feb. 2007, pp. 13–24.

[83] A. Bialecki, M. Cafarella, D. Cutting, and O. OMalley, "Hadoop: a framework for running applications on large clusters built of commodity hardware," Internet Website, last accessed, May 2009, http://hadoop.apache.org/core/index.html.

[84] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of

a next generation MPI implementation," in *Proceedings of the 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[85] S. S. Choi, S. H. Cha, and C. Tappert, "A survey of binary similarity and distance measures," *Journal on Systemics, Cybernetics and Informatics*, vol. 8, no. 1, pp. 43–48, 2010.

[86] N. Salim, J. Holliday, and P. Willett, "Combination of fingerprint-based similarity coefficients using data fusion," *Journal of Chemical Information and Computer Sciences*, vol. 43, no. 2, pp. 435–442, March 2003.

[87] D. P. Faith, "Asymmetric binary similarity measures," *Oecologia*, vol. 57, no. 3, pp. 287–290, 1983.

[88] D. Ellis, J. Furner-Hines, and P. Willett, "Measuring the degree of similarity between objects in text retrieval systems," *Perspectives in Information Management*, vol. 3, no. 2, pp. 128–149, 1993.

[89] J. H. Ward Jr., "Hierarchical grouping to optimize an objective function," *Journal of the American Statistical Association*, vol. 58, no. 301, pp. 236–244, 1963.

[90] N. Mantel, "The detection of disease clustering and a generalized regression approach," *Cancer Research*, vol. 27, no. 2, pp. 209–220, 1967.

[91] S. J. Matthews and T. L. Williams, "TreeZip: Project website," Internet Website, last accessed, March 2012, http://treezip.googlecode.com.

[92] H. E. Williams and J. Zobel, "Compressing integers for fast file access," *The Computer Journal*, vol. 42, no. 3, pp. 193–201, 1999.

[93] J. E. Janecka, W. Miller, T. H. Pringle, F. Wiens, A. Zitzmann, K. M. Helgen, M. S. Springer, and W. J. Murphy, "Molecular and genomic data identify the closest living relative of primates," *Science*, vol. 318, pp. 792–794, 2007.

[94] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part i," *Communications of the ACM*, vol. 3, no. 4, pp. 184–195, 1960.

[95] J. Loeliger, *Version control with Git*. O'Reilly Media, 2009.

[96] A. L. Glasser, "The evolution of a source code control system," in *Proceedings of the software quality assurance workshop on Functional and performance issues*. Association of Computing Machinery, 1978, pp. 122–125.

[97] W. F. Tichy, "RCS — a system for version control," *Software: Practice and Experience*, vol. 15, no. 7, pp. 637–654, 1985.

[98] J. MacDonald, P. Hilfinger, and L. Semenzato, "PRCS: The project revision control system," in *System Configuration Management*, ser. Lecture Notes in Computer Science. Springer-Verlag, 1998, vol. 1439, pp. 33–45.

[99] D. Grune, "Concurrent versions system, a method for independent cooperation," Vrije University, Amsterdam, Tech. Rep. IR-114, 1986.

[100] OpenBSD, "OpenCVS," Internet Website, last accessed, December 2011, http://www.opencvs.org/.

[101] T. Lord, "The Gnu Arch distributed revision control system," Internet Website, last accessed, December 2011, http://www.gnu.org/software/gnu-arch.

[102] The Monotone Team, "Monotone," Internet Website, last accessed, December 2011, http://www.monotone.ca/.

[103] V. Henson and J. Garzik, "Bitkeeper for kernel developers," in *Ottawa Linux Symposium*, 2002, p. 197.

[104] A. Dabbs and J. Carucci, *Adobe Creative Suite 2 Integration: Photoshop, Illustrator, InDesign, GoLive, Acrobat, Bridge and Version Cue.* Focal Press, 2005.

[105] PixelNovel Ltd, "PixelNovel: Version control for designers," Internet Website, last accessed, December 2011, http://pixelnovel.com/.

[106] Black Pixel, "Compare files with Kaleidoscope," Internet Website, last accessed, December 2011, http://www.kaleidoscopeapp.com/.

[107] "libgit2: a linkable library for git," Internet Website, last accessed, March 2012, http://libgit2.github.com/.

[108] P. O. Lewis, "NCL: a C++ class library for interpreting data files in NEXUS format," *Bioinformatics*, vol. 19, no. 17, pp. 2330–2331, 2003.

[109] K. A. Baggerly and D. A. Berry, "Reproducible research," *AMSTAT NEWS*, January 2011.

[110] Z. Merali, "Error: why scientific programming does not compute," *Nature*, vol. 467, p. 775, October 2010.

[111] Yale Law School Roundtable on Data and Code Sharing, "Reproducible research: Addressing the need for data and code sharing in computational science," *Computing in Science and Engineering*, vol. 12, no. 5, pp. 8–13, 2010.

[112] D. L. Donoho, A. Maleki, I. U. Rahman, M. Shahram, and V. Stodden, "Reproducible research in computational harmonic analysis," *Computing in Science and Engineering*, vol. 11, no. 1, pp. 8–18, 2009.

VITA

Suzanne Jude Matthews received her Ph.D. in Computer Science from the department of Computer Science & Engineering at Texas A&M University in May 2012. Her thesis advisor was Dr. Tiffani L. Williams. In August 2011, she was recognized as a Texas A&M University Dissertation Fellow for academic year 2011-2012. She received her M.S. in Computer Science in May 2008 from Rensselaer Polytechnic Institute, where she completed her masters thesis titled "Visualizing pathways: An exploration of the protein unfolding process". During her graduate career at Rensselaer, she was recognized as a Master Teaching Fellow for academic year 2007-2008. She also received her B.S. in Computer Science from Rensselaer in 2006. As an undergraduate, she participated in the Computer Research Association Distributed Mentoring Program (CRA-W DMP) under the direction of Dr. Tiffani L. Williams on a project titled "On the Importance of Starting Trees". Her research experience is in computational biology, particularly high performance phylogenetic analysis, version control systems and protein folding.

More information about Suzanne Matthews' research and publications may be found at http://students.cse.tamu.edu/sjm. She may be reached at 301 Harvey R. Bright Bldg. Texas A&M University - TAMU 3112, College Station, TX 77843-3112. Her e-mail address is sjm@cse.tamu.edu.