

FAST HASH-BASED ALGORITHMS FOR ANALYZING
LARGE COLLECTIONS OF EVOLUTIONARY TREES

A Dissertation

by

SEUNG JIN SUL

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

December 2009

Major Subject: Computer Science

FAST HASH-BASED ALGORITHMS FOR ANALYZING
LARGE COLLECTIONS OF EVOLUTIONARY TREES

A Dissertation

by

SEUNG JIN SUL

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
	Ricardo Gutierrez-Osuna
	James B. Woolley
Head of Department,	Valerie E. Taylor

December 2009

Major Subject: Computer Science

ABSTRACT

Fast Hash-based Algorithms for Analyzing Large Collections of Evolutionary Trees.

(December 2009)

Seung Jin Sul, B.S., Dongguk University;

M.S., Dongguk University

Chair of Advisory Committee: Dr. Tiffani L. Williams

Phylogenetic analysis can produce easily tens of thousands of equally plausible evolutionary trees. Consensus trees and topological distance matrices are often used to summarize the evolutionary relationships among the trees of interest. However, current approaches are not designed to analyze very large tree collections. In this thesis, we present two fast algorithms— HashCS and HashRF —for analyzing large collections of evolutionary trees based on a novel hash table data structure, which provides a convenient and fast approach to store and access the bipartition information collected from the tree collections.

Our HashCS algorithm is a fast $O(nt)$ technique for constructing consensus trees, where n is the number of taxa and t is the number of trees. By reprocessing the bipartition information in our hash table, HashCS constructs strict and majority consensus trees. In addition to a consensus algorithm, we design a fast topological distance algorithm called HashRF to compute the $t \times t$ Robinson-Foulds distance matrix, which requires $O(nt^2)$ running time. A RF distance matrix provides plenty of data-mining opportunities to help researchers understand the evolutionary relationships contained in their collection of trees. We also introduce a series of extensions based on HashRF to provide researchers with more convenient set of tools for analyzing their trees. We provide extensive experimentation regarding the practical performance of our hash-

based algorithms across a diverse collection of biological and artificial trees. Our results show that both algorithms easily outperform existing consensus and RF matrix implementations. For example, on our biological trees, HashCS and HashRF are 1.8 and 100 times faster than PAUP*, respectively.

We show two real-world applications of our fast hashing algorithms: (i) comparing phylogenetic heuristic implementations, and (ii) clustering and visualizing trees. In our first application, we design novel methods to compare the PaupRat and Rec-I-DCM3, two popular phylogenetic heuristics that use the Maximum Parsimony criterion, and show that RF distances are more effective than parsimony scores at identifying heterogeneity within a collection of trees. In our second application, we empirically show how to determine the distinct clusters of trees within large tree collections. We use two different techniques to identify distinct tree groups. Both techniques show that partitioning the trees into distinct groups and summarizing each group separately is a better representation of the data. Additional benefits of our approach are better consensus trees as well as insightful information regarding the convergence behavior of phylogenetic heuristics.

Our fast hash-based algorithms provide scientists with a very powerful tools for analyzing the relationships within their large phylogenetic tree collections in new and exciting ways. Our work has many opportunities for future work including detecting convergence and designing better heuristics. Furthermore, our hash tables have lots of potential future extensions. For example, we can also use our novel hashing structure to design algorithms for computing other distance metrics such as Nearest Neighbor Interchange (NNI), Subtree Pruning and Regrafting (SPR), and Tree Bisection and Reconnection (TBR) distances.

To my family

ACKNOWLEDGMENTS

I am sincerely grateful to my advisor, Dr. Tiffani L. Williams, for allowing me to conduct research with her. I am constantly amazed by her extraordinary ability in transforming seeming unsolvable problems into a tractable form, her infinite knowledge on subject matters, and her relentless attention to detail. Her exceptional commitment to research and strong demand for excellence have guided me this far. I am truly grateful for her insightful advice, encouragement, and constant motivation throughout this work.

I would also like to thank Dr. Nancy M. Amato, Dr. Ricardo Gutierrez-Osuna, and Dr. James B. Woolley for their service on my advisory committee. Also thank you to Dr. Sing-Hoi Sze for his time for substituting for Dr. Gutierrez-Osuna at my defense. Their insightful comments and constructive criticism helped me improve my research.

I wish to thank Nick Pattengale, Eric Gottlieb, and Bernard Moret for providing the code for Day's algorithm as well as the code for their PGM-Hashed RF distance matrix algorithm. I would also like to thank Matthew Gitzendanner, Bill Murphy, Paul Lewis, and David Soltis for providing us with the Bayesian tree collections used in this thesis. I also thank Matt Yoder for providing us with some of the biological sequences used in this thesis.

In addition, I would like to thank my friends and fellow students at Texas A&M University for numerous discussions related to my research work. I sincerely thank current members of our lab for being supportive of me during this work.

Last, but not least, I would like to thank my parents and my family members for their continuous support and encouragement. I am especially grateful to my wife

for her endless support and love. Without her dedication and belief in me, this work would have been impossible. Also I'd like to thank my daughter, Claire. Her lovely smiles always remind me of the reason for this work.

TABLE OF CONTENTS

CHAPTER		Page
I	INTRODUCTION	1
	A. Objective and Approach	1
	B. Contributions	7
	C. Dissertation Organization	9
II	A PRIMER ON EVOLUTIONARY TREES	10
	A. Representation of Taxonomic Information	10
	B. Representation of Evolutionary Relationships	11
	C. Representing Evolutionary Trees in Newick Format	14
	D. Applications of Evolutionary Trees	15
	E. Reconstruction of Evolutionary Trees	17
	1. Step 1 – Determining the Range of Study	18
	2. Step 2 – Performing Multiple Sequence Alignment	18
	3. Step 3 – Building Evolutionary Trees	19
	a. Neighbor-Joining (NJ)	19
	b. Maximum Parsimony (MP)	20
	c. Maximum Likelihood (ML)	20
	d. Bayesian Inference	21
	e. Tree Rearrangement	21
	F. Evaluating Evolutionary Trees	22
	G. Summary	24
III	RELATED WORK	25
	A. Consensus Tree Algorithms	25
	1. Motivation	25
	2. Strict Consensus Algorithm	25
	3. Majority Consensus Algorithm	26
	4. TASPI	28
	5. Discussion	29
	B. RF Distance Algorithms	30
	1. Day’s Algorithm	30
	2. PGM-Hashed Algorithm	31
	3. Discussion	32

CHAPTER	Page	
C.	Applications of Summarization Techniques	34
1.	Motivation	34
2.	Tree Islands	34
3.	Statistical Post-processing	34
4.	Multi-Dimensional Scaling (MDS) and Visualization .	35
5.	Trees of Trees	35
6.	Discussion	35
IV	USING HASH TABLES TO STORE EVOLUTIONARY TREES	37
A.	Hashing Technique	37
1.	Hash Table	37
2.	Hash Function	40
3.	Universal Hash Functions	41
4.	Collisions	42
B.	Evolutionary Trees and Hash Tables	43
1.	Tree Bipartitions and their Representations	43
2.	n -bitstring Representation	44
3.	k -bitstring Representation	46
4.	Implicit Bipartition Representation	48
a.	Universal Hash Functions, h_1 and h_2	50
b.	Collision Types and Probability	51
C.	Deterministic and Randomized Algorithms	55
D.	Summary	56
V	HASHCS: COMPUTING THE CONSENSUS TREES	58
A.	Motivation	58
B.	HashCS Algorithm Description	61
1.	Step 1: Populating the Hash Table	61
2.	Step 2: Constructing the Consensus Tree	65
C.	Theoretical Analysis	66
D.	Experimental Analysis	66
1.	Motivation	66
2.	Phylogenetic Tree Collections	67
a.	Biological Tree Collections	67
b.	Artificial Tree Collections	68
3.	Implementations and Platform	70
4.	Experimental Results	70
a.	Performance on Biological Trees	71

CHAPTER	Page
	b. Performance on Artificial Trees 73
	E. Summary 75
VI	HASHRF: COMPUTING THE ROBINSON-FOULDS DISTANCE 80
	A. Motivation 80
	B. Definition: Robinson-Foulds (RF) Distance 81
	C. HashRF Algorithm Description 82
	1. Step 1: Populating the Hash Table 82
	2. Step 2: Calculating the RF Distance Matrix 83
	3. Handling Collisions 85
	D. Extensions of HashRF 86
	1. WHashRF: Computing Weighted RF Distance 86
	2. HashRF(p,q): Computing Arbitrarily-sized RF Matrix 87
	3. HashRF(p,q) vs. HashRF 89
	4. Generate RF Distances of Arbitrary Cells in the Matrix 90
	E. Theoretical Analysis 91
	F. Experimental Analysis 92
	1. Motivation 92
	2. Implementations and Platform 92
	3. Experimental Results 92
	a. Performance on Biological Trees 93
	b. Performance on Artificial Trees 98
	G. Summary 101
VII	HASHCS AND HASHRF APPLICATIONS 104
	A. Application #1: Comparing Phylogenetic Search Heuristics 104
	1. Motivation 104
	2. Comparison Methods 106
	a. Maximum Parsimony Heuristics 106
	b. Comparing Collections of Trees 108
	3. Experimental Methodology 109
	4. Results and Discussion 111
	5. Summary 118
	B. Application #2: Effective Techniques for Summarizing Large Collections of Evolutionary Trees 121
	1. Motivation 121
	2. Previous Approaches 122
	3. Clustering Methodology 123

CHAPTER	Page
a. Clustering Large Tree Collections	123
b. Computing Tree Distances	124
c. Computational Platform and Implementations . .	125
4. Results and Discussion	125
a. Technique #1: Using MrBayes Run Labels to Cluster Trees	126
b. Technique #2: Using CLUTO to Cluster Trees . .	128
5. Summary	135
VIII CONCLUSIONS AND FUTURE WORK	138
A. Conclusions	138
B. Future Work	141
REFERENCES	144
VITA	161

LIST OF TABLES

TABLE		Page
I	Collision types in our randomized algorithms.	52
II	The table shows the number of top-scoring trees found by each algorithm.	114
III	Statistics for the two tree collections of interest.	124
IV	Detailed information for the $k = 2$ clustering of the 150 taxa trees. .	132
V	Detailed information for the $k = 8$ clustering of the 567 taxa trees. .	136

LIST OF FIGURES

FIGURE	Page
1	An example of a phylogenetic tree depicting the hypothesized evolution of human, chimp, mouse, rat, and fish. 2
2	Overview of the steps required to estimate the evolutionary relationships between human, chimp, mouse, rat, and fish. 2
3	Overview of our research work. 3
4	Overview of the summarization techniques of interest. 4
5	Phylogenetic rooted and unrooted trees depicting the hypothesized evolution of human, chimp, mouse, rat, and fish. 12
6	An example of Newick format representation of an evolutionary tree. 14
7	The neighborhood relationship between NNI, SPR, and TBR swapping operations. 23
8	An example of an NNI operation. 23
9	An example of a SPR operation. 23
10	An example of a TBR operation. 23
11	The relabeling and rerooting phase of Day's algorithm on trees T_1 and T_2 27
12	Each unique bipartitions in the tree is represented by a unique integer value (k -bit). 33
13	An illustration of our HashCS and HashRF algorithms, whose core feature is based on the novel use of hash tables. 38
14	The names are used as hash keys and the hash function computes the hash code for each key. 38

FIGURE	Page
15	An example of hash collision. 43
16	An example of bipartitions from evolutionary trees. 45
17	n -bit bitstring representation of tree T_1 's bipartitions. 45
18	The 5-bit bitstring bipartitions, B_1 and B_2 are collected from T_1 and B_3 and B_4 are collected from T_2 in Figure 4. 47
19	Calculating the h_1 hash code from n -bitstring bipartitions representations based on a post-order traversal of an example tree T 49
20	An example of populated hash table with the bipartitions from trees in Figure 16. 53
21	Overview of the consensus tree techniques of interest. 60
22	Overview of the HashCS algorithm. Bipartitions are from Figure 4. 62
23	Majority consensus tree for the input trees shown in Figure 4. 70
24	Running time and speedup of the strict consensus tree algorithms. 72
25	Speedup of the HashCS algorithm over PAUP*, its top consensus tree competitor. 73
26	Running time and speedup of the majority consensus tree algorithms. 74
27	Running time of the consensus tree algorithms on our artificial tree collections. 76
28	Speedup of the consensus tree algorithms on our artificial tree collections. 77
29	Overview of the HashRF algorithm. 84
30	Overview of computing the RF distance matrix using the HashRF(p,q) algorithm. 88
31	An example of 4×4 RF distance matrix. 91
32	Running time and speedup of the RF distance matrix algorithms. 95

FIGURE	Page
33	A closer view at the performance of HashRF. 96
34	Distribution of unweighted and weighted RF distances for the 150 taxa and 567 taxa datasets. 97
35	Speedup of unweighted HashRF over WHashRF. 98
36	The performance of the various RF matrix algorithms to compute a $t \times t$ matrix, where t is the number of trees, our 567 taxa dataset. . . 99
37	RF matrix algorithms performance on four of our artificial tree collections. 100
38	Memory usage of the HashRF and PGM-Hashed algorithms. 102
39	Comparing the topologies of the top-scoring trees found by the Pauprat and Rec-I-DCM3 heuristics. 113
40	Comparing the strict resolution rates of the top-scoring trees found by the Pauprat and Rec-I-DCM3 heuristics. 116
41	Comparing the majority resolution rates of the top-scoring trees found by the Pauprat and Rec-I-DCM3 heuristics. 117
42	Comparing the resolution rates of the top-scoring trees found by the Pauprat heuristics. 119
43	Comparing the resolution rates of the top-scoring trees found by the Rec-I-DCM3 heuristic. 119
44	Average RF rate between runs for 150 and 567 taxa tree collections. . . 127
45	Visualizing the 150 taxa trees with MDS based on RF rates. 129
46	Visualizing the 567 taxa trees with MDS based on RF rates. 129
47	Selecting the appropriate number of clusters, k 131
48	Visualizing the 150 taxa trees with MDS based on RF rates. 131
49	Average RF rate between the two clusters for the 150 taxa dataset. . . 132

FIGURE	Page
50	Visualizing the 567 taxa trees with MDS based on RF rates. 133
51	Average RF rate between the clusters for 567 taxa dataset. 134

CHAPTER I

INTRODUCTION

A. Objective and Approach

An evolutionary tree (or phylogenetic tree) is a hypothesis for depicting evolutionary relationships among organisms (or taxa) [1, 2, 3]. In a evolutionary tree, taxa are placed at the leaves and hypothetical ancestors occupy internal nodes, with the edges of the tree denoting evolutionary relationships (see Figure 1). The objective of a phylogenetic analysis (or inference) is to reconstruct the evolutionary relationships for a given set of taxa. Phylogenetic analyses can produce easily tens of thousands of equally plausible evolutionary trees (see Figure 2). Current approaches are not designed to analyze such large tree collections. In this thesis, we propose two algorithms (HashCS and HashRF) based on a novel hash table data structure to analyze very large tree collections (tens of thousands of trees) returned from a phylogenetic analysis. Furthermore, we demonstrate clustering and visualization applications of our algorithms in order to improve our understanding of a phylogenetic analysis techniques and produce more robust estimations of the underlying evolutionary tree. Figure 3 provides an overview of our work.

The journal model is *IEEE Transactions on Computational Biology and Bioinformatics*.

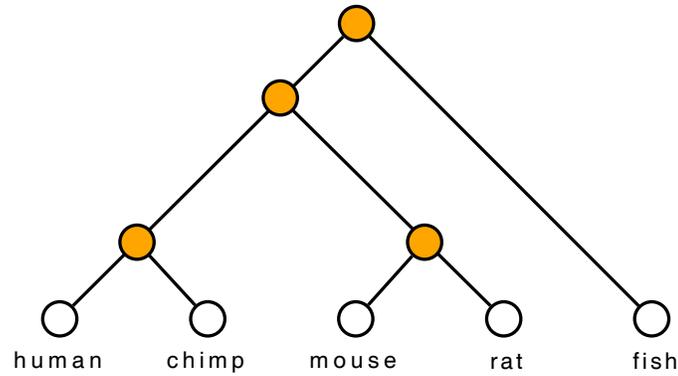


Fig. 1. An example of a phylogenetic tree depicting the hypothesized evolution of human, chimp, mouse, rat, and fish. The internal nodes (color nodes) of the trees represent hypothesized, extinct ancestors.

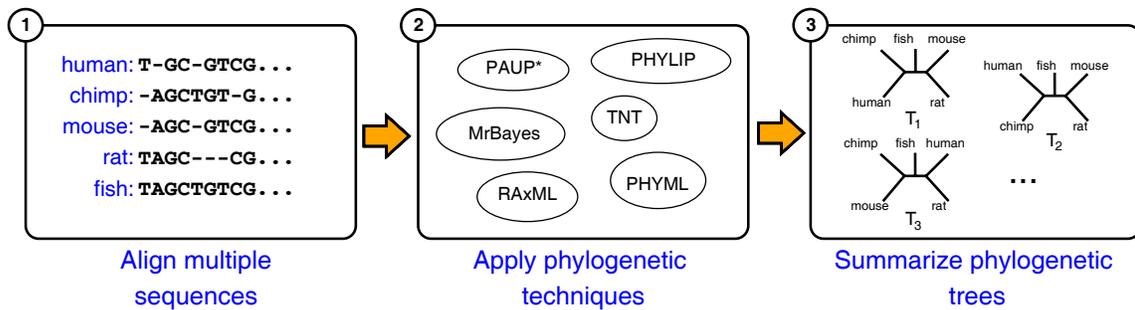


Fig. 2. Overview of the steps required to estimate the evolutionary relationships between human, chimp, mouse, rat, and fish. Summarization of trees is necessary since phylogenetic techniques often return many equally plausible evolutionary trees for the taxa of interest.

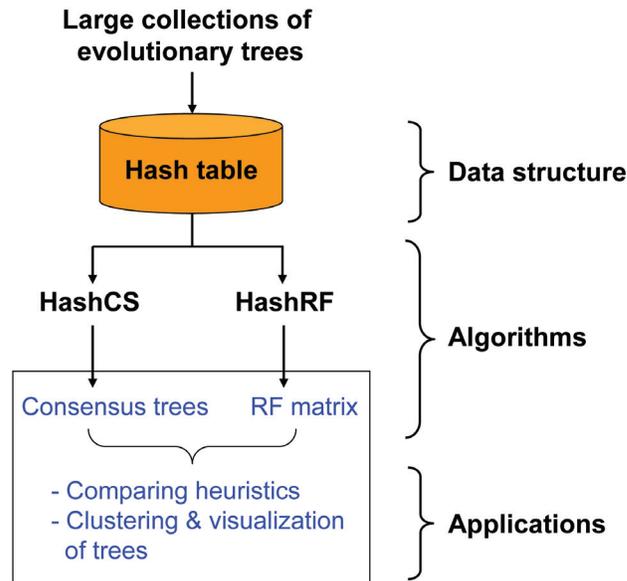


Fig. 3. Overview of our research work. Our hashing algorithms are based on hash tables. HashCS constructs a single consensus tree from a collection of t trees and HashRF computes $t \times t$ Robinson-Foulds (RF) distance matrix. We show two real-world applications, comparing heuristics and clustering trees.

Given the large tree collections often produced by phylogenetic techniques, summarizing and comparing the trees is necessary to better understand the true evolutionary history based on the multiple trees. Consensus trees and topological distances are often used to summarize and compare the evolutionary relationships among the trees of interest. Figure 4 shows four evolutionary trees with five taxa. A phylogenetic tree can be defined by a set of bipartitions. Removing an edge from a tree separates the leaves on one side from the leaves on the other and this partition related with the edge is called bipartition. A strict consensus tree is consisted of the bipartitions which are agreed among all input trees and a majority consensus tree is constructed from bipartitions which appear in more than half of the input trees.

Consensus tree methods, such as strict consensus and majority consensus have long been used for summarizing trees. However, the current implementations of con-

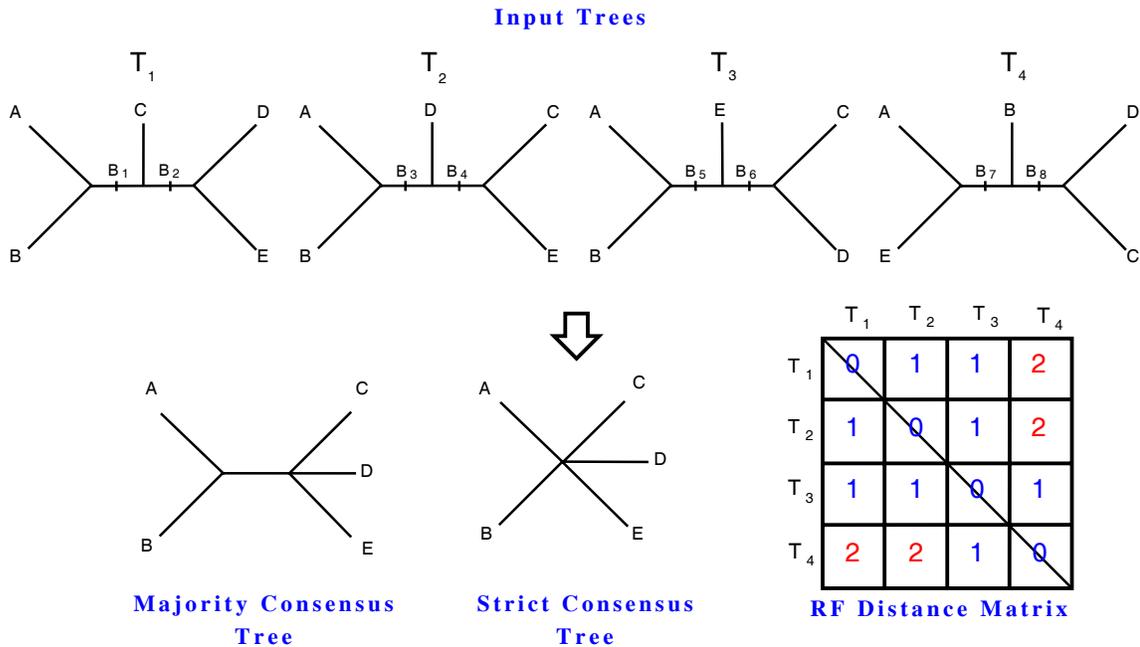


Fig. 4. Overview of the summarization techniques of interest. The tree collection consists of four phylogenies: T_1, T_2, T_3 , and T_4 . There are three different ways (majority consensus tree, strict consensus tree, Robinson-Foulds distance matrix) to summarize the information from the input trees. Bipartitions (or internal edges) in a tree are labeled B_i , where i ranges from 1 to 8.

sensus algorithms are too slow to summarize large collection of trees [4, 5, 6, 7]. Fast implementations of consensus tree algorithm allows researchers to analyze larger collections of phylogenetic trees in a smaller amount of time. Thus, we design and implement a hash-based consensus tree algorithm, called HashCS. Furthermore, such fast implementations, such as HashCS, can be easily incorporated into wide range of real-time applications. For example, phylogenetic inference software can be augmented with a fast consensus tree algorithm to determine the convergence of the search. Also one can add a topological distance criteria while navigating tree space and scoring trees.

Even though consensus trees are popular, summarizing many trees into a single

consensus tree loses valuable evolutionary information presents in the trees because disagreement among the branches results in unresolved trees. Computing the all-to-all relationships among the trees in terms of a distance matrix can preserve more of the relationships between the collection of trees [8, 9]. Computation of tree distance is supported by the current phylogenetic software packages, but they are inefficient for tree collections obtained from large-scale phylogenetic analyses. Thus, our research is targeted towards developing efficient tree topology distance algorithms to help researchers summarize and understand their large collections of trees that are obtained from a phylogenetic analysis. We choose the Robinson-Foulds (RF) topological distance [10, 11, 12, 13, 14, 15, 16] as our measure for the distance between trees, since RF distance is the most popular metric and it is widely used in the literatures. The RF distance between two trees is a normalized count of the bipartitions induced by one tree, but not by the other. We design and implement a hash-based RF distance algorithm, called HashRF that computes the $t \times t$ RF matrix (see Figure 4).

We study the performance of our hash-based algorithms (HashCS and HashRF) on both biological and artificial tree collections. Our experimental results show that our HashCS algorithm is up to 1.8 times faster than PAUP*, its closest competitor, and 100 times faster than MrBayes in computing consensus trees. Our HashRF algorithm is up to 2 times faster than PGM-Hashed, the second fastest RF matrix algorithm, and over 100 times faster than PAUP* in computing all-to-all RF distance matrix.

To show the merit of such fast algorithms, we study two applications: comparing phylogenetic search heuristics, and clustering and visualizing trees. In our first application, we discuss the traditional score and speed-based criteria to compare different heuristics. The superiority of heuristics were traditionally compared using score-based methods [17, 18, 19, 20, 21, 22, 23, 24, 25], which biases scientists toward

choosing faster heuristics. However, what value do slower heuristic implementations provide? Are score-based methods effective in distinguishing between different tree topologies? In this work, we shed light on the answers to these questions. We design novel methods to compare PaupRat [26] and Rec-I-DCM3 [27, 28], two popular search algorithms that use the Maximum Parsimony criterion. We empirically show topological distance methods such as Robinson-Foulds are more effective than parsimony scores at identifying heterogeneity within a collection of trees. Our entropy-based methods and heatmap visualizations show that PaupRat identifies more diverse trees than Rec-I-DCM3 in larger datasets. This ability suggests that PaupRat, while a slower heuristic implementation, has more opportunities to escape local optima. Furthermore, slower heuristics can produce different candidate trees in the course of their search. Lastly, our results imply that more powerful heuristics can be designed by combining tree topology with scores in the optimization criteria. Our entropy-based methods shed light on the behavior of individual heuristics, thus allowing for the design of better heuristics.

In our second application, we develop techniques to analyze two very large tree collections obtained from biologists. One of the tree collections contains 20,000 trees on 150 taxa (23 desert taxa and 127 others from freshwater, marine, and soil habitats). The other dataset consists of 33,306 trees on 567 taxa (560 angiosperms, seven outgroups). Oftentimes, a heuristic is executed several times (multiple runs) using the given sequence datasets and the same parameter settings to navigate tree space effectively. For our datasets, two and twelve runs of MrBayes were used to generate the tree files for 150 taxa and 567 taxa data files, respectively. Our empirical results show that there are distinct clusters of trees in our tree collections. We use two different techniques to identify distinct tree groups. Our first technique partitions the trees by the Bayesian run that produced them. The second technique uses a cluster-

ing algorithm to group the trees. Both techniques show that considering each tree collection as a single entity is a significant underrepresentation of the data. Partitioning the trees into distinct groups and summarizing each group separately is a better representation of the data. Furthermore, our results show that the benefits of our approach are better consensus trees as well as insightful information regarding the convergence behavior of the multiple Bayesian runs. Due to the level of dissimilarity of trees between runs it can be said that the different runs of the search algorithm did not converge, hence further runs may be desirable for convergence of the Bayesian analyses across runs [29, 30].

B. Contributions

The interdisciplinary research described in this thesis impacts both the communities of life scientists and computer scientists. Below, we discuss the contributions of our work to both communities in detail.

- *Novel solution for storing and processing evolutionary trees.* Our first and most important contribution is providing an information data repository for tree collections, or the hash table. Our hash table works as a compressed representation of large collections of phylogenetic trees. Our consensus tree algorithm implementation (HashCS) is based on the hash table and designed to efficiently deal with a number of trees for computing consensus trees. Our HashRF is another example of utilizing the hash table. Based on the same hashing technique, HashRF computes RF distance matrix. Both approaches help researchers to deal with an increasing number of trees.
- *A fast consensus tree algorithm to support researchers to analyze large collections of phylogenetic trees efficiently.* We design and implement, HashCS, a

efficient hash-based algorithm to compute large-scale strict and majority consensus trees. Our novel approaches demonstrate to deal with the challenging size and amount of data. We analyze the efficiency of our HashCS algorithm theoretically. Our experimental analysis also supports the efficiency based on both artificial and biological trees. Our experimental algorithmics also helps researchers to determine how to design and perform their empirical studies.

- *A fast all-to-all Robinson-Foulds distance algorithm based on a novel hashing technique.* We show that our HashRF algorithm effectively computes very large RF distance matrices. Such matrices provide researchers with the opportunity to apply data-mining methodologies in order to better understand their trees. Furthermore, experimental analysis based on both artificial and biological datasets show the effectiveness of HashRF.
- *An investigation of the real-world applications of our algorithms.* We introduce two practical applications for our efficient algorithms. First, by using relative entropy, our results show that for trees obtained from our larger datasets, there is more information content in topological distance measures than in parsimony scores. Second, we show that clustering based on RF can improve the resolution rates of the resulting consensus trees.
- *Open-source implementations of our hashing algorithms.* HashCS can be downloaded from <http://hashcs.googlecode.com>. HashRF can be downloaded from <http://hashrf.googlecode.com>. As a result, future work from researchers can use our implementations as a foundation for their work.

C. Dissertation Organization

The rest of the dissertation is organized as follows. Chapter II gives a primer on evolutionary trees. In Chapter III, we describe the related work of this thesis. The basics on hashing technique and the method to store evolutionary trees in the hash table is described in Chapter IV. Chapter V describes our implementation of HashCS algorithms for computing strict and majority consensus trees and shows the experimental results. Chapter VI discusses our HashRF algorithm and its extensions for computing the RF distance matrix between a collection of trees and shows extensive experimental results regarding the performance. Chapter VII presents two practical applications of our implementations. Finally, we conclude this work and discuss future research directions in Chapter VIII.

CHAPTER II

A PRIMER ON EVOLUTIONARY TREES

Our research is centered on analyzing large collections of evolutionary trees. In this chapter, we discuss evolutionary trees and their applications, tree reconstruction methods, and evaluation techniques.

A. Representation of Taxonomic Information

Biologists attempt to classify organisms based on their similarities and/or differences of many different characters. The traditional method of classification looks at the overall similarities between organisms. In phenetic taxonomy, groupings of organisms are based on mutual similarity of phenotypic (physical and chemical) characteristics. Phenetic groupings may or may not correlate with evolutionary relationships. Numerical taxonomy is a common approach to phenetic taxonomy, which employs a number of phenotypic characteristics to generate similarity or distance coefficients that may be represented in tree-like diagrams.

The fundamental idea that has driven recent advances in phylogenetics is known as the Hennig Principle [31], and is as elegant and fundamental in its way as was Darwin's principle of natural selection. The modern concept is based on evidence for historical continuity of information; closely related organisms have similar sequences and more distantly related organisms have more dissimilar sequences. Hennig's seminal contribution was to note that in a system evolving via descent with modification and splitting of lineages, characters that changed state along a particular lineage can serve to indicate the prior existence of that lineage, even after further splitting occurs. The principles represents homologous similarities among organisms come in two basic kinds: (i) synapomorphies due to immediate shared ancestry, and (ii) symplesiomor-

phies due to more distant ancestry. Synapomorphies are the key to reconstructing truly natural relationships of organisms.

Biologists also can compare DNA, RNA, and protein sequences among different organisms to unravel evolutionary relationships and common ancestry. Comparisons of the DNA sequences of various genes between different organisms can tell a scientist a lot about the relationships of organisms that cannot otherwise be inferred from morphology, or an organism's outer form and inner structure. Because genomes evolve by the gradual accumulation of mutations, the amount of nucleotide sequence difference between a pair of genomes from different organisms should indicate how recently those two genomes shared a common ancestor. Two genomes that diverged in the recent past should have fewer differences than two genomes whose common ancestor is more ancient. Therefore, by comparing different genomes with each other, it should be possible to derive evolutionary relationships between them, the major objective of molecular phylogenetics.

B. Representation of Evolutionary Relationships

Systematists describe the pattern of evolutionary relationships among taxa to help us understand the history of all life. The most convenient way of visually presenting evolutionary relationships among a group of organisms is through tree structure called evolutionary trees (or phylogenetic trees). In an evolutionary tree, known organisms (or taxa) are placed at the leaves and hypothetical ancestral organisms occupy internal nodes, with the edges of the tree denoting evolutionary relationships (see Figure 5). Phylogenetic trees have been used successfully in designing more effective drugs, tracing the transmission of deadly viruses, and guiding conservation and biodiversity efforts [32, 33].

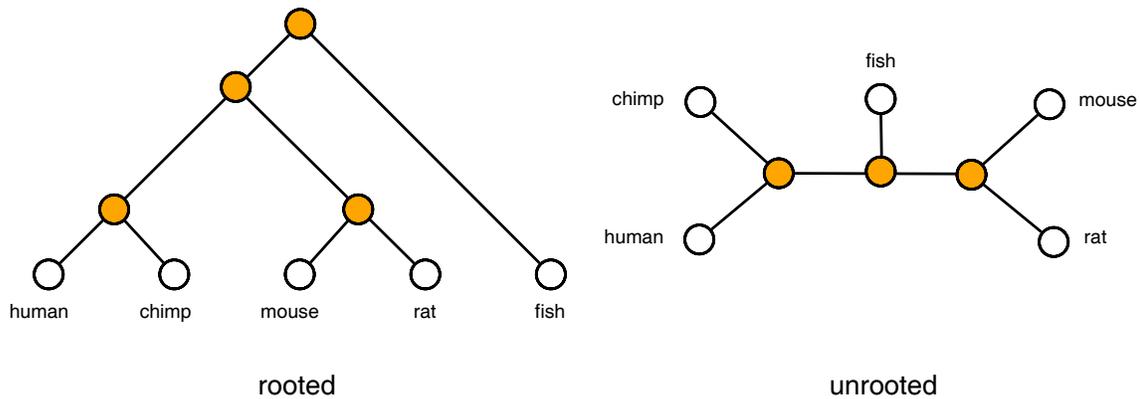


Fig. 5. Phylogenetic rooted and unrooted trees depicting the hypothesized evolution of human, chimp, mouse, rat, and fish. The internal nodes (color nodes) of the trees represent hypothesized, extinct ancestors.

A phylogenetic tree is composed of nodes, each representing a taxonomic unit. Leaf nodes show the current living species and internal nodes represent hypothetical ancestor of the descendants. Edges (or branches) illustrate the relationship between the taxa in terms of descent and ancestry. The topology defines the evolutionary relationships among the nodes and the branch length usually represents the number of changes that have occurred in the branch. A phylogenetic tree can be represented in both *rooted* and *unrooted* (see Figure 5). A rooted tree is a directed tree with an explicit ancestral node. On the other hand, a unrooted tree does not have a root node. An unrooted tree only specifies the relationship among species, without identifying a common ancestor, or evolutionary path. Operational Taxonomic Units (OTUs) are the species or other groups under study.

There are three types of groups which depict evolutionary relationships [31]. In an evolutionary tree, a group which consists of two or more taxa or DNA sequences that includes both their common ancestor and all of their descendants is called a clade or *monophyletic* group. *Paraphyletic* taxon is a group of organisms in which the most recent common ancestor of all those organisms and some, but not all, of

that ancestor's descendents are included. *Polyphyletic* taxon is a group composed of a number of organisms which might bear some similarities, but does not include the most recent common ancestor of all the member organisms (usually because that ancestor lacks some or all of the characteristics of the group). Among a given group of organisms, the shared derived characters are generally the less common characters. The evolutionary interpretation is that these characters of organisms are more recently evolved. They are contrasted with primitive characters. A polyphyletic group is an example of convergent evolution. An example of convergent evolution is the bat and the bird. They may look similar, but it's not because they're close relatives but because they've evolved similar adaptations.

When constructing a phylogenetic tree, detecting shared, derived similarity among organisms under study is very important. *Homology* denotes the relationships of characteristics which are shared among species due to descent homologous similarities from a common ancestor and are the basic criteria in the phylogenetics and comparative biology for determining the place of each taxon in the phylogenetic tree. For example, if twins share the same features like black eyes and black hair which are descended from their parents, the features are homologous. However, the similarity between species is not only from the homology. *Analogy* means the similarity between species that have no common ancestor. If two species show relationships, those relationships are not only caused by homology between them, but also by analogy. The alignment of DNA sequences establishes positional homology of individual characters (nucleotides). In phylogenetics, one attempts to estimate the true evolutionary history based on similarity which can due to homology, but not analogy. When homology is applied to genes, at least two fundamentally different subclasses must be distinguished: *paralogy*, the relationship between genes that have originated by gene duplication; and *orthology*, which refers to genes that originated by speciation. Systematic biologists

try to identify and understand the evolutionary relationships among the many different forms of life on earth. Phylogenetic trees are the critical tools for them to depict the relationships they find.

C. Representing Evolutionary Trees in Newick Format

In addition to the graphical representations of evolutionary tree, a string format, called "Newick" format (or "New Hampshire" format), is often used to represent the trees in text file and is supported by most software tools related with phylogenetic analysis. Because of the simplicity of the format, it is widely used for exchanging trees between different types of softwares. For example, Figure 6 shows an evolutionary tree and its Newick format representation.

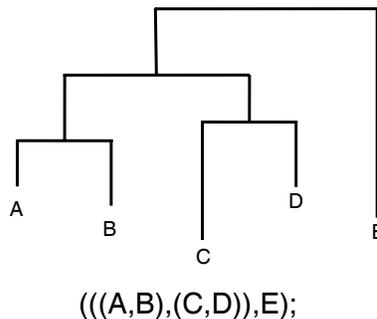


Fig. 6. An example of Newick format representation of an evolutionary tree.

A Newick format tree uses parenthesis and commas to maintain the hierarchical relationship between taxon names and needs a semi-colon at the end of the tree string. Both binary and multifurcating trees can be represented. Internal nodes are represented by a pair of matched parentheses. Between them are representations of the nodes that are immediately descended from that node, separated by commas. In the above example, "(A,B)" means there is an internal node which has two child

nodes, "A" and "B". The order of taxon names are meaningless. For example, "(A,B)" and "(B,A)" have the same meaning. Branch lengths can follow each taxon name and internal node starting with colon. Branch lengths can be incorporated into a tree by putting a real number, with or without decimal point, after a node and preceded by a colon. This represents the length of the branch immediately below that node. Thus the above tree might have lengths represented as:

$$(((A:0.2,B:0.3):0.3,(C:0.5,D:0.3):0.2):0.3,E:0.7):0.0;$$

Bootstrap values can be shown before the branch lengths or after the lengths. Whitespace (spaces, tabs, carriage returns, and linefeeds) is not allowed in both numbers and strings. Whitespace elsewhere is ignored. If one wants a space in taxon name, the underscore character can be used and any Newick parser will convert the underscores within names into spaces.

D. Applications of Evolutionary Trees

Phylogenetic analysis has been gaining its popularity since the start of the 1990s, largely because of the explosion of DNA sequence information obtained initially by PCR analysis and more recently by genome projects. The application of phylogenetic tree is not limited to biological area. It is not hard to find successful applications of phylogenetic analysis. Notable examples are found in the following areas:

- Evolution studies
- Systematic biology
- Medical research and epidemiology
- Ecology

- Forensic investigation
- Linguistics

A famous example of applying phylogenetic trees can be found in tracing HIV (Human Immunodeficiency Virus). In the early 1990s, a Florida dentist was convicted of transmitting HIV to his patients by injecting his blood to them. The conclusion was reached by epidemiologic investigation and comparing the genetic sequences of his virus to the virus in his patients and of local HIV-infected control people. The genetic comparison found that five of the patients had closely-related viruses, while other patients had unrelated virus. Phylogenetics was also used to study the evolutionary relationship between human and other primates. Phylogenetics also contributed to revealing the origins of HIV and researchers found the tree constructed from samples of HIV-1 looked like a star tree which means the global AIDS epidemic began with a very small number of viruses, perhaps just one, which have spread and diversified since entering the human population. They also found that the closest relative to HIV-1 among primates is the SIV of chimpanzees, the implication being that this virus jumped across the species barrier between chimps and humans and initiated the AIDS epidemic. A phylogenetic analysis of HIV-1 sequences has suggested that 1931 is the best estimate for the time HIV started to spread. Such researches make it possible to investigate the historic and social conditions that might have been responsible for the start of the AIDS epidemic. A study based on the molecular data and molecular clock confirmed that humans, chimpanzees and gorillas form a single clade but suggested that the split between human and the primates occurred 5 million years ago (it was 15 million years before). By 1997, it was confirmed that the chimpanzee is the closest relative to human [34]. To conclude, phylogenetic analysis is a powerful tool for organization and interpreting of molecular data and it is possible

to earn valuable information from a phylogenetic tree.

E. Reconstruction of Evolutionary Trees

Since the true evolutionary history is unknown, many phylogenetic techniques use stochastic search algorithms to solve NP-hard optimization criteria such as maximum likelihood and maximum parsimony [17, 25, 35, 36, 37, 38, 39, 40, 41]. Under these criteria, trees that have better scores are believed to be better approximations of the truth. Phylogenetic heuristic methods take a set of aligned sequences as input and generate phylogenetic trees which represent the best hypothesis (or hypotheses) for the true evolutionary history of the taxa. Note that the best (or optimal) tree(s) reconstructed from phylogenetic heuristics does not necessarily mean that the true evolutionary history is found. Figure 2 shows the major steps in phylogenetic analysis. Sequences are collected from five species and aligned with each other. One of the phylogenetic heuristic methods then takes the aligned sequence and infers the trees which are equally plausible, given the data and optimization criterion.

The *true* phylogenetic tree for a set of species is assumed to be unique [2, 42, 43, 44]. Phylogenetic reconstruction methods often estimate the true evolutionary history using the molecular sequences as their sole input. Consequently, phylogenetic reconstruction methods often yield different inferred trees for the same set of organisms. For example, a single phylogenetic approach (such as a Bayesian analysis [45, 46, 47]) can produce tens of thousands of equally plausible trees. Moreover, large tree collections can also be produced by bootstrap tests on phylogenies to access the uncertainty of a phylogenetic estimate [48, 49, 50].

1. Step 1 – Determining the Range of Study

The first thing to resolve before beginning a phylogenetic analysis is the underlying biological questions. When choosing a group to study, the sample range can be either narrow and deep or wide and shallow. If one focuses on a specific gene family the sequences collected should be thoroughly related within the family. Otherwise, the sequences should represent a wide range of species. After determining the ranges of species and genes, there are several sources of collecting sequence data including literature survey and searching sequences through gene database such as NCBI (<http://www.ncbi.nlm.nih.gov/>), Ensembl (<http://www.ensembl.org/index.html>), and UCSC (<http://genome.ucsc.edu/>). Also, biologists can collect the data themselves from the field.

2. Step 2 – Performing Multiple Sequence Alignment

As Figure 2 shows, the sequences for the organisms under study should be aligned to each other. Excellent surveys of the progresses in multiple sequence alignment can be found in [51, 52, 53]. The purpose of the sequence alignment is to establish or estimate the positional homologies between individual nucleotides in the sequences, based on the assumption that two genomes that diverged in the recent past should have fewer differences than two genomes whose common ancestor is more ancient. There are many approaches to perform the multiple sequence alignment such as ClustalW [54] or T-coffee [55, 56]. However, automated approaches often do not correctly identify regions of conservation within a gene and thus manual alignment is often performed to obtain better alignment results. For these reasons, there are tools for supporting manual sequence alignment such as BioEdit [57].

3. Step 3 – Building Evolutionary Trees

Reconstructing evolutionary trees is not a trivial task. Since the true evolutionary history for a set of organisms is unknown, the problem is often reformulated as an NP-hard optimization problem. Trees are given a score, where trees with better scores are believed to be better approximations of the true evolutionary history. For n taxa, there are an exponential number of evolutionary hypothesis: $(2n - 3)!!$ possible solutions to be exact. As a result, an exhaustive exploration of the space of possible solutions (or "tree space") is infeasible except for small numbers of taxa ($n < 30$). Thus, the most popular techniques in the field use heuristics to reconstruct evolutionary trees for a set of taxa.

Phylogenetic search heuristics for reconstructing phylogenetic trees can be divided into two categories: distance matrix based methods such as neighbor-joining [58, 59, 60] and discrete data based methods such as maximum parsimony, maximum likelihood [2] and Bayesian methods [61]. Distance matrix based methods are also categorized into phenetic approaches and involve a computation of pair-wise distance matrix among all taxa and uses the matrix to construct tree diagrams. Phenetic methods are less compute-intensive. On the other hand, the discrete data based methods are generally referred as cladistic approaches. Cladistic approaches are very compute-intensive because they take all possible topologies into account. Those trees are evaluated by a specified criterion to select the best tree(s).

a. Neighbor-Joining (NJ)

NJ method is based on evolutionary distance data as well. The principle of NJ method is to find pairs of OTUs that minimize the total branch length at each stage of clustering of OTUs starting with a starlike tree. This method is widely used for

quickly generating starting tree for other reconstruction method. However, it gives only one possible tree and it is strongly dependent on the used model of evolution. It is a phenetic, not phylogenetic method.

b. Maximum Parsimony (MP)

MP method is based only on the informative sites. The principle behind MP method is Occam's razor which means that the simpler is the better answer. Thus it tries to find a tree with the smallest number of evolutionary changes. It has to deal with all informative sites thus it is slow. However, it does not imply a specific model of evolution and provides all equally parsimonious topologies.

c. Maximum Likelihood (ML)

Given a model of evolution, ML tries to find tree topology which explains the relationships between the sequences with the highest probability. The tree that maximizes the likelihood of the observed data is optimal. Likelihood is the probability of the data (alignment), given a tree (with topology and branch lengths specified) and a probabilistic model of evolution. Thus, ML methods are explicitly based on the model such as Jukes-Cantor (JC69) or Kimura (K80 or K2P). ML is very compute intensive and the use of inadequate likelihood models can lead to interpretation in real data sets. To decide which model best fits the data, the likelihood values given by the different models for the data are calculated and compared. The model to choose is the simplest model that gives a likelihood not significantly lower than the likelihood given by a more general model [2].

d. Bayesian Inference

Bayesian method allows estimation of the posterior probability of a tree, given a prior probability (marginal probability) of the tree. It is closely related to ML methods, differing only in the use of a prior distribution (which would typically be a tree). In Bayesian theory, the summation in the denominator is over all the trees possible for given species. In other words, our prior expectation is that all possible trees are equally probable. All trees are considered equally likely a priori – in other words, the prior probabilities are said to be "flat" or "uniform". Consequently, the posterior probability of a tree cannot be calculated. However, the introduction of Markov Chain Monte Carlo (MCMC) methods has given a new impetus to Bayesian inference which reduce the computational burden of approximating the posterior probabilities of trees [46, 61]. Although Bayesian analysis using MCMC is an elegant method for solving many problems, it is relatively new and there is a number of unsolved questions, e.g. determining convergence of separate Markov Chains [29, 30], and discrepancy between Bayesian posterior probabilities and nonparametric bootstrap test values [62].

e. Tree Rearrangement

In phylogenetic search heuristics, tree topology should be rearranged using different branch swapping algorithms [63] to perform thorough search in the tree space. As shown in Figure 7, NNI is regarded as simple and fast rearrangement operation and all possible NNI operations are a subset of SPR operations. Also all possible SPR operations are included in the set of all possible TBR operations [64]. Branch swapping by nearest neighbor interchange (NNI) interchanges one subtree on one side of each internal branch with one of the two from the other side. In Figure 8, there are

four subtrees (neighbors) represented by S_1 , S_2 , S_3 , and S_4 . For example, the subtree S_2 can be swapped with S_3 or S_4 . In the figure, T' shows the resulting tree after swapping S_2 and S_4 .

Subtree pruning and regrafting (SPR) performs pruning one subtree from the tree, which is subsequently re-grafted to a different location on the tree [9, 65, 66]. Figure 9 shows an example of SPR operation. The edge x in T_1 is cut making the subtree S_1 detached from T_1 . And then S_1 is regrafted on the edge y resulting T_2 .

Tree bisection and reconnection (TBR) consists of dissecting the tree into two subtrees and re-connecting by joining a pair of branches, one from each subtree. In Figure 10, the edge x is cut resulting two subtrees, S_1 and S_2 . Any edges in S_1 can be attached on any edges in S_2 . In the example, the y edge in S_1 is selected and attached on z edge in S_2 . Note that the topology of S_1 is changed to S_1' in the resulting tree T_2 .

F. Evaluating Evolutionary Trees

Once we have the resulting reconstructed trees, the next step measures the confidence level of each branch (or edge) within the tree. The most commonly used method is bootstrapping [48]. In this method, characters are resampled with replacement to create many bootstrap replicate data sets (pseudosamples) and each bootstrap replicate data set is analyzed. Then, the frequency with which the branching pattern occurs in each of these random subsamples is calculated [49]. Agreement among the resulting trees is summarized with a majority consensus tree and the additional information is given in partition tables.

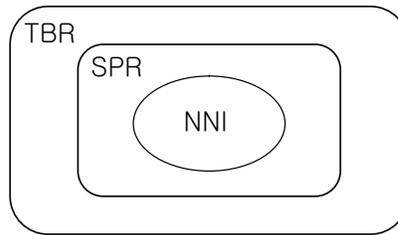


Fig. 7. The neighborhood relationship between NNI, SPR, and TBR swapping operations.

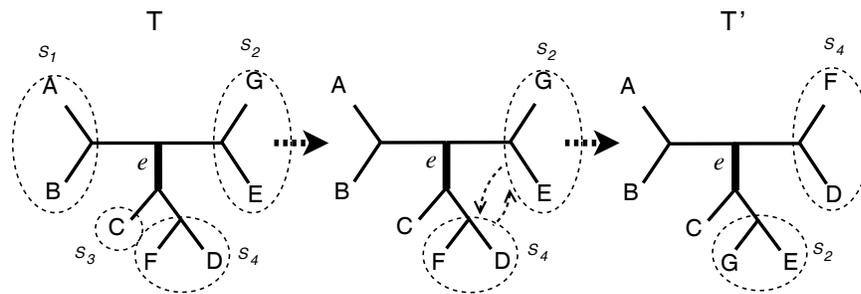


Fig. 8. An example of an NNI operation.

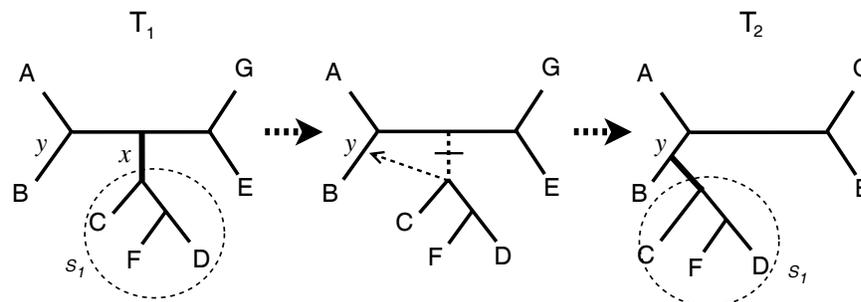


Fig. 9. An example of a SPR operation.

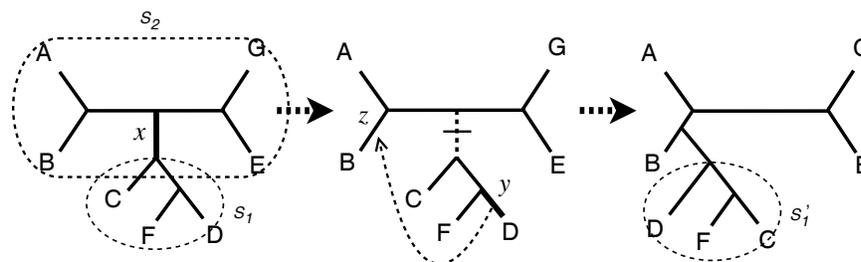


Fig. 10. An example of a TBR operation.

G. Summary

Evolutionary trees are most popular tool for scientists to represent evolutionary relationship among organisms under study. Nowadays, evolutionary history is reconstructed by phylogenetic heuristics using molecular sequences like RNA and DNA. Homology is the basis for the molecular phylogenetics which means the ancestor and descendents in closer evolutionary relationship should share more amount of molecular sequences. Phylogenetic trees have been applied to various areas including systematic biology, forensic study, and medical studies.

A phylogenetic tree can be rooted or unrooted and edges represent the lineage information between ancestor and descendents. Phylogenetic reconstruction is NP-Hard problem. Thus, there are many heuristic approaches to produce more accurate estimate for the true evolutionary history with less amount of computation time. The reconstruction is consisted of: (i) range of study determination, (ii) multiple sequence alignment, (ii) phylogenetic analysis, and (iv) evaluation. More advances in molecular techniques and improvements in computational methodology, larger number of species is being taken into account in phylogenetic analysis. Furthermore, the number of resulting candidate trees are exponentially being increased. All this trends emphasize the need for more delicate approaches to analyze, summarize and visualize the collections of phylogenetic tree.

CHAPTER III

RELATED WORK

In this chapter, we review the prior work in the areas of summarization of phylogenetic trees, which includes computing consensus trees, computing RF (Robinson-Foulds) distance matrices, and analyzing biological trees approaches. Afterward, we compare our HashCS and HashRF algorithms to the related work in the field.

A. Consensus Tree Algorithms

1. Motivation

Even though it is assumed that the true phylogenetic tree for a specific set of organisms is unique, phylogenetic search heuristics often produce tens of thousands of equally plausible trees. A number of different techniques have been developed to summarize tree collections by building consensus trees. Two surveys of consensus methods [67, 68] present a theoretical classification and comparison of various consensus approaches. Traditionally consensus trees are the main approach life scientists use to analyze and understand the evolutionary relationship from the trees. However, current tools for constructing consensus tree cannot accommodate the growing requirements of larger phylogenetic analyses. Before introducing our HashCS algorithm to compute consensus trees, it is worth reviewing previous consensus tree algorithms.

2. Strict Consensus Algorithm

A strict consensus tree is consisted of the bipartitions which are agreed among all input trees. Again, the set of bipartitions from a tree can be obtained through partitioning the tree into two subtrees by removing all internal edges one by one. For

computing the strict consensus tree, Day [69] proposed the first optimal $O(nt)$ time complexity algorithm for computing the strict consensus between t trees, where each tree consists of n taxa. To compute the strict consensus tree from two input trees, the first step requires rerooting the trees to have the same taxon as its root node and relabeling the taxon names from both trees. Figure 11 shows an example of rerooting and relabeling of two input trees with six taxa. Rerooting is performed with "6" as a new root and T_1 is relabeled by the rule that each leaf node is numbered in increasing order starting with '1' based on a post-order traversal of the tree. Then, T_2 is relabeled corresponding to the place of each name in T_1 . Next, by representing trees by their post-order sequence with weights (PSW) [70], Day's algorithm constructs a special cluster representation which gives the algorithm the ability to determine in constant time whether a bipartition found in one tree exists in the other tree. Final step finds the set of bipartitions which exist in both trees and returns the strict consensus tree.

3. Majority Consensus Algorithm

Amenta, Clark, and St. John [71] developed an $O(nt)$ algorithm for computing the majority consensus tree. The novelty of their work was to propose the first linear time algorithm for computing majority consensus tree. Although Amenta et al. focus on majority trees, their work is relevant to strict trees as they are a more stringent type (100% agreement instead of 50% agreement) of majority tree. Their approach takes advantage of using a hash table and consists of two major steps: (i) reading the t trees and inserting the relevant bipartition information into the hash table and (ii) using the bipartition information in the hash table to construct the majority tree. However, constructing the majority tree from the entries in the hash table requires an additional traversal of the entire tree collection. HashCS is the name of our fast implementation of Amenta et al.'s algorithm – although there are some differences

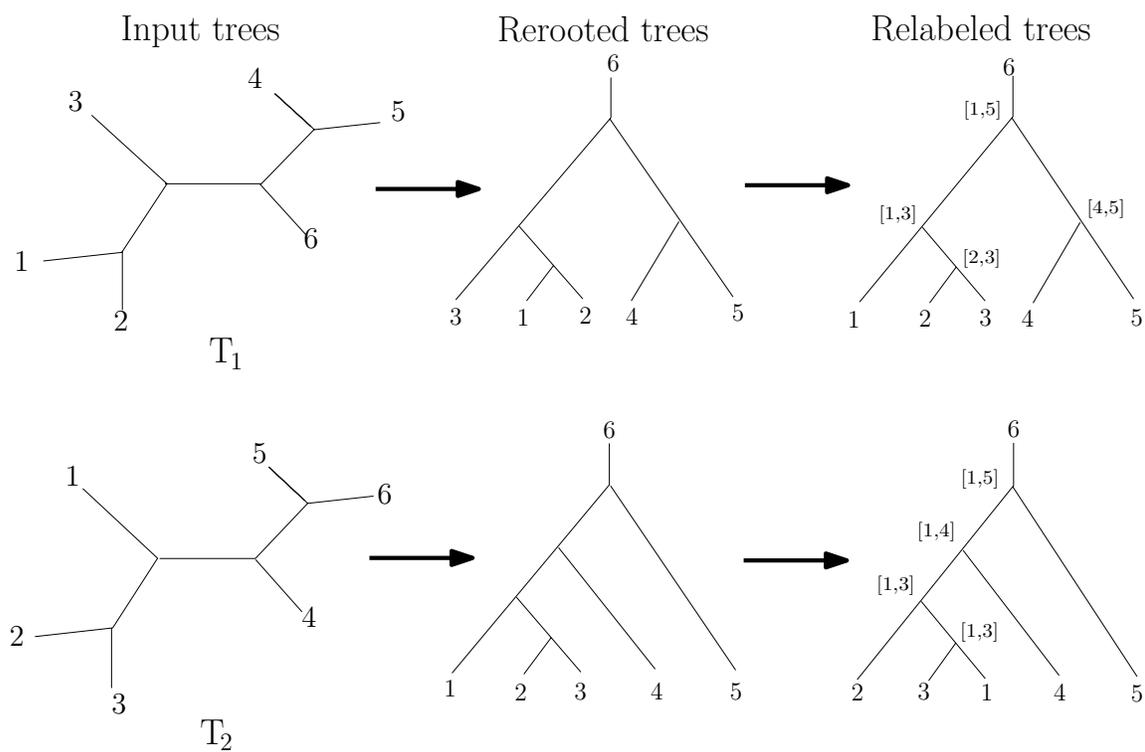


Fig. 11. The relabeling and rerooting phase of Day's algorithm on trees T_1 and T_2 .

between HashCS and what Amenta et al. describe in their work.

4. TASPI

The Texas Analysis of Symbolic Phylogenetic Information (TASPI) system is a recently proposed technique to compute consensus trees [7, 72, 73]. One of the novelties of TASPI is that it incorporates a new format for compactly storing and retrieving phylogenetic trees. TASPI stores a common subtree once, and then each further time the common subtree is mentioned, TASPI references the first occurrence. This saves considerable space since potentially large common subtrees are only stored once, and the references are much smaller. Basically TASPI representation is based on Lisp [74] language and convert Newick trees into Lisp lists. The lists are stored in the system using "hash-consing" which also a feature of Lisp to share values that are structurally equal. Hash-consing is known to reduce the memory allocation overhead and provides the fast access performance. Experimental results on several collections of maximum parsimony trees show that the TASPI system outperforms PAUP* [75] and TNT [76] in constructing consensus trees. Their maximum parsimony tree collections—inferred with parsimony ratchet [26, 77] and Rec-I-DCM3 [39] algorithms—have trees consisting of 328 to 8,506 taxa. The number of trees in the collection range from 47 to 10,000. The 10,000 trees were based on the 500 taxa rbcL dataset [78]. The 8,506 taxa dataset consisted of 47 trees. All other collections had 2,505 or fewer trees. An implementation of the TASPI system does not appear to be available to use for experimental comparison in our work. However, since it was demonstrated clearly that TASPI and PAUP* are much faster than TNT's consensus algorithm [73], we do not explore the performance of TNT. A comparison of Phylip and MrBayes consensus methods was not included in the experiments with TASPI. Hence, we include those techniques in our study.

5. Discussion

Our HashCS algorithm supports both strict and majority consensus trees and has $O(nt)$ time complexity to compute consensus tree between two trees. The details of our HashCS algorithm for computing consensus trees are discussed in Chapter V. Further, HashCS provides an option to set the resolution rate for computing majority consensus tree. From the performance point of view, our implementation is based on hash tables which allows HashCS to have $O(1)$ access time in average. We take advantage of randomness to make the probability of failure very small, which is $O(\frac{1}{c})$, where in our experiments c is 1,000. In practice, the probability of failure is 0%. Also we have our implementation opened for public access (<http://hashcs.googlecode.com>).

Day's algorithm is an $O(nt)$ time complexity algorithm for computing only strict consensus tree. Even though it has the same time complexity with HashCS algorithm, our experimental results with t trees show that the performance of Day's algorithm is much slower than HashCS since the cluster table in Day's algorithm should be updated for each of the input trees. On the other hand, Day's algorithm does not have a probability of producing incorrect answer because it is an exact algorithm.

Our HashCS algorithm is motivated by Amenta et al.'s work. Our work differs from Amenta et al.'s in two distinct ways. First, from an algorithmic perspective, one difference lies in how often the two approaches traverse the t input trees. Our HashCS implementation requires a processing the collection of t trees once to construct a consensus tree. The original specification of Amenta et al.'s algorithm requires processing the t trees at least two times. The other difference between the two approaches is that our HashCS implementation does not insert all bipartitions into the hash table. Only bipartitions that satisfy the strict or majority consensus criteria are inserted into the

table. Thus, by processing the collection of trees once and only storing relevant bipartitions in the hash table significant time can be saved by our HashCS implementation in practice—especially for large tree collections.

Secondly, our work takes a much broader view to designing and empirically analyzing consensus tree approaches. For example, Amenta et al. do not provide any empirical evidence of their algorithm’s running time. Nor, do they empirically compare their approach to other consensus tree implementations. In this thesis, we extensively compare the performance of consensus-tree building implementation on a diverse collection of trees. An additional difference between our work and Amenta’s approach is that HashCS doesn’t not explicitly check for double (Type 2) collisions. However, Amenta et al. do.

Newick format is the standard way of storing a collection of phylogenetic trees in a string format [79]. Our hash-based algorithms read the Newick trees and store the trees in hash tables. The novel idea results in fast access time and efficiency in storage requirements. TASPI also incorporates a new format for compactly storing and retrieving phylogenetic trees and uses a kind of hash technique. However, the implementation is built upon Lisp environment which may cause additional overheads because Lisp needs a Lisp runtime to execute codes. Further, the implementation of the TASPI system does not appear to be available to use for experimental comparison.

B. RF Distance Algorithms

1. Day’s Algorithm

As introduced in computing consensus trees, the Day’s optimal algorithm [69] can also be used to compute the RF distance between two trees and it has $O(nt)$ time complexity. Thus, Day’s algorithm to compute all-to-all RF distances between t trees

needs $O(nt^2)$ time. By using the PSW, Day’s algorithm constructs cluster table representation which gives the algorithm the ability to determine in constant time whether a bipartition found in one tree exists in the other tree. The rerooting and relabeling routine and the creating cluster table routine is same with the case of the consensus tree. In the final step, the number of different bipartitions is counted instead of finding shared bipartitions. To convert the difference into RF distance value, Equation B is computed.

2. PGM-Hashed Algorithm

Pattengale, Gottlieb, and Moret [15] develop an $O(nt^2)$ algorithm that uses k -length bitstrings to represent each tree’s bipartitions. In their paper, Pattengale et al. describe a number of exact and approximate algorithms to compute the RF distance matrix. Even though their RF distance approximation algorithm carries a bounded error rate, the novelty of their approaches lies in presenting approximation paradigm in computing RF matrix and the approach reduces the running time enormously.

Since we focus strictly on exact approaches, we study Pattengale et al.’s Hashed algorithm, which we call PGM-Hashed. The algorithm starts by assigning a 64-bit integer random number to each taxon and using the XOR accumulator to combine the taxa numbers to represent the bipartition found during depth-first search traversal. Since each binary tree contains $n - 3$ bipartitions, the entire set of bipartitions collected from the t trees are stored in a $(n - 3) \times t$ two-dimensional array (or bipartition table). Entry (i, j) in the table represents the integer (converted from the 64-bitstring) representing bipartition i from tree j (see Figure 12). Although the PGM-Hashed does not explicitly use a hash table, it is considered a hashing approach because different bipartitions may be represented with the same 64-bit integer. Hence, the probability of collision is defined as $\frac{1}{2^{64}}$. If the chance of collision is larger than

$\frac{1}{2^{64}}$, the algorithm stops. In other words, the bitstring is not large enough to express all of the bipartitions from the set of trees collections uniquely.

Once the $(n - 3) \times t$ bipartition table is constructed, the RF distance matrix is computed. Each of tree j 's bipartitions (i.e., column j in the bipartition table) are sorted since they are stored as integer values. After the sort, the RF distance between the trees is computed. For each pair of trees T_i and T_j , two pointers p and q are used to compare the bipartitions of T_i and T_j , respectively. If the bipartition pointed to by p is equal to the one referred to by q , then both pointers are incremented which means they have the same bipartition. However, if the bipartitions are different, a difference counter is incremented, and either the p or q pointer is incremented appropriately. To get the RF distance, the value of the difference counter is subtracted from $n - 3$, which is the maximum number of bipartitions in a binary tree.

3. Discussion

We develop the HashRF algorithm to compute the $t \times t$ RF distance matrix between a collection of t trees. Day's algorithm for computing the RF distance between two trees is easily extended to computing the RF distance matrix for t trees. The resulting complexity of the algorithm is $O(nt^2)$, which is same with our HashRF algorithm. However, HashRF runs faster than Day's RF matrix algorithm in practice.

PGM-Hashed uses a k -bit bitstring instead of a n -bit bitstring to represent a bipartition, where $k < n$ and n represents the number of taxa. The method results in reduced storage requirements. The algorithm starts by assigning a 64-bit integer random number to each taxon and using the XOR accumulator to combine the taxa numbers to represent the bipartition, because they have to assign unique random numbers with bipartitions. In other words, if OR operator is used in PGM-Hashed, the probability of having same random integer numbers between different bipartitions

Bipartition Matrix

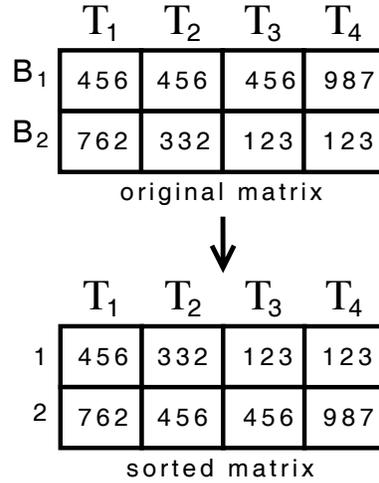


Fig. 12. Each unique bipartitions in the tree is represented by a unique integer value (k -bit). Both the original matrix and its sorted version, which is then used to compute the RF distance matrix, are shown.

is increased severely. On the other hand, the accumulative operator in HashRF is "OR" operator as a part of computing implicit bipartition representation using arrays of n random numbers and the modulo operator.

Another major difference between PGM-Hashed and HashRF is the bipartition storage. HashRF stores bipartition information into the hash table based on two universal hash functions and implicit bipartition representation and maximally realizes the $O(1)$ optimal lookup time. It computes the location in the hash table using the first hash function, and then the tree ID is stored at the location with BID which is computed by the second hash function.

The mechanism to compute the RF distance values from the bipartition storage differs as well. Once the hash table is populated with the bipartition IDs and the tree IDs, HashRF retrieves the vector of tree IDs one by one and computes the similarity matrix since the vector represents the trees which share the bipartition. Finally,

we have our implementation opened for public access (<http://hashrf.googlecode.com>).

C. Applications of Summarization Techniques

1. Motivation

Traditionally, the most common technique for summarizing large tree collections is to construct a single strict or majority consensus tree. A single consensus tree is a compact way to summarize the data. Unfortunately, its compactness obscures a lot of the information present in the original collection of t trees. Researchers have examined different methods to summarize large collections of trees that retain more information than a single consensus tree.

2. Tree Islands

Maddison [64] explored partitioning a collection of trees based upon the lengths (parsimony score) of the trees and the number of branch rearrangements by which they differ. He defines an *island* as a collection of interconnected shorter trees that is separated from other islands by longer trees. Two trees are considered connected if they differ by a single rearrangement (e.g., NNI, SPR, TBR) of branches.

3. Statistical Post-processing

Consensus trees have a lot of uses in terms of understanding the results of a phylogenetic analysis. For example, a Bayesian analysis often times uses a majority tree to summarize the sampled trees. Stockham, Wang, and Warnow [80] present an alternative approach by using clustering algorithms on the set of candidate trees. They propose bicriterion problems, in particular using the concept of information loss, and

new consensus trees called characteristic trees that minimize the information loss.

4. Multi-Dimensional Scaling (MDS) and Visualization

Hillis, Heath, and St. John explore the use of Multi-Dimensional Scaling (MDS) of tree-to-tree pairwise distances to visualize the relationships among sets of phylogenetic trees [81]. For example, to compare two Bayesian analysis, they obtained 6,000 total trees from MrBayes on a 44 taxon dataset [82]. Afterwards, they computed the unweighted and weighted RF distance matrices and fed them into their MDS algorithm. Hillis et al. also found their technique to be useful for exploring "tree islands" (sets of topologically related trees among larger sets of near-optimal trees), for comparing sets of trees obtained from bootstrapping and Bayesian sampling, for comparing trees obtained from the analysis of several different genes, and for comparing multiple Bayesian analysis.

5. Trees of Trees

In addition to the above approaches, a "trees of trees" (or meta-tree) approach has been recently developed [83]. Here, leaf nodes represent trees themselves and internal nodes represent consensus trees. Finally, Bonnard et al. has suggested multipolar consensus trees as an effective technique for summarizing a collection of trees [84]. The novelty of their method lies in displaying effectively all bipartitions that occur at a frequency rate greater than α , where $0 < \alpha \leq 100$.

6. Discussion

In Chapter VII, we show how to use our HashCS and HashRF algorithms for clustering and visualizing evolutionary trees. Our goal is to cluster tree collections effectively based on RF distance and quantify the results by analyzing consensus resolution

rate and introduce two methodologies. In our work, we are interested in identifying underlying information in the data through clustering. While the results of other methods are effected by the data, they do not necessarily point to the presents or absence of any trends in how the data was gathered.

Secondly, we use a combination of clustering and visualization to analyze collections of trees. To cluster large groups of trees, we consider two techniques: grouping trees by run and grouping them using a clustering algorithm. For visualization, we use heatmaps instead of MDS to quantify the similarities and dissimilarities among the clusters of trees. MDS visualizes the points (trees) of the clusters into two-dimensional space. It does not automatically determine whether points should be in the same cluster. Hence, under MDS, it is up to the human observer to group the points into clusters. The focus of our approach is on identifying similar groups of trees so that they can be summarized more effectively. Multipolar and meta-tree techniques are interested in effectively showing differences such as outlier trees and distinct bipartitions. Thus, these techniques would be good companions to the methods described in this thesis.

Finally, we are interested in studying collections containing tens of thousands of trees on large numbers of taxa (150 and 567 taxa). Although most of the previous work study collections with hundreds of trees, there have been studies that analyzed datasets with around 6,000 trees. One of Stockham et al. [80] analyses looks at 5,630 trees on 129 taxa. Hillis et al. [81] study 6,000 trees on 40 taxa. However, the largest tree collection (33,306 trees on 567 taxa) available to study is analyzed in this thesis.

CHAPTER IV

USING HASH TABLES TO STORE EVOLUTIONARY TREES

As shown in Figure 13, the hash table is the core technology of our work. Based on the hashing technique, we store the evolutionary trees using bipartition information into hash tables, and use the information to solve two important biological problems. In this chapter, we introduce the basic notion of hash table and hash functions. We also describe the collisions and collision resolution methods. We explain how to store the evolutionary trees represented in the form of bipartitions into hash tables and how to convert the information for computing consensus tree and RF (Robinson-Foulds) distance matrix.

A. Hashing Technique

1. Hash Table

The main idea behind the *hash table* implementation is to store a set of $k = |S|$ elements in an array (the hash table) of length $m \geq k$. Hence, we require a function, $h(x)$ that maps any element x (also called the *hash key*) to an array location. This function is called a *hash function* h and the value $v = h(x)$ is called the *hash code* or *hash value* of x . That is, the element x gets stored at the array location $H[h(x)]$.

Figure 14 shows an example of a hash table for storing names and telephone numbers. In the hashing technique, those names are hash keys and a hash function takes the keys and produces a hash code. The hash code determines the location in the hash table where the related information to be stored. The location is called as bucket. The hash function in the example takes "Sam" and computes the hash code, "5". Then Sam's telephone number is inserted into the bucket at the location of "5".

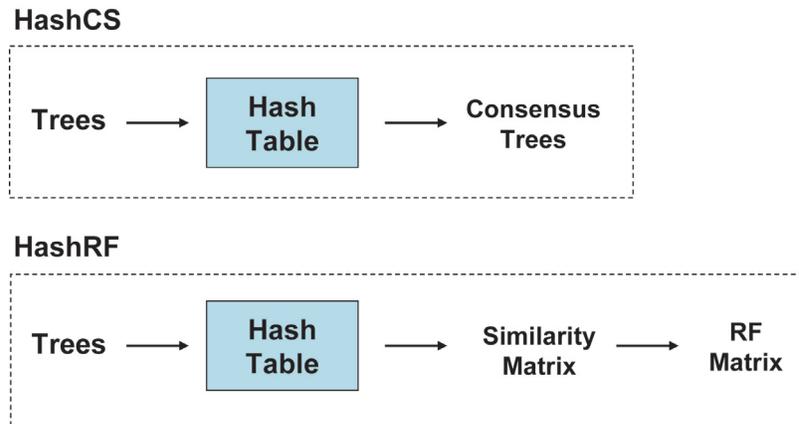


Fig. 13. An illustration of our HashCS and HashRF algorithms, whose core feature is based on the novel use of hash tables.

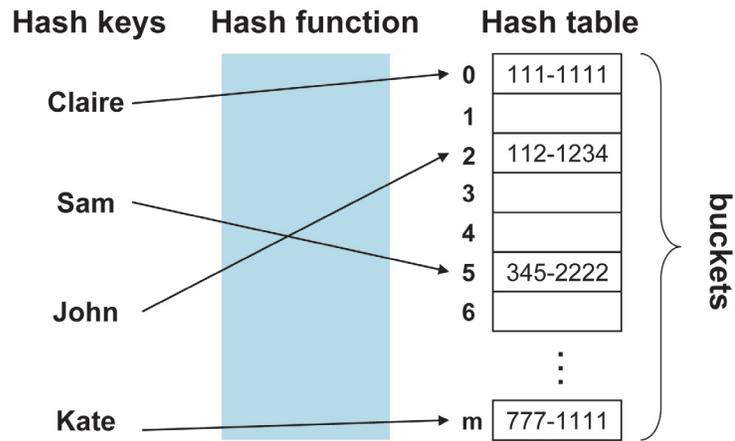


Fig. 14. The names are used as hash keys and the hash function computes the hash code for each key. Hash codes determines the bucket location to store the telephone number in the hash table.

Once the hash table is populated with telephone numbers related with names, one can find a person's telephone number by only one retrieval operation. For example, if the key is "Sam", $h("Sam")$ must return the hash table location "5". Accessing $H[h("Sam")]$ directly gets the telephone number.

Even though not every linear or binary search can be replaced by a hash table, there are many successful applications of hash tables [85, 86]. The above example is a simple implementation of personal information database system based on hash table. Compiler is a good example which extensively uses hash tables [87, 88]. A symbol table in an assembler, compiler, or interpreter is usually implemented as hash table. Each time an identifier is encountered as the source program is processed, we must find the record which contains the related information on the identifier. Kernels in operating systems also utilize hash table. For example, Linux, an open-source POSIX-compliant operating system, relies on hash tables to manage pages, buffers, inodes, and other kernel-level data objects [89, 90]. Linux performance depends on the efficiency and scalability of these tables. Hash tables are also used to detect errors caused by either hardware or software. Examples are TCP checksums, ECC memory, and MD5 checksums on downloaded files. In this case, the hash provides additional assurance that the data we received is correct. Finally, hashes are used to authenticate messages [91, 92]. In this case, we are trying to protect the original input from tampering, and we select a hash that is strong enough to make malicious attack infeasible or unprofitable.

Using hash table, the insertion of an item performs in $O(1)$ expected time. Delete and Search operations needs $O(n/m)$ (where n is the number of items in the table and m is the number of buckets). Ideally the operations' complexity approaches $O(1)$ when the items evenly spreads into the hash table and there are enough number of buckets compared with the number of items. Thus, the performance of a hash table is

closely dependent on the design of hash functions. Hence, hashing is a transformation of a set of input characters into a fixed length value or key that represents the original string. Hashing is used to index and find items in a table because it is faster to find the item using the shorter hashed key than to find it using the original value. By using hashing to store the key-and-value pair, ideally both the insertion and searching operations are $O(1)$ in the worst case. However, this kind of performance can only be achieved with complete *a priori* knowledge. We need to know beforehand specifically which items are to be inserted into a table. Unfortunately, we do not have this information in general. So, if we cannot guarantee $O(1)$ performance in the worst case, then we make it our design objective to achieve $O(1)$ performance in the average case.

2. Hash Function

As mentioned, a hash function h is a transformation that takes a variable-size input x and returns a fixed-size string, which is called the *hash value* v (that is, $v = h(m)$). Hash functions with this property have a variety of general computational uses, but hash functions are usually chosen to have some additional properties. Certainly the integer hash function is the most basic form of the hash function. The integer hash function transforms an integer hash key into an integer hash result. For a hash function, the distribution should be uniform. This implies when the hash result is used to calculate bucket location in the hash table, all buckets are equally likely to be picked. In addition, similar hash keys should be hashed to very different hash results. Ideally, a single bit change in the hash key should influence all bits of the hash result.

The implementation of a hashing function is divided into three categories. First, *perfect hashing* guarantees no collisions. A collision occurs when two different hash keys, x and y come to have the same hash code, or $h(x) == h(y)$. It is possible when

you know exactly what set of keys you are going to be hashing and design your hash function based on the set of keys. This method is popular for hashing keywords for compilers. Second, *static hashing* has fixed number of primary locations in the table. This location is called a bucket. Thus, when a bucket is full, an overflow bucket is needed to store any additional records that hash to that full bucket. This can be done with a linked list of overflow pages. Third, *dynamic hashing* enables the size of the table to grow with the number of collisions to accommodate new records and avoid long overflow linked chains.

3. Universal Hash Functions

Our hash-algorithms are implemented using universal hash functions. A universal hash function is a theoretical construct primarily used to show that an algorithm based on a hash function cannot be forced to have bad performance by an adversary. Bad performance in hashing comes from collisions, and a universal hash function guarantees that these cannot be forced to occur too often. The formal definition involves a set of keys, K , a set of values, V , and a family of hash functions, H which maps keys to values. Let $|V|$ denote size of V , the number of possible values. Then for all pairs of distinct keys x and y in K , H is a *2-universal family* of hash functions if

$$\text{Prob}_{h \in H}[h(x) = h(y)] \leq \frac{1}{|V|}. \quad (4.1)$$

More stringently, H is a *strongly 2-universal family* if for all pairs of distinct keys x and y in K , and for all pairs x' and y' in V ,

$$Prob_{h \in H}[h(x) = x' \text{ and } h(y) = y'] = \frac{1}{|V|^2}. \quad (4.2)$$

Every time a randomized algorithm runs, a set of random values are generated, which in turn is used in the hash functions to generate the hash values. If a hash function is based on the set of random numbers for computing a hash value, the behavior of the hash function comes to be non-deterministic.

4. Collisions

Given two distinct elements x_1 and x_2 , a *collision* occurs if $h(x_1) = h(x_2)$. For example, Figure 15 shows an example of collision. Hash functions have potential collisions, but with good hash functions they occur less often than with bad ones. In certain specialized applications where a relatively small number of possible inputs are all known ahead of time it is possible to construct a perfect hash function which maps all inputs to different outputs. But in a function which can take input of arbitrary length and content and returns a hash of a fixed length (such as MD5), there will always be collisions, because any given hash can correspond to an infinite number of possible inputs. When multiple lookup keys are mapped to identical indices, however, a hash collision occurs.

Ideally, one would be interested in a perfect hash function, which guarantees no collisions. However, this is only possible when the set of keys are known *a priori* (e.g., compiler keywords). Thus, most hash table implementations must explicitly handle collisions—especially since the performance of the underlying implementation is dependent upon the operations used to resolve the collision. There are two categories of collision resolution techniques [93]. First, closed addressing (or chaining) keeps a

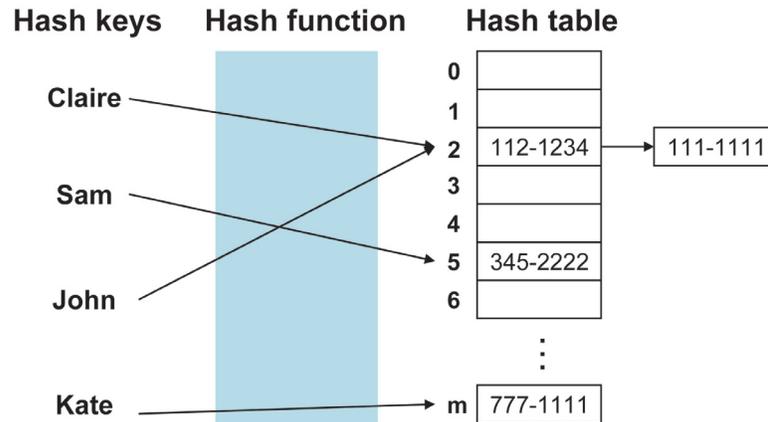


Fig. 15. An example of hash collision. The hash keys, "Claire" and "John" have the same hash value, "2". Thus, a collision occurs. In this example, the collision is resolved by chaining.

linked list for each location of the hash table. If two keys hash to the same location, both of them appear in the linked list for that location. Open addressing, on the other hand, folds the chaining lists back into the table. Here, a bucket location in a hash table, $H[i]$ is allowed to store only one value. When a collision occurs at location i , an empty table location is found in order to store one of the elements.

B. Evolutionary Trees and Hash Tables

1. Tree Bipartitions and their Representations

In a phylogenetic tree, modern organisms (or taxa) are placed at the leaves and ancestral organisms occupy internal nodes, with the edges of the tree denoting evolutionary relationships. Oftentimes, it is useful to represent phylogenies in terms of their *bipartitions*. Removing an edge e from a tree separates the leaves on one side from the leaves on the other. The division of the leaves into two subsets is the bipartition B_e associated with edge e . In Figure 16, two bipartitions, $AB|CDE$ and $ABC|DE$ are collected from tree T_1 by removing e_1 and e_2 , respectively. Likewise, tree T_2 has

$AB|CDE$ and $ABD|CE$. An evolutionary tree is uniquely and completely defined by its set of $O(n)$ bipartitions, where n is the number of taxa. A binary tree has exactly $n - 3$ bipartitions [94, 95].

For each tree in the collection of input trees, we find all of its bipartitions (internal edges) by performing a depth-first search. In order to process the bipartitions, we need some way to store them in the computer’s internal memory. Each bipartition in an input tree can be represented by:

- an n -bitstring, where n is the number of taxa;
- a k -bitstring, where $k < n$; or
- an implicit representation, which is an integer value, that is constructed by applying a hashing function to an n -bitstring.

Many algorithms use an n -bitstring representation. The PGM-Hashed approach uses a k -bitstring, and our HashCS and HashRF algorithms use an implicit representation.

2. n -bitstring Representation

An intuitive bitstring representation requires n bits, one for each taxon. The first bit is labeled by the first taxon name, the second bit is represented by the second taxon, etc. We can represent all of the taxa on one side of the tree with the bit ‘0’ and the remaining taxa on the side of the tree with the bit ‘1’. Consider the bipartition $AB|CDE$ from tree T_1 (see Figure 17). This bipartition would be represented as 11000, which means that taxa A and B are one side of the tree, and the remaining taxa are on the other side. Here, taxa on the same side of a bipartition as taxon A receive a ‘1’.

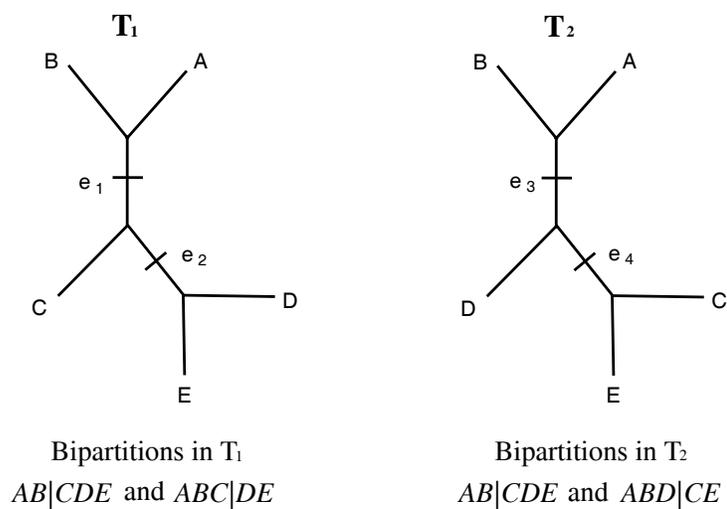


Fig. 16. An example of bipartitions from evolutionary trees.

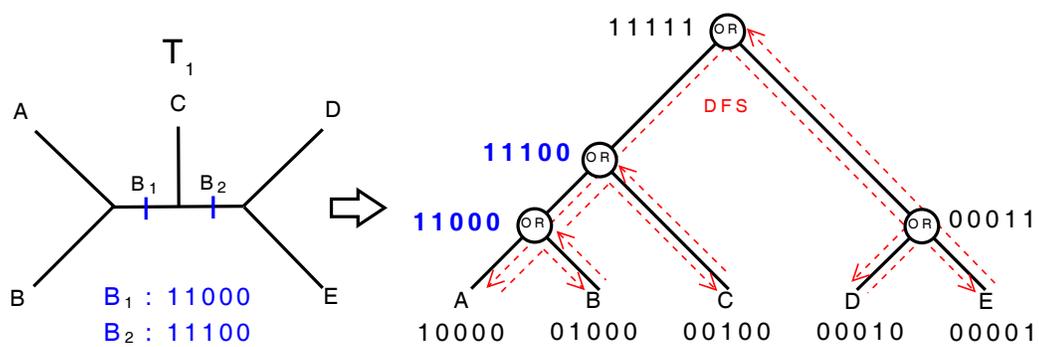


Fig. 17. n -bit bitstring representation of tree T_1 's bipartitions.

In order to perform a depth-first traversal in the collection of unrooted input trees, we arbitrarily root the tree. We find all of its bipartitions (internal edges) by performing a depth-first search traversal performing an *OR* operation to the bitstrings of an internal node's (parent's) children. Computing the *OR* between the two child bipartitions requires visiting each of the n columns of these two n -bitstrings. The *OR* operation is needed to keep the locations of "1"s while traversing and collecting bipartitions in depth-first search. In other words, if at least one of the bits in column j is a '1', then a '1' bit is produced for the column j in the bitstring representation of the parent. Figure 17 presents an example of n -bitstrings and Figure 18 shows an example of the hash table populated with 5-bit bitstrings.

The n -bitstring representation is intuitive way to collect and store bipartition information in "0/1" bitstring format. Also it is easy to make the taxon labels encoded in the bitstring. Our HashCS algorithm stores n -bitstrings because the label information is used to construct the final consensus trees. However, it costs more storage space and also consumes more CPU time for performing bitwise operations in hash value computation.

3. k -bitstring Representation

The size of the bitstring affects the algorithmic speed. As a result, the PGM-Hashed algorithm [15] uses a compressed k -bitstring. Each input taxon is represented by a random k -bitstring. Similarly to the n -bitstring case, all bipartitions are found by performing a depth-first search traversal of the tree. However, the bitstrings of an internal node's (parent's) children are *exclusive-OR*'ed together in the k -bitstring representation. Computing the *exclusive-OR* between two child bipartitions requires visiting each of the k columns of the two k -bitstrings. If the bits in column j are different (same), then a '1' ('0') bit is produced in column j of the parent node. For

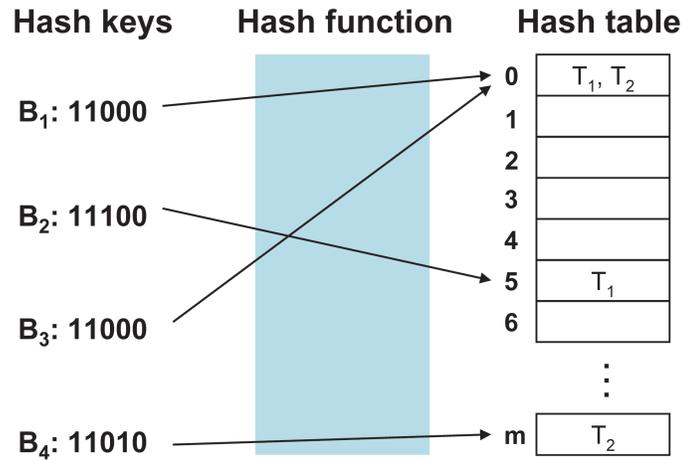


Fig. 18. The 5-bit bitstring bipartitions, B_1 and B_2 are collected from T_1 and B_3 and B_4 are collected from T_2 in Figure 4. B_1 and B_3 are identical thus those bipartitions have the same hash values and share the same location in the hash table.

each k -bitstring, the XOR operator minimizes the probability of having the same k -bitstring for different bipartitions. Unlike the case of n -bitstring in which we use the OR operator to maintain the locations of "1"s, k -bitstring needs random locations of "1"s.

There are several consequences of using a compressed bitstring to represent the bipartitions of an evolutionary tree. First, it is impossible to tell by looking at the k -bitstring representation of a bipartition what taxa are on different sides of the tree. Hence, it is impossible to construct consensus trees from k -bitstrings without additional information. Secondly, there is a possibility that two different bipartitions may in fact be represented by the same compressed bitstring. If this happens, then the resulting RF matrix will be incorrect. Pattengale et al. show that the probability of colliding compressed bitstrings decreases exponentially with the number of bits chosen for representing the bitstrings [15]. However, there are real-world examples when a k -bitstring representation will in fact produce colliding (i.e., the same) bitstring for

different bipartitions. In the implementation of the PGM-Hashed algorithm, $k = 64$. Our experimental results show that PGM-Hashed fails to operate on 567 taxa dataset consisting of 16,384 trees as a result of the high probability of colliding bipartitions represented by a k -bitstring.

4. Implicit Bipartition Representation

To represent bipartitions with implicit bipartition representation, we use universal hash functions. Similarly to Amenta et al. [71], we define our universal hashing functions as follows:

$$h_1(B) = \sum b_i r_i \bmod m_1. \quad (4.3)$$

$$h_2(B) = \sum b_i s_i \bmod m_2. \quad (4.4)$$

m_1 represents the number of entries (or locations) in the hash table ($m_1 > n \cdot t$, where n is the number of taxa and t is the number of trees). m_2 represent the largest bipartition ID (BID) that we can be given to a bipartition ($m_2 > c \cdot n \cdot t$, where c is a large constant). That is, instead of storing the n -bitstring, a shortened version of it (represented by the BID) will be stored in the hash table. Our implementations requires two sets of random integers r_i and s_i in the intervals $(0, \dots, m_1 - 1)$ and $(0, \dots, m_2 - 1)$, respectively. b_i represents the i th bit of the n -bitstring representation of the bipartition B . r_i and s_i are randomly generated every time when the algorithm runs and characterize our HashCS and HashRF algorithms as randomized algorithms. Figure 19 shows an example of n -bitstring and implicit bipartition representation based on the universal hash functions.

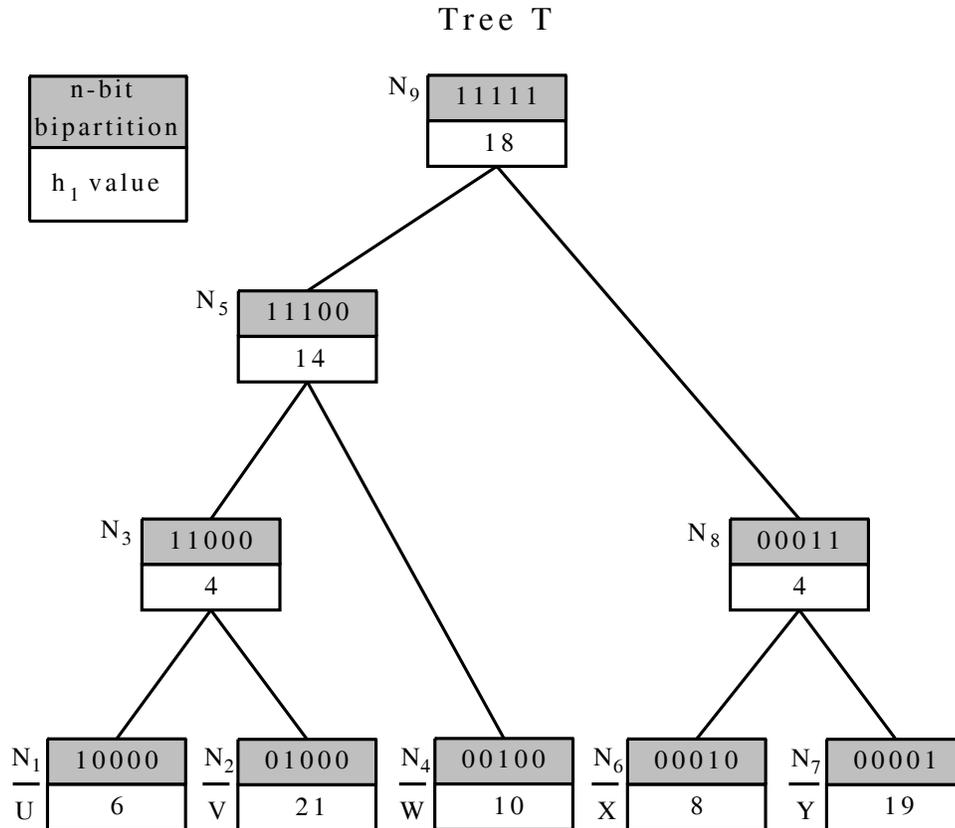


Fig. 19. Calculating the h_1 hash code from n -bitstring bipartitions representations based on a post-order traversal of an example tree T . Each node is labeled by N_i , where i is the order in which it was visited during the post-order traversal. The input taxa are U, V, W, X , and Y and are represented by nodes N_1, N_2, N_4, N_6 , and N_7 , respectively. To compute the h_1 hashing function, $m_1 = 23$ and the contents of array R are $r_0 = 6, r_1 = 21, r_2 = 10, r_3 = 8$, and $r_4 = 19$. Each node in the tree T shows an n -bitstring representation (shaded) along with the corresponding h_1 value (unshaded). The h_1 values of the nodes are computed based on Equations 2 and 5. For node N_5 , $h_1(11100) = (1 \cdot r_0 + 1 \cdot r_1 + 1 \cdot r_2 + 0 \cdot r_3 + 0 \cdot r_4) \bmod 23 = 14$.

a. Universal Hash Functions, h_1 and h_2

For faster performance, we actually avoid sending the n -bitstring and k -bitstring representations of each bipartition B to our hashing functions. Instead, we use an implicit bipartition representation to compute the hash functions quickly. An implicit bipartition is simply an integer value that provides the representation of the bipartition. Consider an internal node B whose bipartition is represented by a n -bitstring. Let the two children of this node have their n -bitstring representations labeled B_{left} and B_{right} , which represent disjoint sets of taxa. Then, the hash value for our h_1 hash function is

$$h_1(B) = \left(\sum_{B_{left}} b_i r_i \bmod m_1 \right) + \left(\sum_{B_{right}} b_i r_i \bmod m_1 \right). \quad (4.5)$$

We can use Equation 4.5 to compute implicit bipartitions, which replace the need for bitstrings in our hashing functions. Computing h_2 works similarly. The above equation is valid since B_{left} and B_{right} represent disjoint sets of taxa. We can use Equation 4.5 to compute implicit bipartitions, which replace the need for bitstrings in our hashing functions. An implicit representation is simply the hash code for a bipartition. Consider the h_1 hashing function. For each bipartition B , the h_1 hash values (implicit representations) for its two children B_{left} and B_{right} , have already been computed. They are $x = h_1(B_{left})$ and $y = h_1(B_{right})$. Thus, the implicit representation for node B is $h_1(B) = (x + y) \bmod m_1$. Computing the h_2 value implicitly for each bipartition works similarly.

Our HashRF algorithm is implemented only based on implicit bipartitions. However, our HashCS algorithm uses a combination of implicit and n -bitstring represen-

tations. The implicit representation is the only representation fed to the hashing functions. The n -bitstring is additional information that is collected from the first input tree to determine how the taxa should be grouped in the consensus tree and is used for constructing the resulting consensus trees.

Figure 19 provides an example of how to compute the h_1 hash code implicitly. Assume the set of random numbers $R = (6, 21, 10, 8, 19)$ and $m_1 = 23$. The h_1 value for node N_3 is computed from the n -bitstrings of its two children, N_1 and N_2 , resulting in $B_{left} = 10000$ and $B_{right} = 01000$. Therefore, $h_1(11000) = h_1(10000) + h_1(01000) = 4$. The implicit h_1 values are represented by the integer values (unshaded boxes). The hash code for the leaf node containing taxon i is r_i . These hash codes are propagated up the tree to compute the implicit representations of the parent bipartitions. Thus, to compute the h_1 hash code implicitly for N_3 , we calculate $(6 + 21) \bmod 23$, which is 4. Computing the h_2 hash code for representing a bipartition's ID (BID) works similarly.

b. Collision Types and Probability

A consequence of using hash functions is that bipartitions may end up residing in the same location in the hash table. Such an event is considered a collision, and there are three types to consider (see Table I). *Type 0* collisions are not regarded as a problematic condition. In our algorithms, this type of collision occurs when the same bipartition has already been populated into the hash table. In other words, *Type 0* collision notifies us it finds a shared bipartition among input trees. Even if it is called as a collision in this thesis, it is one of the reasons to make our algorithms faster and efficient.

Type 1 collisions result from two different bipartitions B_i and B_j (i.e., $B_i \neq B_j$)

Table I. Collision types in our randomized algorithms.

Collision Type	$B_i = B_j?$	$h_1(B_i) = h_1(B_j)?$	$h_2(B_i) = h_2(B_j)?$
Type 0	Yes	Yes	Yes
Type 1	No	Yes	No
Type 2	No	Yes	Yes

resided in the same location in the hash table. That is, $h_1(B_i) = h_1(B_j)$. Figure 20 shows two *Type 0* collisions between two same bipartition, B_1 and B_3 and a *Type 1* collision caused by B_4 . partition which is collided at location 4 in the hash table.

Type 2 (or double) collisions are serious and require a restart of the algorithm if such an event occurs. Otherwise, the resulting output will possibly be incorrect. Suppose that $B_i \neq B_j$. A *Type 2* collision occurs when two different bipartitions B_i and B_j hash to the same location in the hash table and the bipartition IDs (BIDs) associated with them are also the same. That is, $h_1(B_i) = h_1(B_j)$ and $h_2(B_i) = h_2(B_j)$. The probability of our randomized algorithms restarting because of a double collision among any pair of the bipartitions is $O(\frac{1}{c})$ [71]. The probability restarting because of a double collision among any pair of the bipartitions is $O(\frac{1}{c})$. Given that we can make c arbitrarily large, we do not explicitly check for *Type 2* collisions. Thus we have the following result.

Theorem 1. *Our HashCS and HashRF algorithms have a theoretical error rate of $O(\frac{1}{c})$, where c is an arbitrarily large constant number.*

Proof. Double collision occurs when for two bipartitions B_1, B_2 , we have $h_1(B_1) = h_1(B_2)$ and also $h_2(B_1) = h_2(B_2)$. Thus, given two different bipartitions, B_i and B_j , where $i \neq j$, the probability of $h_1(B_i)$ is equal to $h_1(B_j)$ is

$$Prob(h_1(B_i) = h_1(B_j)) = \frac{1}{m_1}, \text{ where } i \neq j. \quad (4.6)$$

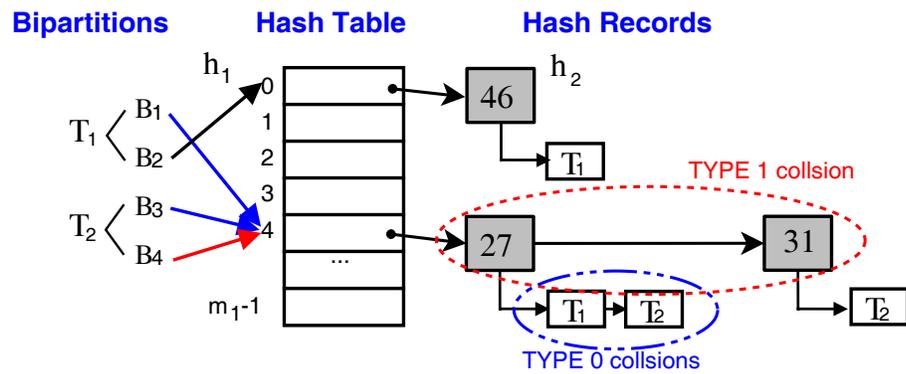


Fig. 20. An example of populated hash table with the bipartitions from trees in Figure 16. That is, B_1 and B_2 define T_1 , B_3 and B_4 are from T_2 , etc. The implicit representation of each bipartition is fed to the hash functions h_1 and h_2 . The shaded value in each hash record contains the bipartition ID (or h_2 value). Each bipartition ID has a linked list of tree indexes that share that particular bipartition. B_1 and B_3 are identical bipartition and causes TYPE 0 collisions. B_4 from T_2 is actually different bipartitions with B_1 and B_3 but hashed to the same location. It occurs TYPE 1 collision and thus the BID, "31" for the bipartition B_4 is chained with "27".

Similarly, the probability of $h_2(B_i)$ is equal to $h_2(B_j)$ is

$$\text{Prob}(h_2(B_i) = h_2(B_j)) = \frac{1}{m_2}, \text{ where } i \neq j. \quad (4.7)$$

Note that m_1 is a prime number which satisfies $m_1 \geq nt$ and m_2 is also a prime number which is bigger than cnt , where n is the number of taxa, t is the number of trees, and c is a constant integer value. Thus, the probability of $h_1(B_i)$ is equal to $h_1(B_j)$, and $h_2(B_i)$ is equal to $h_2(B_j)$ is

$$\text{Prob}(h_1(B_i) = h_1(B_j) \text{ and } h_2(B_i) = h_2(B_j)) = \frac{1}{m_1 m_2}, \text{ where } i \neq j. \quad (4.8)$$

The equation 4.8 is for a pair of bipartitions. Therefore, if there are t binary trees with n taxa, we have $(nt)^2$ pairs of bipartitions. Thus, for a given set of all bipartitions, B from the set of input trees, we can conclude the probability of double collisions for all pairs of bipartitions is

$$\begin{aligned} \forall B_i, B_j \in B, \text{ Prob}(h_1(B_i) = h_1(B_j) \text{ and } h_2(B_i) = h_2(B_j)) &= \frac{1}{m_1 m_2} \cdot (nt)^2 \\ &= \frac{1}{nt} \cdot \frac{1}{cnt} \cdot (nt)^2 \\ &= \frac{1}{c}. \end{aligned}$$

□

In practice, however, the error rate of HashCS and HashRF algorithms is much better. Our experiments with varying c from 1 to 10,000, and running HashCS at least

100 times for each c value, resulted in our algorithm producing the correct consensus tree every time for an overall error rate of 0%. Similarly, HashRF shows the same overall error rate as the verification result.

C. Deterministic and Randomized Algorithms

A deterministic algorithm is one that always behaves the same way given the same input. In other words, the input completely determines the sequence of computations performed by the algorithm. For example, the famous quicksort algorithm needs $O(n \log n)$ comparisons for sorting an array of items in the average case and requires $O(n^2)$ in the worst case. Assuming that the input array is not changed for every execution and is in the form of the worst case (i.e., the items are already sorted), it is clear that the complexity always remains $O(n^2)$ regardless of the number of executions.

On the other hand, if the sorting algorithm is shaped in a randomized approach which means it selects pivot elements uniformly at random, it has *higher probability* of having the complexity of $O(n \log n)$ time regardless of the characteristics of the input, even though the worst case complexity remains $O(n^2)$. A randomized (Monte Carlo) algorithm generates pseudo-random numbers each time the algorithm is executed and the logic of the algorithm is decided by the randomness. Randomized algorithms are particularly useful when faced with a malicious "adversary" who deliberately attempts to feed a bad input to the algorithm. By randomly modifying the behavior of the algorithm, it is quite difficult for an adversary to find a bad set of inputs that degrade the performance of the algorithm. The other advantage is that randomized algorithm is fast and simple. For problems in which deterministic algorithms are too slow or even not feasible, but randomized algorithms can give us good results with

high probability.

Our randomized algorithms, HashCS and HashRF incorporate hash tables, which are data structures that associate keys (e.g., evolutionary relationships contained in a tree) with values (e.g., tree identities) and make our algorithms less sensitive to bad inputs. Hence, the randomness employed in our approach is related to how we define our hash functions. Furthermore, by leveraging randomization, our algorithms are quite fast. For n taxa (or species) and t trees, our HashCS and HashRF algorithms run in $O(nt)$ and $O(nt^2)$ time, respectively. One of the main disadvantages of randomized algorithm is the possibility that an incorrect solution will occur. However, a well-designed randomized algorithm will have a very high probability of returning a correct answer. For our algorithms, there is a $O(\frac{1}{c})$ chance that the algorithms will return the incorrect consensus tree or RF matrix. Hence, with large c , we can control the level of desired accuracy of our algorithms.

D. Summary

Our HashCS and HashRF implementations are built upon the hash technique. Our HashCS algorithm reprocess the hash table for computing consensus trees, where as our HashRF used the table for compute RF distance. Hash tables provides not only fast average access time ($O(1)$), also works as a compressed repository of evolutionary trees. In this chapter we introduce the basic concept of hash table and hash functions based on general examples. Hashing should always deal with the possibility of collisions. If there is a perfect hash function, It is no need to mention about the collisions but perfect hash functions are very difficult to design and implement. Among collision resolution methods, our hash table is implemented by chaining which means collided items are chained together in the same hash table location.

To store evolutionary trees into hash tables, we need effective representation of the trees. Bipartitions are basic components in phylogenetic trees collected from removing each internal edges. By storing bipartitions, evolutionary trees can be stored. We introduced three types of bipartition representation: n -bitstring, k -bitstring, and implicit. Our HashCS algorithm uses both implicit and n -bitstring representation for storing bipartitions and for storing information to construct the resulting consensus tree, respectively. Our HashRF only uses implicit bipartitions. We briefly describe how to construct consensus trees and how to compute RF distance matrix from the hash tables. Further, the probability of error is discussed.

Again, the hash table is the core technology of our work. The details of our HashCS and HashRF algorithms are introduced in Chapter V and Chapter VI. We also discussed the possible future extensions based on our current hash technique in Chapter VIII.

CHAPTER V

HASHCS: COMPUTING THE CONSENSUS TREES

Our first hash-based randomized algorithm, HashCS is described in this chapter. HashCS is based on the hash table using universal hash functions (see Chapter IV). Figure 13 illustrates the major flow of our algorithm. The main part is collecting necessary information from the input evolutionary trees and storing the information in the hash table. In this implementation, the collected information is both bipartitions and the frequency of each bipartition. After collecting the information, the hash table is reprocessed to produce consensus trees. Using biological and artificial tree collections, we do the performance analysis on our implementation.

A. Motivation

Given our interest in consensus trees, our research question is: *How to design efficient algorithms for large-scale consensus analysis?* Phylogenetic methods (such as Bayesian analysis) to reconstruct an evolutionary tree can easily produce tens of thousands of potential trees that must be summarized in order to understand the evolutionary relationships among the taxa. Moreover, large tree collections can also be produced by bootstrap tests on phylogenies to assess the uncertainty of a phylogenetic estimate [48, 49, 50]. Currently, constructing majority or strict consensus trees of them is a popular way to analyze those trees and biologists use popular phylogenetic software packages such as PAUP*, MrBayes, and TNT summarize their large tree collections into a single consensus tree.

Advanced techniques in reconstructing phylogenetic trees and faster processors produce more large collection of trees. However, current tools for constructing consensus tree can not accommodate the growing requirements of larger phylogenetic

analysis, such as those necessary for building the *Tree of Life*, the grand challenge problem in phylogenetics. Further, some only support binary trees as input for computing consensus trees. We designed and implemented HashCS based on hashing technique to provide researchers with more efficient and convenient tools for constructing consensus trees.

Consensus trees summarize the information of a collection of trees into a single output tree. Bryant provides an excellent survey of different consensus techniques [68]. In our work, we consider the most popular consensus approaches: strict and majority consensus trees. The strict consensus tree contains bipartitions that appear in all of the input trees. To appear in the majority tree, a bipartition must appear in more than half of the input trees. In Figure 21, no evolutionary relationship (bipartition) in the tree collection appears in all four trees. Hence, the resulting strict consensus tree is completely unresolved, which is represented as a star tree. The majority consensus tree consists of only the bipartition $AB|CDE$.

Oftentimes, a consensus tree will not be binary since there will be bipartitions that are not shared across the tree collection. One way to measure the quality of a consensus tree is its *resolution rate*, which represents the percentage of the tree that is binary. The resolution rate of a tree T is $\frac{b}{n-3}$, where b is the number of bipartitions in the tree T and $n - 3$ is the number of possible resolved bipartitions. Consider the majority consensus tree in Figure 21 where the number of taxa, n , is 5. This majority tree consists of a single bipartition, but the total number of possible resolved bipartitions is 2. Hence, the resolution rate for this tree is 50%. The strict tree in Figure 21 has a resolution rate of 0%. Overall, the resolution rate of a tree T varies between 0% (a star) and 100% (completely resolved binary tree).

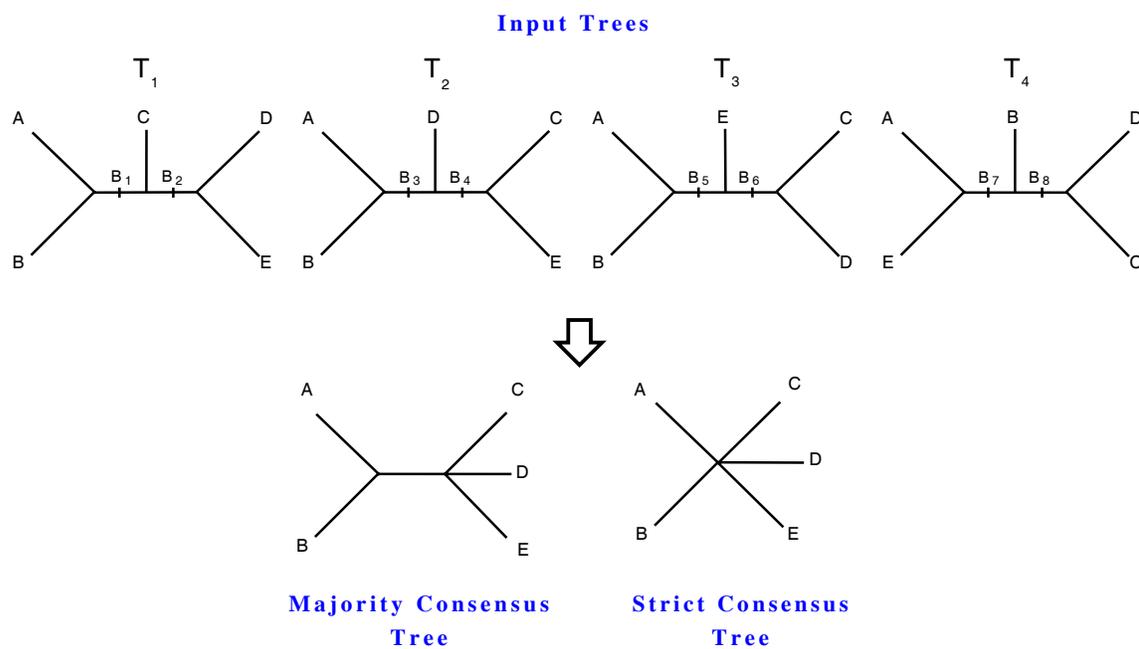


Fig. 21. Overview of the consensus tree techniques of interest. The example collection consists of four trees: T_1 , T_2 , T_3 , and T_4 . Bipartitions (or internal edges) in a tree are labeled B_i , where i ranges from 1 to 8. The majority tree consists of those bipartitions that appear in over 50% the trees. The strict consensus tree, on the other hand, consists of those bipartitions that appear in all four trees.

B. HashCS Algorithm Description

Our HashCS [96, 97] is based on our *novel* use of hash tables. It consists of 2 major steps: (i) populate hash table with collected bipartitions and (ii) compute consensus tree from the bipartition information in a hash table.

1. Step 1: Populating the Hash Table

Figure 22 provides an overview of this step of the HashCS algorithm. Algorithm V.1 and Algorithm V.1 introduce the pseudocode for the HashCS algorithm in strict and majority cases, respectively. We assume that input phylogenetic trees are stored in a file using Newick format which is a standard format for representing phylogenetic trees [79]. As each input tree, T_i is traversed in post-order, where the representation of the bipartition is fed through two hash functions, h_1 and h_2 . Hash function h_1 is used to generate the location needed for storing a bipartition in the hash table. h_2 is responsible for creating bipartition identifiers (BID). The hash table record for bipartition B_i consist of B_i 's BID and frequency. For each unique BID, the bipartition frequency counter is set to 1. Identical (shared) bipartitions from other trees in the collection result in incrementing the bipartition frequency counter by one. Details on the hash functions and bipartition population into the hash table are described in Chapter IV.

We use two different insertion policies depending on the type of consensus tree constructed. For the strict consensus tree, a bipartition must appear in all t trees in the tree collection. Since the first tree in the collection determines the possible set of strict consensus bipartitions, only the first tree's bipartitions are inserted into the hash table. For the last tree, the n -bitstring representation for each bipartition is computed along with the implicit representation. However, the n -bit representation is

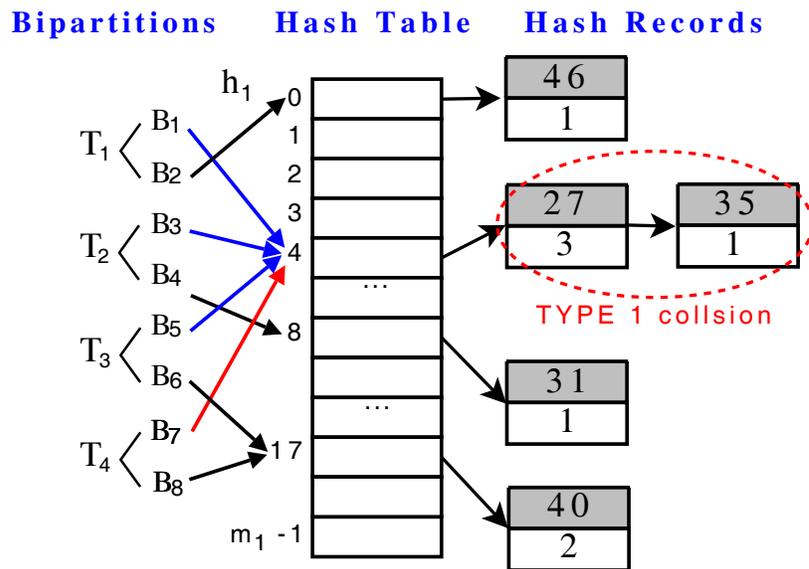


Fig. 22. Overview of the HashCS algorithm. Bipartitions are from Figure 4. That is, B_1 and B_2 define T_1 , B_3 and B_4 are from T_2 , etc. The implicit representation of each bipartition is fed to the hash functions h_1 and h_2 . The shaded value in each hash record contains the bipartition ID (or h_2 value) whereas the unshaded value shows the frequency of that bipartition.

Algorithm 1 HashCS algorithm for computing strict consensus tree.

Require: A set of trees, T_1, T_2, \dots, T_t , where $t \geq 2$

```

1: for  $i = 1$  to  $t$  do
2:   if  $i == 1$  then
3:     Traverse tree  $T_i$  in post order and find  $B_j$ 
4:     Insert  $B_j$  into the hash table  $H[h_1(B_j)]$  and set  $freq(B_j) = 0$ 
5:   else if  $1 < i < t - 1$  then
6:     Traverse tree  $T_i$  in post order and find  $B_j$ 
7:     for all bipartition  $B_j \in T_i$  do
8:       Compute  $h_1(B_j)$  and  $h_2(B_j)$ , implicitly.
9:       Check the BID list at the hash table  $H[h_1(B_j)]$  for  $h_2(B_j)$ .
10:      if  $h_2(B_j)$  exists then
11:        Increment the frequency of  $h_2(B_j)$ 
12:      end if
13:    end for
14:   else
15:     Traverse tree  $T_t$  in post order, find  $B_j$ , and collect  $n$ -bitstring  $BS_j$ 
16:     for all bipartition  $B_j \in T_t$  do
17:       Compute  $h_1(B_j)$  and  $h_2(B_j)$ , implicitly.
18:       Check the BID list at the hash table  $H[h_1(B_j)]$  for  $h_2(B_j)$ .
19:       if  $h_2(B_j)$  exists and  $freq(h_2(B_j)) == t - 1$  then
20:         Insert  $BS_j$  into the bipartition list,  $bl$ 
21:       end if
22:     end for
23:   end if
24: end for
25: Construct the strict consensus tree from the bipartitions in  $bl$ 

```

Algorithm 2 HashCS algorithm for computing majority consensus tree.

Require: A set of trees, T_1, T_2, \dots, T_t , where $t \geq 2$

```

1: for  $i = 1$  to  $t$  do
2:   if  $i \leq \lfloor \frac{t}{2} \rfloor + 1$  then
3:     Traverse tree  $T_i$  in post order and find  $B_j$ 
4:     if  $h_2(B_j)$  exists then
5:       Increment the frequency of  $h_2(B_j)$ 
6:     else
7:       Insert  $B_j$  into the hash table  $H[h_1(B_j)]$  and set  $freq(B_j) = 0$ 
8:     end if
9:   else
10:    Traverse tree  $T_t$  in post order, find  $B_j$ , and collect  $n$ -bitstring  $BS_j$ 
11:    for all bipartition  $B_j \in T_i$  do
12:      Compute  $h_1(B_j)$  and  $h_2(B_j)$ , implicitly.
13:      Check the BID list at the hash table  $H[h_1(B_j)]$  for  $h_2(B_j)$ .
14:      if  $h_2(B_j)$  exists and  $freq(h_2(B_j)) < \lfloor \frac{t}{2} \rfloor$  then
15:        Increment the frequency of  $h_2(B_j)$ 
16:      else if  $h_2(B_j)$  exists and  $freq(h_2(B_j)) == \lfloor \frac{t}{2} \rfloor$  then
17:        Insert  $BS_j$  into the bipartition list,  $bl$ 
18:        Invalidate  $h_2(B_j)$ 
19:      end if
20:    end for
21:  end if
22: end for
23: Construct the majority consensus tree from the bipartitions in  $bl$ 

```

not feed to the h_1 and h_2 hash functions. Instead, they are stored into an array if the bipartition if the frequency count for that bipartition is t . The array of n -bitstring representations will be used to build the consensus tree in Step 2 of the HashCS algorithm.

To construct the majority consensus tree, all unique bipartitions are inserted into the hash table until tree $\lfloor \frac{t}{2} \rfloor + 1$ is read. At this time, the n -bitstrings are computed along with the implicit bipartitions. Similarly to the insertion policy for constructing the strict consensus, the n -bit representation is not feed to the hash functions. Instead, they are stored into an array if the bipartition they represent has a frequency of $\lfloor \frac{t}{2} \rfloor + 1$ in the hash table. For our majority algorithm, once a node's bipartition frequency has reached $\lfloor \frac{t}{2} \rfloor + 1$, it is invalidated so that its resulting n -bit representation doesn't appear multiple times in the array. During Step 2 of the algorithm, this array of n -bitstrings will be used to build the majority tree.

2. Step 2: Constructing the Consensus Tree

Initially, the consensus tree is a star tree of n taxa. Bipartitions are added to refine the consensus tree based on the number of 1's in its n -bitstring representation. (The number of 0's could have been used as well.) The more 1's in the bitstring representation, the more taxa that are grouped together by this bipartition. A star tree is an n -bitstring representation of all 1's. During the collection of n -bitstrings in Step 1, a count of the number of 1's was stored for each bipartition. In step 2, these counts are then sorted in increasing order, which means that the bipartitions that groups together that most taxa appears first. The bipartition that groups together the fewest taxa appears last in the sorted list of '1' bit counts.

For each bipartition, a new internal node in the consensus tree is created. Hence, the bipartition is scanned to put the taxa into two groups—taxa with '0' bits compose

one group and those with '1' bits compose the other group. The, the taxa indicated by the '1' bits become children of the new internal node. The above process repeats until all bipartitions in the sorted list are added to the consensus tree.

C. Theoretical Analysis

Our analysis assumes that the number of trees, t is much greater than the number of taxa, n . We believe this assumption is especially valid for trees obtained from a Bayesian analysis, which can sample trees from runs consisting of well over a million generations. Based on the assumption, Step 1 requires $O(nt)$ time. A similar analysis can be done for the HashCS majority algorithm resulting also in an $O(nt)$ time for the first step. Step 2 requires $O(nx)$ time to construct the consensus tree from the n -bitstring bipartitions, where x is the number of bipartitions in the consensus tree. In the worst case, $x = n - 3$, the maximum number of edges in a binary tree. Thus, the overall running time for HashCS is $O(nt)$.

D. Experimental Analysis

1. Motivation

Experimental algorithmics [98, 99, 100, 101] combines algorithmic work and experimentation. After designed and implemented, algorithms should be tested and analyzed on a variety of instances. Designing an algorithm is just a beginning of developing robust and efficient software for applications. Hence, we wondered *how does the performance of our HashCS algorithm compare to the existing approaches?* In this section, we analyze the running performance of our HashCS algorithm using variety of tree collections which are not only collected from biological trees but also generated to inspect the behavioral aspects of algorithms in detail.

2. Phylogenetic Tree Collections

a. Biological Tree Collections

The biological trees used in this study were obtained from three recent Bayesian analysis, which we describe below.

1. 8,000 trees obtained from an analysis of a 14,085-bp DNA alignment from 19 nuclear gene segments for 16 euarchontoglires [102]. Two independent runs of MrBayes were performed with 4 independent chains, using the GTR+I+ Γ model, sampled every 1,000th generation for 5 million generations, which resulted in 10,000 total trees. However, the authors discarded the first 1 million generations (1,000 trees) as burn-in from each of the two runs. Thus, producing the collection of 8,000 trees used in our experiments.
2. 20,000 trees obtained from a Bayesian analysis of an alignment of 150 taxa (23 desert taxa and 127 others from freshwater, marine, and soil habitats) with 1,651 aligned sites [103]. Two independent runs consisting of 25 million generations (trees were sampled every 1,000 generations) were performed using the GTR+I+ Γ model in MrBayes with four independent chains. The authors constructed a majority consensus tree in their study using the 20,000 trees from the last 10 million generations from each of the two runs.
3. 33,306 trees obtained from an analysis of a three-gene, 567 taxa (560 angiosperms, seven outgroups) dataset with 4,621 aligned characters, which is one of the largest Bayesian analysis done to date [104]. Twelve runs, with four chains each, using the GTR+I+ Γ model in MrBayes ran for at least 10 million generations. Trees were sampled every 1,000 generations. The authors discuss the difficulties with combining trees from multiple runs. As a result, they de-

cided to combine the trees from generations 7,372,000 – 10,160,000 that had a likelihood score of at least -238,050 to produce the majority consensus tree and posterior probabilities. Only trees from 7 of the 12 runs fit this criteria. For our experiments, in order to use the data from all 12 runs, the trees from the first 8 million generations are discarded. This resulted in the 33,306 trees considered in our experiments.

In our experiments, for each (n, t) pair, t trees with n taxa were randomly sampled without replacement from the appropriate tree collection, where $n = 150$ and 567 , and t is $128, 256, 512, \dots, 16384$ trees. For each (n, t) pair, we repeated the above sampling process five times. Our experimental results show the average algorithmic performance for each (n, t) pair.

The resolution rate of a consensus tree is the percentage of bipartitions that are resolved in the consensus tree. A low resolution rate implies that there is large disagreement among the evolutionary relationships depicted in the phylogenetic trees. A high resolution rate implies that the input trees agree on a large number of evolutionary relationships. A tree that is 0% resolved (i.e., a star) implies that all of the bipartitions in the input trees are unique. A tree that 100% resolved implies that the input trees are identical. The resolution rate impacts the performance of algorithms for computing consensus trees and topological distances. Our artificial trees are generated with varying the resolution rates.

b. Artificial Tree Collections

We generate artificial tree collections to show how the performance of the algorithms would scale across different taxa sizes (i.e., $n = 128, 256, 512$, and 1024) represented by our biological tree sets. Our generation of artificial trees is based on using majority

consensus to create the collection of t trees, where $t = 16,384$. Our biological trees have a very high majority resolution rate (up to 93%), which means that the trees are highly similar. During our artificial generation of trees, majority consensus tree resolution rate, r , varies between 0%, 25%, 50%, 75%, and 100%. If the majority consensus resolution of t trees is 0%, it essentially means that the trees have very little bipartition sharing and not very similar. A resolution rate of 100% represents extremely high bipartition sharing among the trees resulting in very similar trees.

To create our artificial tree collections, we used `apTreeshape` [105]—a R package for the simulation and analysis of phylogenetic tree topologies—to create a random, Yule model tree consisting of n taxa. Next, we transform this tree into a $r\%$ resolved multifurcating tree by randomly removing bipartitions. We take our $r\%$ resolved tree and use it to generate t input trees. The resolved tree represents the majority consensus tree, $T_{r\%}$, of interest. Each bipartition in tree $T_{r\%}$ is given a weight in the interval $(50\%, 100\%]$, which represents the percentage of the t trees that have that bipartition (see Figure 23).

Once all of the bipartitions from the majority tree $T_{r\%}$ have been distributed, each of the t trees is constructed. For each tree T_i , where $1 \leq i \leq t$, we construct tree T_i with the bipartitions that have been distributed to it. After the construction, any remaining multifurcating nodes are randomly resolved into binary nodes. These randomly resolved bipartitions (non-majority bipartitions) are then distributed to $\lfloor p \rfloor$ trees, where $1 < p \leq 0.50t$ and $i < p \leq t$. We distribute the non-majority bipartitions to the remaining trees in order to increase the amount of sharing among them in the tree collection. The above process is repeated five times for each n, t , and r . Thus, our plots show the average performance on our artificial datasets.

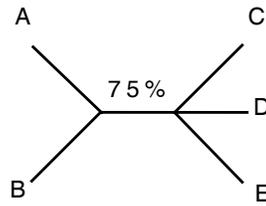


Fig. 23. Majority consensus tree for the input trees shown in Figure 4. The bipartition weight implies that 75% of the t trees must have the bipartition $AB|CDE$.

3. Implementations and Platform

HashCS and Day's algorithm were written in C++ and compiled with gcc 4.2.4 with the `-O3` compiler option. PAUP* is commercially available software and we used version 4.0b10 in our experiments. Phylip and MrBayes are freely available and we used software versions 3.65 and 3.1.2, respectively. All experiments were run on an Intel Pentium platform with a 3.0GHz processor and a total of 2GB of memory. We also used the Linux operating system (Red Hat 2.5.22.14-17.fc6).

4. Experimental Results

We compare our HashCS algorithms to four different approaches: MrBayes, Phylip, PAUP*, and Day's algorithm. Day's algorithm only computes the strict consensus tree as it cannot be extended to compute majority trees. We do not explicitly compare the algorithms to TNT it has been shown that PAUP* is much faster than TNT in constructing strict consensus trees [7, 72, 73]. For the experiments with our artificial tree collections, HashCS is compared with PAUP*, the closest competitor. We use $c = 1,000$ for HashCS and each plot shows the average performance over five runs.

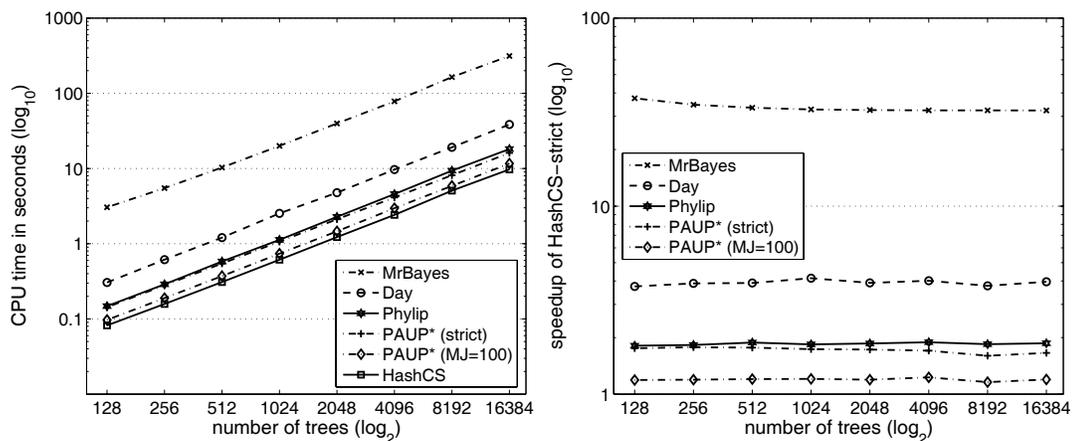
a. Performance on Biological Trees

First, we consider the running time of our HashCS algorithm against its competitors for computing the strict consensus tree. Figures 24(a) and 24(c) show the results. Overall, HashCS is the fastest algorithm for computing strict consensus trees. MrBayes is the slowest approach requiring 1.1 hours compared to 38.3 seconds by HashCS on the largest dataset (567 taxa, 16384 trees). Surprisingly, Day’s algorithm is not the fastest approach in practice even though it is optimal from a theoretical complexity standpoint.

In our plots for Figure 24, we show two performance results for PAUP*. PAUP*(strict) is the running time of the algorithm when using the strict command in the Nexus file. PAUP*(MJ=100) computes the strict consensus tree, but using the majority option with percent equal to 100%. Surprisingly, using the strict option takes considerably more time to compute the same tree than using the majority option. The work of Boyer et al. observed the same behavior [7, 72]. In any case, PAUP*(MJ=100) is the second faster performer behind HashCS.

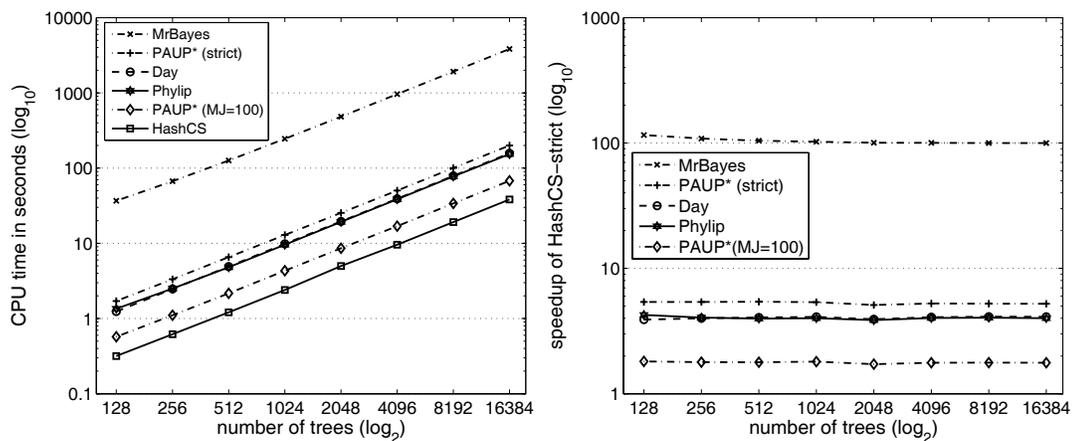
Figures 24(b) and 24(d) shows the speedup of the HashCS algorithm in comparison to the other approaches. Speedup is calculated as $\frac{x}{y}$, where x is the execution time required by Day, MrBayes, PAUP*, or Phylip and y is the running time of HashCS. On the largest dataset, HashCS is over 100 times faster than MrBayes and approximately 4 and 1.8 times faster than Phylip and PAUP*, respectively. Figure 25 provides a closer look at the speedup of the HashCS algorithm over its top competitor for constructing the strict consensus tree. The plot clearly shows that the speedup of HashCS is unaffected by the size of the tree collection. Instead, the performance of HashCS improves significantly with increasing number of taxa.

For majority trees, Figures 26 (a) and 26(c) clearly demonstrate HashCS and



(a) 150 taxa (time)

(b) 150 taxa (speedup)



(c) 567 taxa (time)

(d) 567 taxa (speedup)

Fig. 24. Running time and speedup of the strict consensus tree algorithms. The scale of the y-axis is different for all the plots.

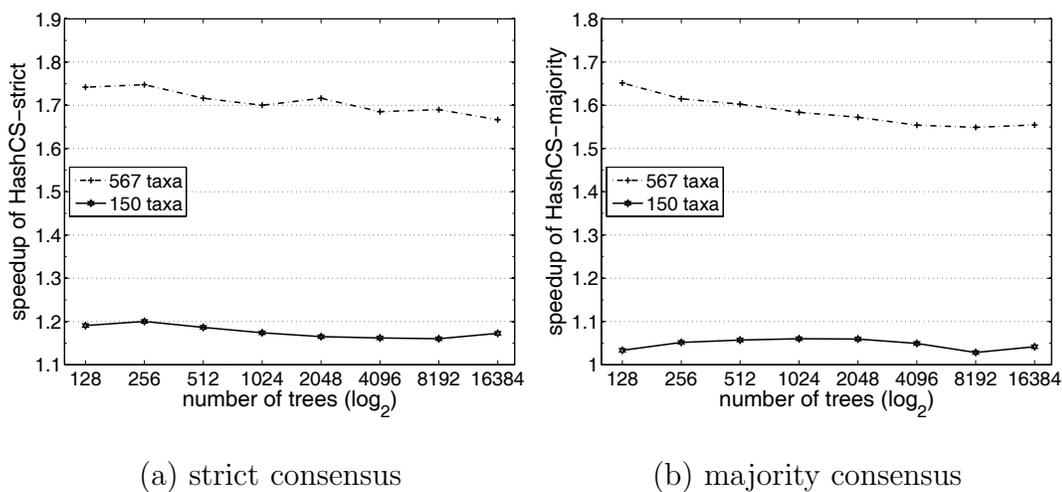


Fig. 25. Speedup of the HashCS algorithm over PAUP*, its top consensus tree competitor. (a) Speedup of HashCS over PAUP* to compute the strict consensus tree. (b) Speedup of HashCS over PAUP* to compute the majority tree. The scale of the y-axis is different for all the plots.

MrBayes are the fastest and slowest algorithms, respectively. The speedup of the HashCS algorithm in comparison to the other majority approaches is shown in Figures 26(b) and 26(d). The speedup of HashCS is over 1.7 and 1.6 for computing strict and majority consensus trees on 567 taxa trees, respectively. Again, HashCS's performance increases significantly with larger taxa sizes. However, its performance is essentially unaffected by the number of trees in the collection (see Figure 25(b)). For 567 taxa, HashCS computes a majority tree at least 60% faster than PAUP*, its nearest competitor.

b. Performance on Artificial Trees

The previous figures clearly demonstrate that MrBayes, Phylip, and Day's algorithm are not competitive as HashCS and PAUP*. So, we don't consider the slower implementations any further.

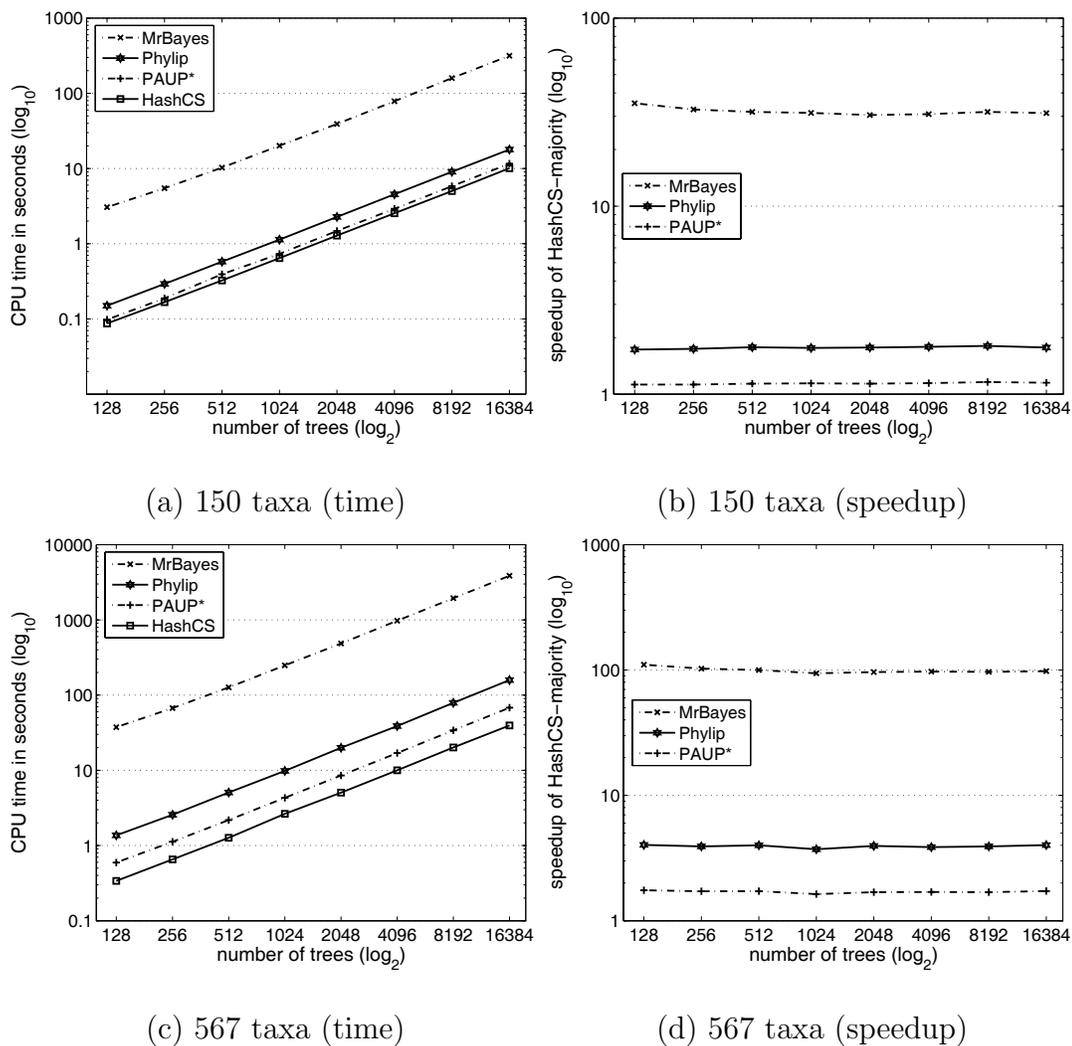


Fig. 26. Running time and speedup of the majority consensus tree algorithms. The scale of the y-axis is different for all the plots.

Figure 27 shows the running time of HashCS and PAUP* to compute the majority or strict consensus tree on our artificial tree collections. The plots clearly show that PAUP*'s running time is impacted by the amount of bipartition sharing in the collection of trees. For example, in Figure 27(a), PAUP* requires approximately 30,000 seconds (or 8.3 hours) if the resulting consensus tree of 16,384 input trees is 0% resolved. However, Figure 27(c) shows that PAUP* executes much faster (around 180 seconds) if the 16,384 input trees result in a 100% resolved majority tree. Hence, PAUP* computes the majority tree faster as the input trees become more similar. HashCS, on the other hand, requires under 100 seconds to compute the majority tree irrespective of the bipartition distribution of the 16,384 input trees.

Figure 28 shows the resulting speedup of HashCS over PAUP*. Since PAUP*'s performance varies as a function of the amount of shared bipartitions among the input trees, the speedup varies as well. PAUP* performs the worst when constructing a 0% consensus tree resulting in HashCS being over 300 times faster than PAUP* for $n = 1,024$. The speedup gap closes as the 16,384 input trees become more similar. Hence, when the consensus tree is 100% resolved, then HashCS is up to 2.3 times faster than PAUP*.

E. Summary

We introduced the HashCS algorithm as a fast technique for summarizing evolutionary relationships contained in a set of phylogenetic trees. The novelty of our approach is our use of hash table, which provides a convenient and fast approach to store and access the bipartition information collected from the tree collections. Using our collections of biological and artificial trees, we shown that our hash-based approach significantly improves the performance of computing consensus trees.

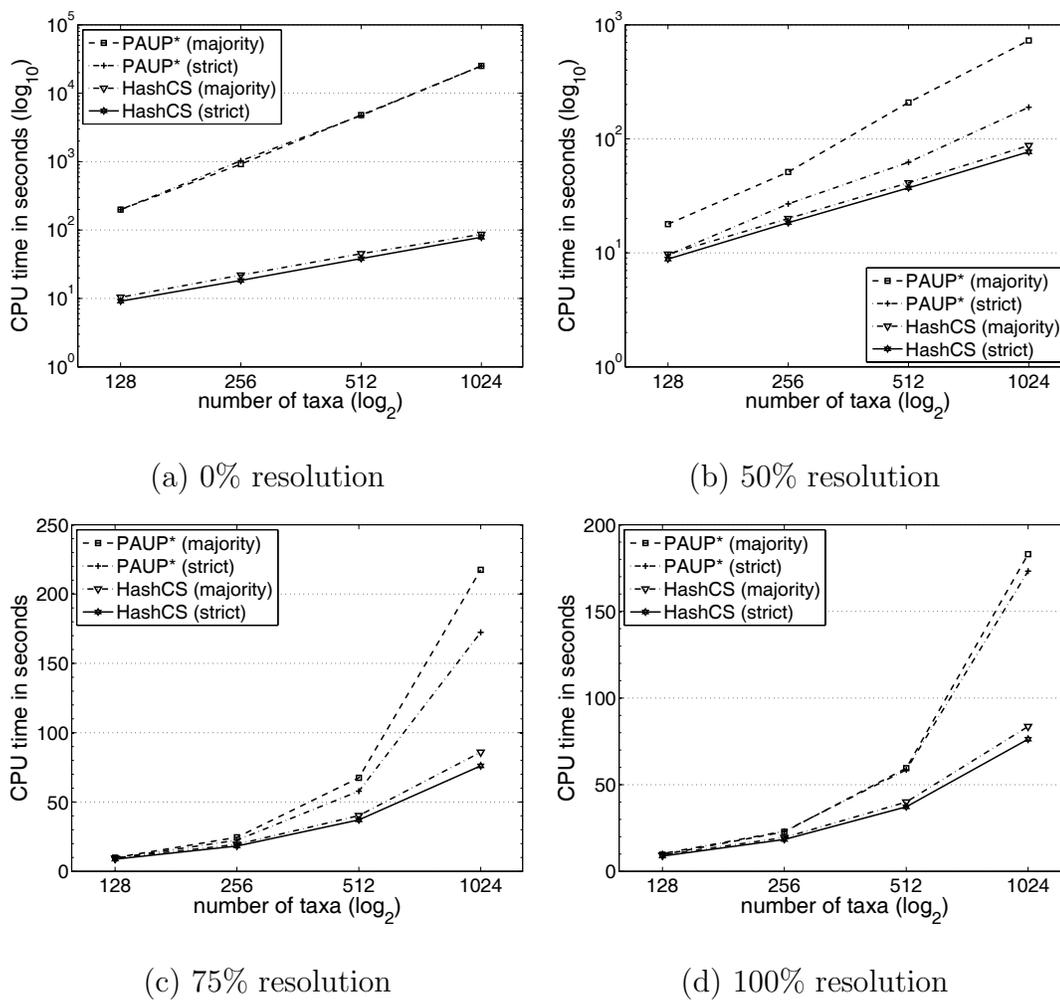
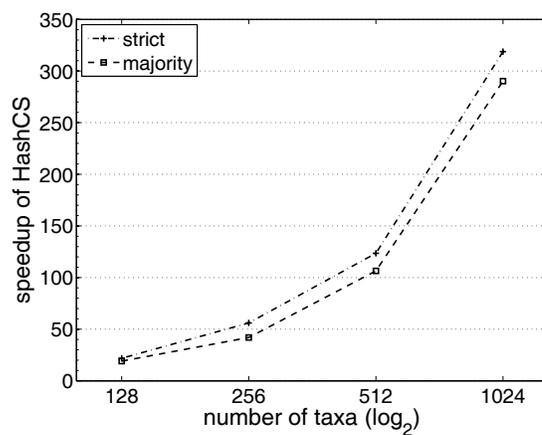
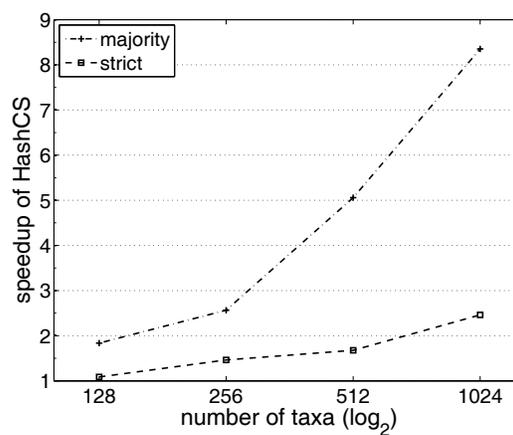


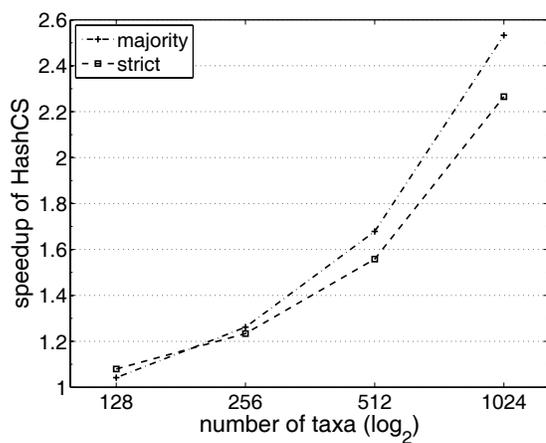
Fig. 27. Running time of the consensus tree algorithms on our artificial tree collections. The scale of the y-axis is different for all the plots.



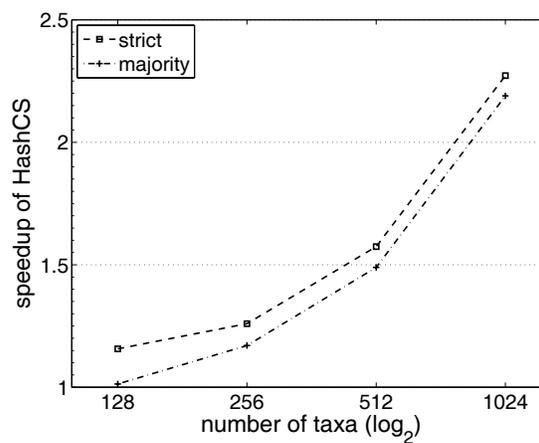
(a) 0% resolution



(b) 50% resolution



(c) 75% resolution



(d) 100% resolution

Fig. 28. Speedup of the consensus tree algorithms on our artificial tree collections. The scale of the y-axis is different for all the plots.

We provided extensive experimentation regarding testing the practical performance of our HashCS algorithm across a diverse collection of biological and artificial trees. For biological trees of 150 and 567 taxa, HashCS is up to 80% faster than PAUP*, the second fastest consensus algorithm.

Next, we showed performance results on artificial trees. For computing consensus trees, PAUP* gets slower when the amount of shared bipartitions is decreased. PAUP* performs the worst when constructing a 0% consensus tree resulting in HashCS being over 300 times faster than PAUP* for $n = 1,024$. The speedup gap between HashCS and PAUP* closes as the 16,384 input trees become more similar. When the consensus tree is 100% resolved, then HashCS is up to 2.3 times faster than PAUP*.

Our experimental results show that our HashCS implementation, which is based on Amenta et al.'s work [71], is the fastest approach for building large-scale consensus trees. Fast algorithms such as HashCS make it feasible to study the relationships contained in a collection of trees in a reasonable amount of time. Much deserved attention has been placed on speeding up phylogenetic search heuristics (such as maximum likelihood, maximum parsimony, and Bayesian approaches) used to produce a phylogenetic analysis. However, auxiliary techniques such as consensus tree algorithms must be scaled up as well. Summarizing the evolutionary relationships within a collection of trees (which could be produced from a Bayesian analysis or bootstrap tests) potentially becomes a bottleneck in completing a phylogenetic analysis if fast algorithms are not designed to handle the ever-increasing size of collections of trees.

Finally, our results indicate that HashCS can be used for more than post-processing trees. Given that it can produce a majority or strict consensus tree quickly, it can potentially be used in other ways to aid in inferring phylogenetic trees. For example, it is often difficult to determine whether a phylogenetic heuristic such as

MrBayes or TNT has converged. With fast consensus algorithms, one can take a collection of trees that have been sampled from tree space and construct their strict or majority tree. If the consensus tree has not changed significantly in some sufficient amount of time, then the search has potentially reached a local optima and it could be terminated. The resulting consensus resolution rates can vary significantly depending on the sampling of the trees in tree space. Hence, HashCS is a good consensus approach to use given that its fast performance is not impacted by the degree of bipartition sharing among the trees. Furthermore, as described by Amenta et al., fast consensus algorithms can be used as the foundation for an interactive system for visualizing collections of trees [71]. Thus, HashCS provide scientists with a fast approach for summarizing their trees in new and interesting ways.

CHAPTER VI

HASHRF: COMPUTING THE ROBINSON-FOULDS DISTANCE

Our second hash-based randomized algorithm, HashRF is described in this chapter. HashRF is implemented using the hash table described in Chapter IV. Figure 13 illustrates the flow of our algorithm. The main part is collecting necessary information from the input evolutionary trees and storing the information in the hash table. In this implementation, the collected information is the tree identities which share the bipartitions. After collecting the information, the hash table is reprocessed to produce similarity matrix and the similarity matrix is converted to RF (Robinson-Foulds) distance matrix. Using biological and artificial tree collections, we show how HashRF performs in practice as well as explore its scalability across a diverse set of trees.

A. Motivation

Consensus trees are commonly used for analyzing phylogenetic trees but summarizing large number of trees (which are believed to be equally plausible) into a single consensus tree loses valuable evolutionary information contained in the trees. Given a collection of trees, all-to-all relationship among the trees in a form of similarity/dissimilarity matrix can help researchers extract more information from the trees. Figure 4 shows the alternative way to represent all-to-all relationships among trees in a distance matrix. Among various metrics to measure the difference between trees, Robinson-Foulds (RF) distance metric is widely used to measure the topological distance between phylogenetic trees.

Here, our research question is: *How to design efficient algorithms to compute the all-to-all RF matrix for large collections of phylogenetic trees?* We believe that such a $(t \times t)$ RF distance matrix between every pair of trees provides a more information-rich

approach for summarizing t trees. Comparing the competing phylogenetic hypotheses from a phylogenetic search based on RF distance matrix represents tremendous data-mining opportunity for understanding the relationships depicted by the collection of trees.

B. Definition: Robinson-Foulds (RF) Distance

The RF distance between two trees is the number of bipartitions that differ between them. The unweighted RF distance deals with topological distance between trees without considering branch lengths. Let $\Sigma(T)$ be the set of bipartitions defined by all edges in tree T . The RF distance between two trees T_i and T_j is defined as:

$$d(T_i, T_j) = \frac{|\Sigma(T_i) - \Sigma(T_j)| + |\Sigma(T_j) - \Sigma(T_i)|}{2}. \quad (6.1)$$

PAUP* and Phylip compute the symmetric difference instead of the RF distance. The symmetric difference is the numerator of Equation 6.1, and it can easily be converted to the RF distance by dividing by 2. The minimum RF distance is 0. The maximum RF distance between binary trees is $n - 3$ (or number of internal branches), where n is the number of taxa.

We are interested in computing the *RF distance matrix*. Given a set of t input trees, the output is a $t \times t$ matrix of RF distances. Figure 4 shows the RF distance matrix between the four trees T_1, T_2, T_3 , and T_4 . Note that T_4 has no bipartitions in common with T_1 and T_2 . As a result, the maximum RF distance of 2 is shown for pairs (T_1, T_4) and (T_2, T_4) .

C. HashRF Algorithm Description

The Robinson-Foulds (RF) distance between two trees is the number of bipartitions that differ between them. Let $\Sigma(T)$ be the set of bipartitions defined by all edges in tree T . The RF distance between two trees T_i and T_j is defined by Equation 6.1.

PAUP* and Phylip compute the symmetric difference instead of the RF distance. The symmetric difference is the numerator of Equation 6.1, and it can easily be converted to the RF distance by dividing by 2. The minimum RF distance is 0. The maximum RF distance between binary trees is $n - 3$ (or number of internal branches), where n is the number of taxa.

We are interested in computing the *RF distance matrix*. Given a set of t input trees, the output is a $t \times t$ matrix of RF distances. Figure 4 shows the RF distance matrix between the four trees T_1, T_2, T_3 , and T_4 . Note that T_4 has no bipartitions in common with T_1 and T_2 . As a result, the maximum RF distance of 2 is shown for pairs (T_1, T_4) and (T_2, T_4) .

To compute the RF distance efficiently, we implemented our HashRF algorithm which also based on the hash technique. Our HashRF [97, 106, 107, 108] algorithm also consists of two major steps. However, unlike the HashCS algorithm, our HashRF approach does not require the additional use of a n -bitstring. Our approach relies solely on implicit representations of bipartitions.

1. Step 1: Populating the Hash Table

Figure 29 provides an overview of the populating step in HashRF algorithm and Algorithm 1 provides a bird-view of our HashRF implementation.

We also assume that input phylogenetic trees are stored in a file using Newick format [79]. Each implicit bipartition collected from the input trees is fed to hash

functions h_1 and h_2 which are described in Chapter IV. Unlike HashCS, every bipartition is added to the hash table in the HashRF algorithm. For HashCS, it is not necessary to know the origin of a bipartition. It only matters how often the bipartition appears in the set of input trees. For HashRF, a bipartition cannot be anonymous in the hash table. For each bipartition, its associated hash table record contains its bipartition ID (BID) along with the tree index (TID) where the bipartition originated.

2. Step 2: Calculating the RF Distance Matrix

Once all the bipartitions are organized in the hash table, then the RF distance matrix can be calculated. For each non-empty hash table location i , we have a list of $\{BID, TID\}$ objects. Consider the linked list of bipartitions in location 4 of the hash table in Figure 29. The linked list (or chain) at this location contains $\{27, T_1\}$, $\{27, T_2\}$, $\{27, T_3\}$, and $\{35, T_4\}$. Hence, there are two unique bipartitions represented at location i . Trees T_1, T_2 , and T_3 share the same bipartition. The bipartition represented by the pair $\{35, T_4\}$ is unique since it is present only in tree T_4 .

We use a $t \times t$ dissimilarity matrix, D , to track the number of bipartitions that are different between all tree pairs. That is, $D_{i,j} = |\Sigma(T_i - T_j)|$, as described by Equation 6.1. For each tree i , its row entries are initialized to b_i , the number of bipartitions present in tree i . Hence, $D_{i,j} = b_i$ for $0 \leq j < t$ and $i \neq j$. $D_{i,i} = 0$.

For each location l in the hash table, two different hash records, u_l and v_l , with the same BIDs represent identical bipartitions. Let $i = TID(u_l)$ and $j = TID(v_l)$. Then, the counts of $D_{i,j}$ and $D_{j,i}$ are decremented by one. That is, we have found a common bipartition between T_i and T_j and decrement the difference counter by one. Once we have computed D , we can compute the RF matrix, the average distance between all tree pairs, quite easily. Thus, $RF_{i,j} = \frac{D_{i,j} + D_{j,i}}{2}$, for every tree pair i and j .

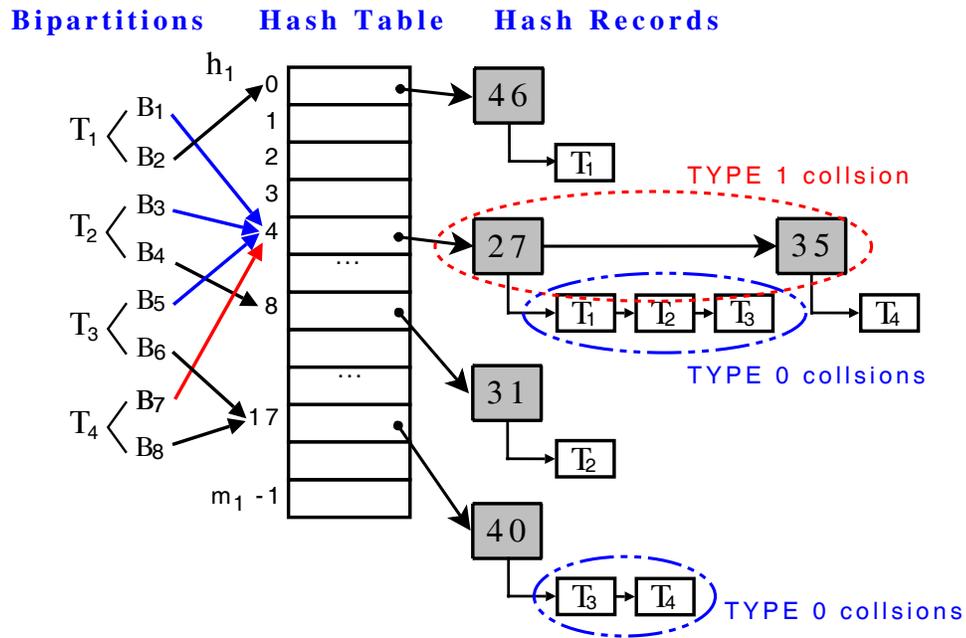


Fig. 29. Overview of the HashRF algorithm. Bipartitions are from Figure 4. The implicit representation of each bipartition is fed to the hash functions h_1 and h_2 . The shaded value in each hash record contains the bipartition ID (or h_2 value). Each bipartition ID has a linked list of tree indexes that share that particular bipartition.

Algorithm 3 HashRF algorithm for computing RF distance.

Require: A set of trees, T_1, T_2, \dots, T_t , where $t \geq 2$

- 1: **for** $i = 1$ to t **do**
- 2: Traverse tree T_i in post order and find B_j
- 3: **for all** bipartition $B_j \in T_i$ **do**
- 4: Compute $h_1(B_j)$ and $h_2(B_j)$, implicitly.
- 5: Check the BID list at the hash table $H[h_1(B_j)]$ for $h_2(B_j)$.
- 6: **if** $h_2(B_j)$ exists **then**
- 7: Insert the tree index i into the TID list of $h_2(B_j)$
- 8: **else**
- 9: Insert a new item, $h_2(B_j)$ at $H[h_1(B_j)]$
- 10: Insert i into the TID list of $h_2(B_j)$
- 11: **end if**
- 12: **end for**
- 13: **end for**
- 14: Retrieve each TID list, l_i from $H[]$
- 15: **for all** $|l_i| \geq 2$ **do**
- 16: **for all** TID pairs $j, k \in l_i$ **do**
- 17: Increment $SIM[j][k]$
- 18: **end for**
- 19: **end for**
- 20: **for all** $i < j \leq t$ **do**
- 21: $RF[i][j] = (n - 3) - SIM[i][j]$
- 22: **end for**

3. Handling Collisions

The collision types in our randomized algorithms is introduced in Chapter IV. The probability of HashRF restarting because of a double collision (i.e. *Type 2* collision) among any pair of the bipartitions is $O\left(\frac{1}{c}\right)$ [71]. The probability of HashRF restarting because of a double collision among any pair of the bipartitions is $O\left(\frac{1}{c}\right)$. Since c can be made arbitrarily large, the probability of the algorithm having to restart as a result of a double collision can be made infinitely small. Thus, as the HashCS implementation, we do not explicitly check for *Type 2* collisions. Our experiments with varying c from 1 to 10,000, and running HashRF at least 100 times for each c value, resulted in our algorithm producing the correct RF distance matrix every time for an overall error rate of 0%.

D. Extensions of HashRF

1. WHashRF: Computing Weighted RF Distance

Our HashRF algorithm ignores the branch lengths of the input trees. In fact, HashRF regards all the branch lengths as '1' and simply computes the number of bipartitions that differs between pairs of trees. The weighted RF distance uses the branch lengths in phylogenetic trees as weights when computing topological distance among trees. Thus, weighted RF distance is very useful when the topological distance among the input trees are small but each tree has different branch lengths. Remember the main purpose of HashRF algorithm is to provide researchers with data-mining opportunities. If the resulting RF matrix has more variances of values in it, the relationship among the trees can be more distinguishable.

The biggest difference between WHashRF and HashRF is that WHashRF must also keep track of the branch length [109] associated with each bipartition B from tree T into the hash table. Suppose a bipartition B is common between two trees T_1 and T_2 . The weight (or branch length) of B in tree T_1 is $w_1(B)$ and its weight in tree T_2 is $w_2(B)$. Thus, the weighted RF difference for bipartition B is $|w_1(B) - w_2(B)|$. To compute the weighted RF distance between two trees T_1 and T_2 , we would apply the above process to all bipartitions in the two trees and sum the weighted difference between the bipartitions. Moreover, we also need to handle appropriately those bipartitions that appear in only one of the two trees.

Formally, suppose that every bipartition B in trees T_1 and T_2 has a positive branch length. Let $B \in \Sigma(T_1) \cup \Sigma(T_2)$. $w_1(B)$ and $w_2(B)$ denote the length of the branch corresponding to the bipartition B from $\Sigma(T_1)$ and $\Sigma(T_2)$, respectively. For all $B \in \Sigma(T_1)$ and $B \notin \Sigma(T_2)$, let $w_2(B) = 0$. Similarly, let $w_1(B) = 0$, for all $B \in \Sigma(T_2)$ and $B \notin \Sigma(T_1)$. The weighted Robinson-Foulds distance d_w between trees T_1 and T_2

is then defined by

$$d_w(T_1, T_2) = \sum_{B \in \Sigma(T_1) \cup \Sigma(T_2)} |w_1(B) - w_2(B)|. \quad (6.2)$$

The weighted RF distance matrix computes the weighted RF distance between every pair of trees in the tree collection of size t . Of the algorithms studied here, only the weighted version of our HashRF algorithm, which we call WHashRF algorithm, can compute the $t \times t$ weighted RF matrix.

2. HashRF(p,q): Computing Arbitrarily-sized RF Matrix

HashRF algorithm has two limitations: (a) only computes $t \times t$ RF matrix, (b) can not compute large RF matrix due to the memory space limitation. To overcome those limitations, we develop the HashRF(p,q) algorithm as the basis for our alternative approach for understanding the relationships among a collection of t trees. The *novelty* of this algorithm is that it can compute arbitrarily-sized ($p \times q$) RF matrices (see Figure 30) and it can minimize the memory requirements of computing extremely, large RF matrices. Several RF implementations are only suitable for small distance matrices. Other RF algorithms, such as Day’s algorithm, PGM-Hashed, and HashRF cannot compute arbitrarily-sized matrices. They are limited to all-to-all ($t \times t$) matrices.

HashRF(p,q) works similarly to HashRF which was described in Chapter V Section C. The first major difference between the two algorithms is that HashRF(p,q) requires as input two sets of trees, S_p and S_q , where $|S_p| = p$ and $|S_q| = q$. HashRF requires only one set of trees, S , as input. HashRF(p,q) requires the sets S_p and S_q of trees since the trees in S_p will only be compared to trees in S_q and vice versa. Trees

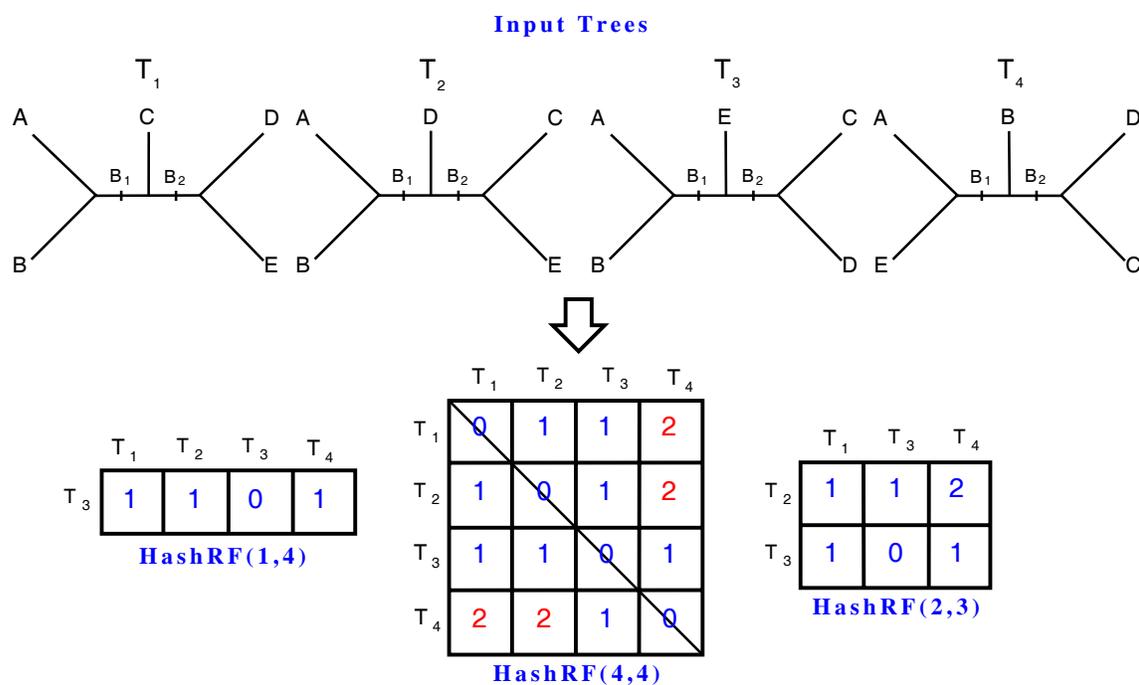


Fig. 30. Overview of computing the RF distance matrix using the HashRF(p,q) algorithm. The tree collection consists of four phylogenies: T_1, T_2, T_3 , and T_4 . Bipartitions (or internal edges) in a tree are labeled B_i , where i ranges from 1 to 2. Three different RF matrices are shown that can be produced by HashRF(p,q). HashRF(1,4) produces a *one-to-all* matrix, HashRF(4,4) computes an *all-to-all* matrix, and HashRF(2,3) computes a 2×3 matrix.

within a set will not be compared to each other. The HashRF algorithm (and all other RF approaches besides HashRF(p,q) that we study) assume that trees within a set must be compared to each other. Hence, only square matrices can be computed by such an approach whereas HashRF(p,q) is capable of computing rectangular matrices. In the HashRF(p,q) approach, we assume that $p \leq q$. As a result, we place all of bipartitions of the trees in S_q into the hash table. Once this is done, then we process each tree T_i in the set S_p . For each bipartition B of tree T_i , we apply our h_1 and h_2 hashing functions as described in Chapter IV. Once we determine where bipartition B would be located (using the h_1 function) in the hash table, we compare it's bipartition ID (using the h_2 function) to those nodes that are at that location. Suppose this location or index is l in the hash table. At location l , for each tree T_j at location l with the same BID as tree T_i , we increase the increment the counter in the matrix at locations (T_i, T_j) and (T_j, T_i) by one—assuming the full matrix is of interest and not the lower or upper triangle. We repeat the above steps for each remaining tree in the set S_p . Afterwards, since we are interested in the RF distance (and not similarity), we subtract $n - 3$ from the values since that is the maximum RF distance for a binary tree consisting of n taxa.

3. HashRF(p,q) vs. HashRF

HashRF(p,q) works similarly to HashRF. The first major difference between the two algorithms is that HashRF(p,q) requires as input two sets of trees, S_p and S_q , where $|S_p| = p$ and $|S_q| = q$. HashRF requires only one set of trees, S , as input and $|S| = t$. HashRF(p,q) requires the sets S_p and S_q of trees since the trees in S_p will only be compared to trees in S_q . Trees within a set will not be compared to each other.

In the HashRF(p,q) approach, we assume that $p \leq q$. As a result, we place all of bipartitions of the trees in S_q into the hash table. Once this is done, then we

process each tree T_i in the set S_p . For each bipartition B of tree T_i , we apply our h_1 and h_2 hashing functions as described in Chapter IV. Once we determine where bipartition B would be located (using the h_1 function) in the hash table, we compare it's bipartition ID (using the h_2 function) to those nodes that are at that location. Suppose this location or index is l in the hash table. At location l , for each tree T_j at location l with the same BID as tree T_i , we increment the counter in the matrix at locations (T_i, T_j) and (T_j, T_i) by one—assuming the full matrix is of interest and not the lower or upper triangle. We repeat the above steps for each remaining tree in the set S_p . Afterwards, since we are interested in the RF distance (and not similarity), we subtract $n - 3$ from the values since that is the maximum RF distance for a binary tree consisting of n taxa.

In terms of running time, the HashRF approach requires $O(nt^2)$ time. For HashRF(p,q) since we assume that $p \leq q \leq t$, then the running time is $O(nq^2)$. Hence, if a smaller sub-matrix is of interest, it will be significantly faster to compute than an all-to-all RF distance matrix.

4. Generate RF Distances of Arbitrary Cells in the Matrix

Our HashRF is a randomized algorithm based on hash table and computes $t \times t$ RF distance matrix, where t is the number of trees. The motivation for our HashRF(p,q) is to support 1-to- t comparison resulting a vector of RF distance values instead of $t \times t$ matrix. Our hash table is also utilized to produce the RF distance value between one or more specified trees without generating the whole $t \times t$ matrix. In other words, it is possible to specify one or more arbitrary cell locations in the RF distance matrix for computing RF distance values. For example, in Figure 31 shows a 4×4 RF distance matrix. If we want to get the circled RF values, producing 4×4 distance matrix is unnecessary. Once the hash table is populated, only the tree indices related with the

cells (i.e. (3, 2) and (4, 3)) need to be searched in the hash table. When we have a large RF distance matrix and we only concern a smaller set of RF values, this should be much faster.

	T ₁	T ₂	T ₃	T ₄
T ₁	0	1	1	2
T ₂	1	0	1	2
T ₃	1	1	0	1
T ₄	2	2	1	0

Fig. 31. An example of 4×4 RF distance matrix. The circled values represent the cell locations of concern.

E. Theoretical Analysis

The performance of our HashRF approach is affected by the number of bipartitions that are shared across the set of trees. Hence, the number of *Type 0* collisions have a significant impact on performance. In the best case, the expected chain length for a BID is $O(1)$. This occurs when all bipartitions in the t trees are unique. The expected chain length of tree index is $O(1)$ and the expected number of unique bipartitions is $O(nt)$. The total complexity is $O(nt)$, if $n > t$, or $O(t^2)$. The worst case occurs when there are t *Type 0* collision for each BID in the hash table. In this case, there will be n such BIDs. In other words, all of the trees are identical and the resulting expected worst case running time is $O(nt^2)$. For HashRF(p,q) since we assume that $p \leq q \leq t$, then the running time is $O(nq^2)$. Hence, if a smaller sub-matrix is of interest, it can be significantly faster to compute than an all-to-all RF distance matrix.

F. Experimental Analysis

1. Motivation

Our purpose of the experimental analysis comes from the question, *how does the performance of our HashRF algorithm compare to the existing approaches?* In this section, we analyze the running performance of our HashRF algorithm using variety of tree collections which are not only collected from biological trees but also generated to inspect the behavioral aspects of algorithms in detail. For the description on the experimental datasets, see Chapter V.

2. Implementations and Platform

HashRF, WHashRF, HashRF(p,q), Day, and PGM-Hashed were written in C++ and compiled with gcc 4.2.4 with the `-O3` compiler option. For PGM-Hashed, we obtained the source code for PGM-Hashed from the authors. PAUP* is commercially available software and we used version 4.0b10 in our experiments. Phylip is freely available and we used software versions 3.65. All experiments were run on the same platform described in Section 3.

3. Experimental Results

With the collection of biological trees, our HashRF algorithm is compared to four different approaches: Phylip, PAUP*, Day’s algorithm, and Pattengale, Gottlieb, and Moret’s Hashed (PGM-Hashed) algorithm [15]. We use $c = 1,000$ for HashRF. Since some of the methods compute the full RF matrix (instead of the upper or lower triangle), all algorithms compute the full matrix in our experiments. Both Phylip and PAUP* actually compute the symmetric difference, but it can be transformed into the RF distance by dividing this value by 2. In our experiments, we show the perfor-

mance of computing the symmetric difference in Phylip and PAUP*. Using the same collection of biological tree, the performance of WHashRF algorithm is compared with HashRF algorithm. Although the benefits of our HashRF(p,q) algorithm is that it can compute arbitrarily-sized matrices, to test it's performance against other RF matrix algorithms (PAUP*, PGM-Hashed, and HashRF) we compute all-to-all matrices in our experiments. We use $c = 1,000$ for HashRF(p,q). Each plot shows the average performance over five runs.

a. Performance on Biological Trees

HashRF Performance on Biological Trees. Figures 32(a) and 32(c) shows the running time of the RF distance matrix algorithms. Phylip and HashRF are the overall worst and best performers overall, respectively. On the largest dataset studied here, HashRF requires 1,353.4 seconds (or 22.6 minutes). The only other RF approach that was able to compute the RF matrix for this dataset was Day's algorithm, which required 13,123.4 seconds (or 3.64 hours). Both PAUP* and Phylip exceeded the time limit of 12 hours to analyze this dataset.

Finally, the PGM-Hashed algorithm will not run this problem size either. PGM-Hashed computes the probability of bipartitions colliding based on the number of taxa (n), number of trees (t), and bitstring length ($k = 64$). If the collision probability exceeds a threshold value, then the algorithm will not run the problem size as it is highly likely that the resulting RF matrix will be incorrect.

Figures 32(b) and 32(d) show the speedup of the HashRF approach over its competitors. Speedup is calculated as $\frac{x}{y}$, where x is the execution time required by PAUP*, PGM-Hashed, and Phylip and y is the running time of HashRF. To compute RF distance of 567 taxa, 4,096 biological trees, HashRF is 200 times faster than PAUP* and 2.1 times faster than PGM-Hashed. With 150 taxa trees, HashRF is

48 and 2.2 times faster than PAUP* and PGM-Hashed, respectively to compute RF distance of 16,384 trees. One trend of interest is that the speedup of HashRF over PGM-Hashed, the second-best competitor, appears to be decreasing with increasing number of trees. Figure 33(a) presents a more in-depth view of the performance of these two algorithms. Clearly, the speedup of HashRF is decreasing with increasing number of trees. Once the bipartitions have all been processed into the hash table, the RF matrix can be calculated (step 2 of the algorithm). Figure 33(b) shows that as the number of trees increases, up to 90% of the time can be consumed by step 2 of the HashRF approach. For the datasets studied here, the majority consensus trees have over an 85% resolution rate. Since many of the bipartitions are shared across the trees, this results in numerous Type 0 collisions in the hash table.

WHashRF Performance on Biological Trees. Figure 34 shows the distribution of the unweighted and weighted RF distance values for our collection of 150 and 567 taxa trees. For the unweighted RF distance, branch lengths are assumed to be one. However, the actual branch lengths returned by the Bayesian analysis have values much smaller than one. Although the weighted RF matrix may prove to be more useful for some datasets, computing the weighted RF matrix does require more time than its unweighted counterpart. Figure 35 shows that the HashRF algorithm is up to 5 times faster than WHashRF algorithm.

HashRF(p,q) Performance on Biological Trees. First, we consider the running time of our HashRF(p,q) algorithm against its competitors for computing an all-to-all (or $t \times t$) RF matrix. We also show the performance of HashRF in comparison with HashRF(p,q). Figure 36 shows the running time and speedup of HashRF(p,q) over its competitors on our 567 taxa tree collection. The results for our 150 taxa tree collec-

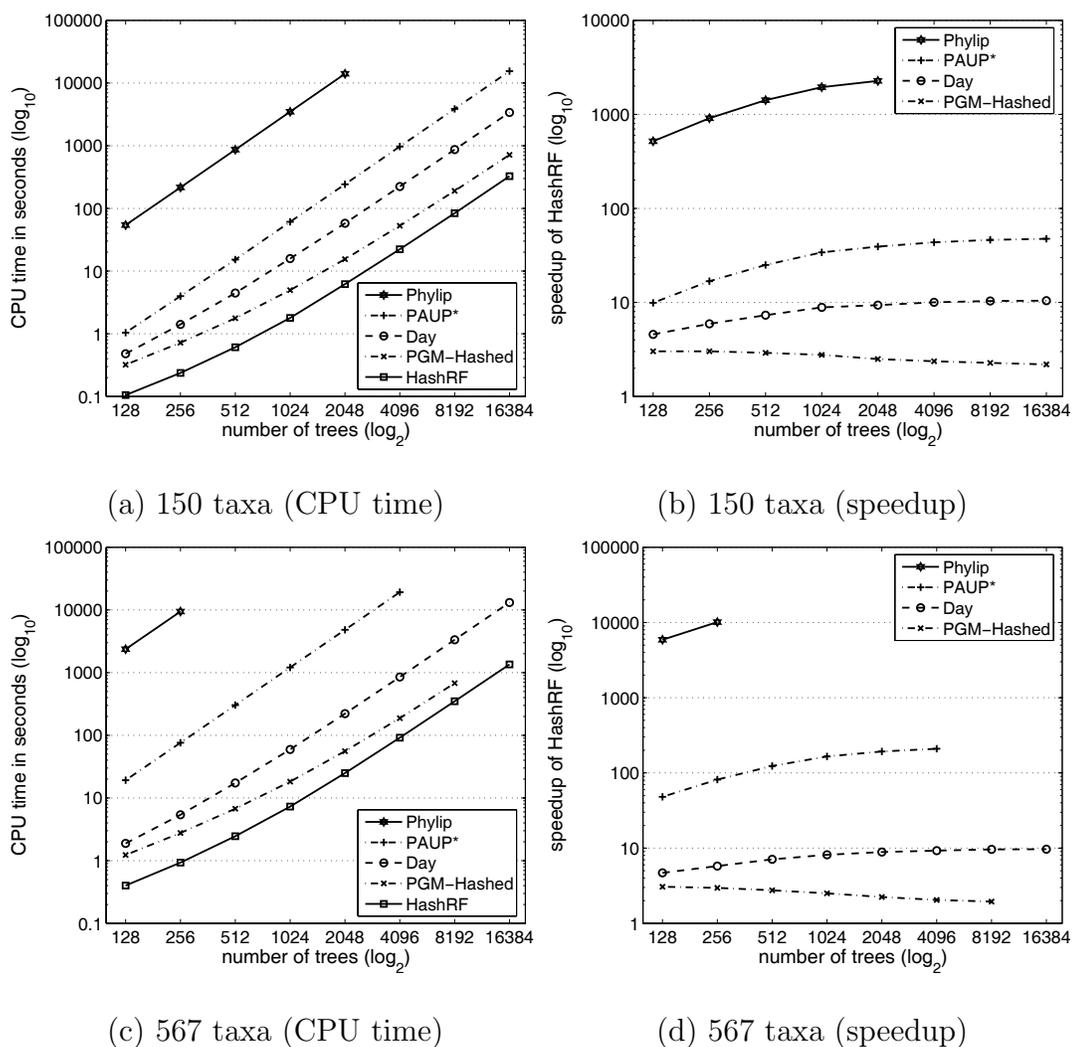
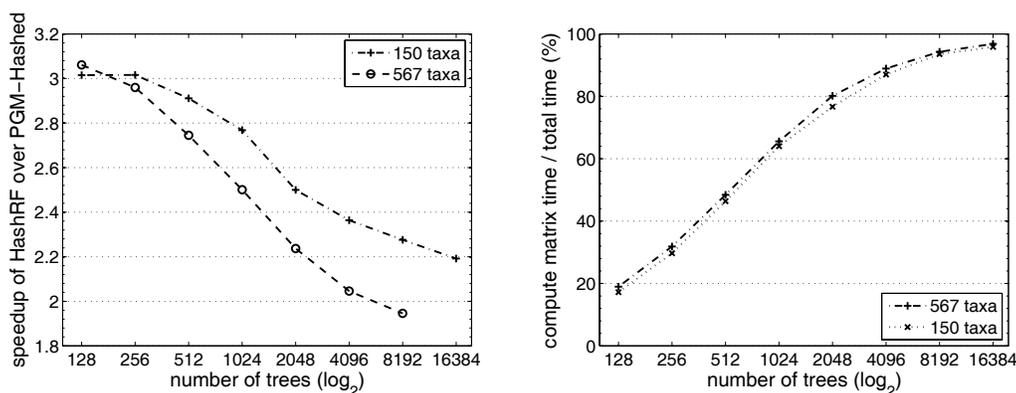


Fig. 32. Running time and speedup of the RF distance matrix algorithms. Data points for Phylip and PAUP* are missing if computation time exceeded 12 hours. Data points for PGM-Hashed are missing if it couldn't run that problem size. The scale of the y-axis is different for all the plots.



(a) Speedup of HashRF over PGM-Hashed (b) percentage of time for step 2 in HashRF

Fig. 33. A closer view at the performance of HashRF. (a) Speedup of HashRF over PGM-Hashed. (b) Percentage of time HashRF spends calculating the RF matrix once all the bipartitions are in the hash table (i.e., step 2 of the algorithm).

tions are similar. To compute the speedup values, the running times of PAUP*, PGM-Hashed, and HashRF are divided by the running time of HashRF(p,q), the algorithm of most interest to us in our experiments. Both HashRF and HashRF(p,q) clearly outperform the other algorithms. PAUP*, while popular, is the slowest algorithm requiring 5.35 hours to compute a $4,096 \times 4,096$ RF distance matrix. HashRF(p,q) only requires 93.9 seconds for the same data set, which results in a speedup of over 200 in comparison to PAUP*. As the number of taxa is increased, the speedup of HashRF(p,q) is increases as well, where the closest competitor PGM-Hashed is up to two times slower than HashRF(p,q).

Figure 36 also depicts several other interesting points. Although HashRF(p,q) was designed to compute sub-matrices, it is the best choice for computing the all-to-all matrix when $t > 2,048$ trees. HashRF is already a fast approach, but HashRF(p,q) can improve performance by as much as 40% when $t = 16,384$. For computing smaller $t \times t$ matrices, HashRF is the preferred approach. Since PAUP* takes over 12 hours

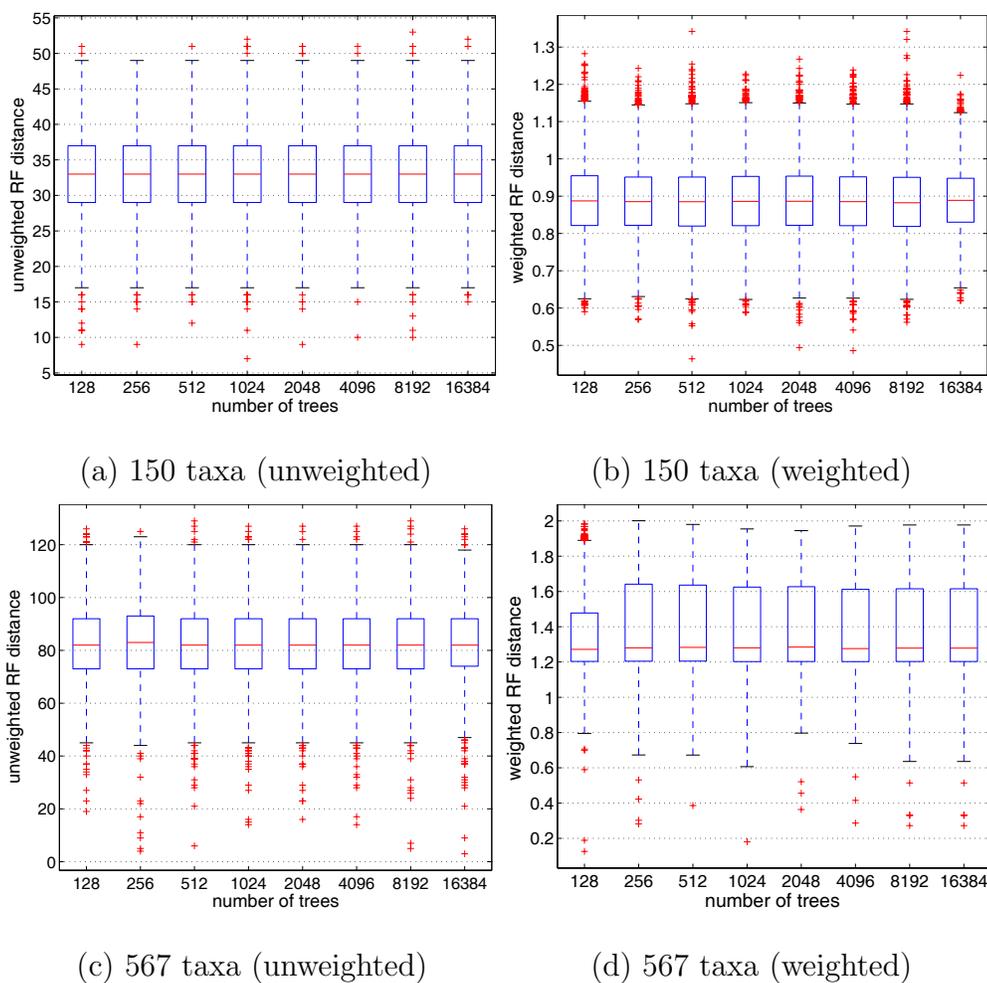


Fig. 34. Distribution of unweighted and weighted RF distances for the 150 taxa and 567 taxa datasets. The scale of the y-axis is different for all the plots.

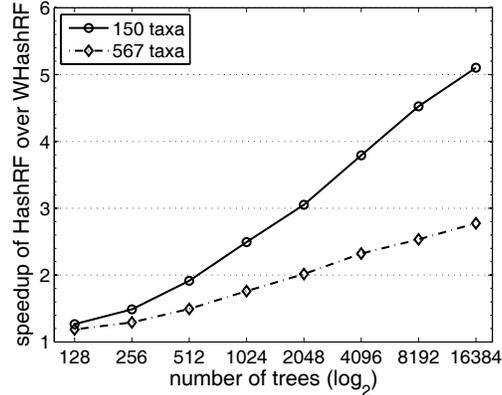


Fig. 35. Speedup of unweighted HashRF over WHashRF.

to compute $t \times t$ matrices, when $t \geq 8,192$, we choose to terminate the run. Finally, PGM-Hashed is unable to compute the $16,384 \times 16,384$ RF matrix.

b. Performance on Artificial Trees

Figure 37 shows the actual CPU time performance and bipartition comparison counts of the HashRF and PGM-Hashed algorithms on four of our artificial tree collections as a function of the consensus tree resolution rate. CPU time includes the time to traverse the input trees, insert each tree's bipartitions into the hash table (HashRF) or bipartition table (PGM-Hashed), and compute the resulting RF distance matrix. Counting the number of bipartition comparisons to compute the RF distance matrix comes into play once all bipartitions have been collected and organized into the appropriate data structure used by the RF matrix algorithm.

HashRF and PGM-Hashed show contrasting results in how they perform under different levels of bipartition sharing. Interestingly, the plots show that counting the number of bipartition comparisons is an effective measure for obtaining insights about algorithmic behavior since the trends shown by CPU time and bipartition comparison counts match very well. HashRF is the best overall performer both in

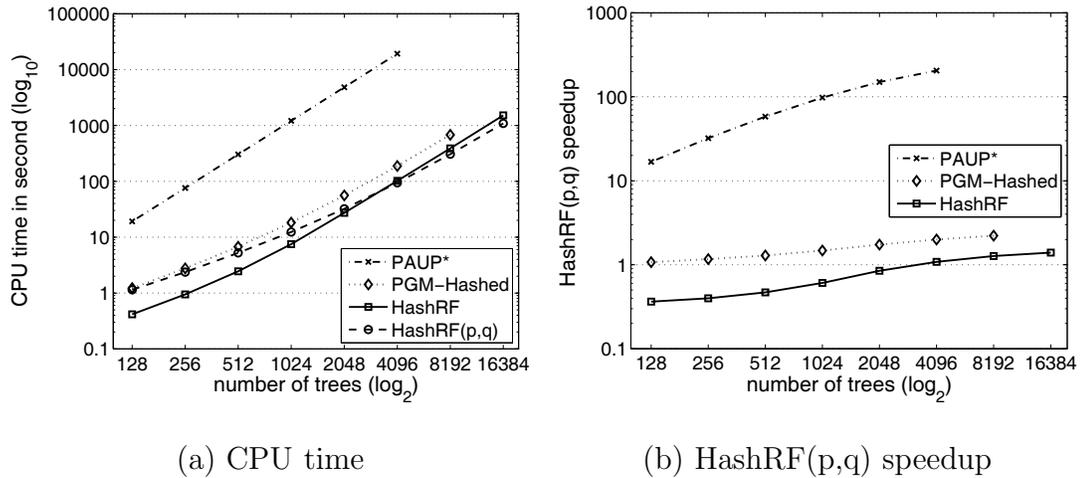
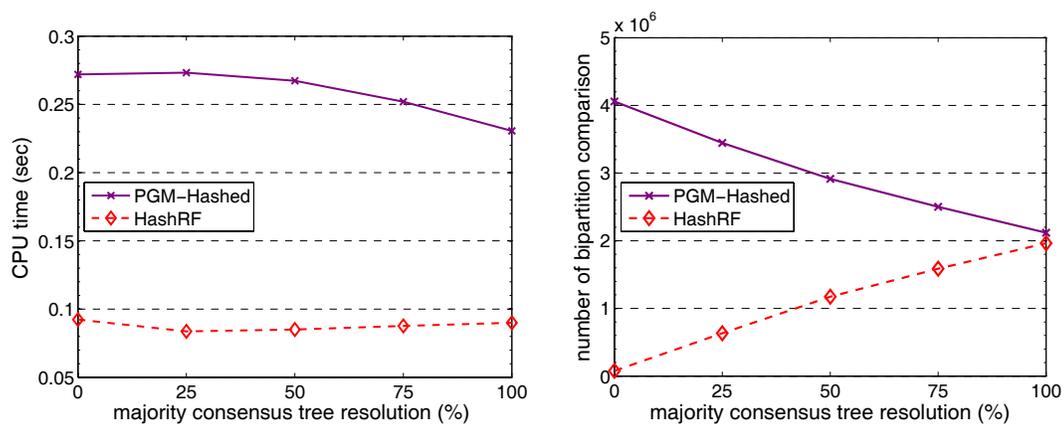


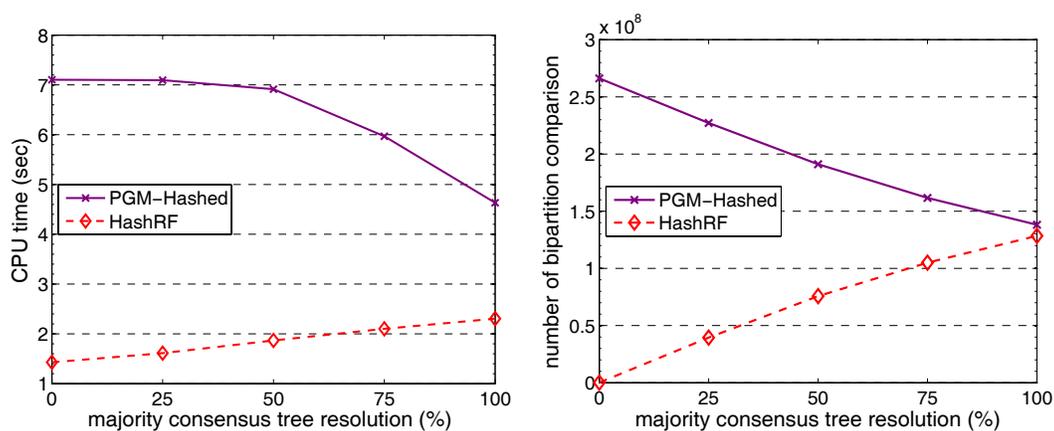
Fig. 36. The performance of the various RF matrix algorithms to compute a $t \times t$ matrix, where t is the number of trees, our 567 taxa dataset. For HashRF(p,q), p and q are both equal to t .

terms of actual CPU time and number of bipartition comparisons performed. In Figure 37, the worst case for HashRF is when many bipartitions are shared, which is depicted with increasing consensus tree resolution rates. HashRF's performance gets worse with increased bipartition sharing as a result of processing longer linked lists of trees in the hash table to compute the RF distance matrix (Longer linked lists results in more bipartition comparisons). PGM-Hashed on the other hand, gets faster as the amount of bipartition sharing increases. In PGM-Hashed, for each pair of trees T_i and T_j , two pointers p and q are used to compare the bipartitions, which are in sorted order based on their integer values. Two trees with identical bipartitions result in $n - 3$ comparisons to compute the RF distance between them. Two trees that do not share any bipartitions require $2(n - 3) - 1$ (or $2n - 7$) bipartition comparisons. Thus, PGM-Hashed runs faster as the similarity among the trees increases.

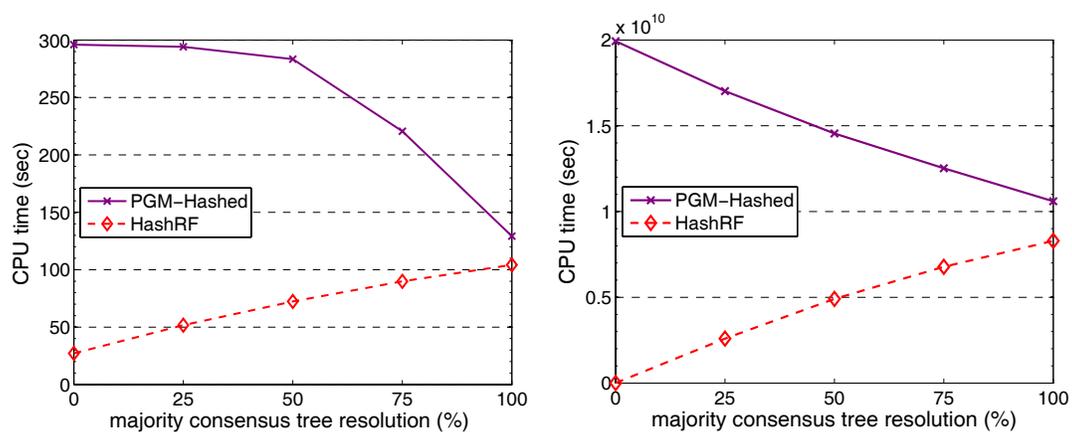
Finally, Figure 38 shows the memory usage of the RF matrix approaches. We used version 3.3.0 of the `valgrind` software package (<http://www.valgrind.org>)



(a) 128 taxa, 128 trees



(b) 512 taxa, 512 trees



(c) 2048 taxa, 2048 trees

Fig. 37. RF matrix algorithms performance on four of our artificial tree collections. The scale of the y-axis is different for all the plots.

to obtain our results. HashRF uses about three times less memory than PGM-Hashed. Interestingly, HashRF memory usage decreases as the number of shared bipartitions increases. When there are many unique bipartitions, a bipartition index (BID) and tree index (TID) have to be stored for each bipartition. However, for shared bipartitions, a single BID is stored for a linked list of TIDs. Thus, for HashRF this translates into less memory consumption. For PGM-Hashed the memory usage is essentially constant.

G. Summary

Phylogenetic reconstruction usually produce many candidate trees as estimations to the true evolutionary tree. Consensus methods are widely used to summarize the trees but much information is lost while producing single consensus tree. We advocate that the RF distance matrix which provides all-to-all topological distance information among trees is more information-rich way to analyze the trees. Further, the RF distance matrix provides plenty of data-mining opportunities to help researchers understand the evolutionary relationships contained in their collection of trees.

We designed and implemented a fast hash-based RF distance algorithm, HashRF, and explored the performance of RF distance algorithms using biological and artificial tree collections. Also we extended HashRF to WHashRF for computing the weighted RF distance matrix. To overcome limitations of our HashRF algorithm, we extend HashRF to a new algorithm called HashRF(p,q) which is not limited to computing the all-to-all (or $t \times t$) matrices between a collection of t trees. HashRF(p,q) can compute arbitrarily-sized $p \times q$ matrices, where $1 \leq p, q \leq t$. Moreover, the HashRF(p,q) approach can be used to compute very large RF matrices, which are not bounded by the amount of physical memory available on a user's system.

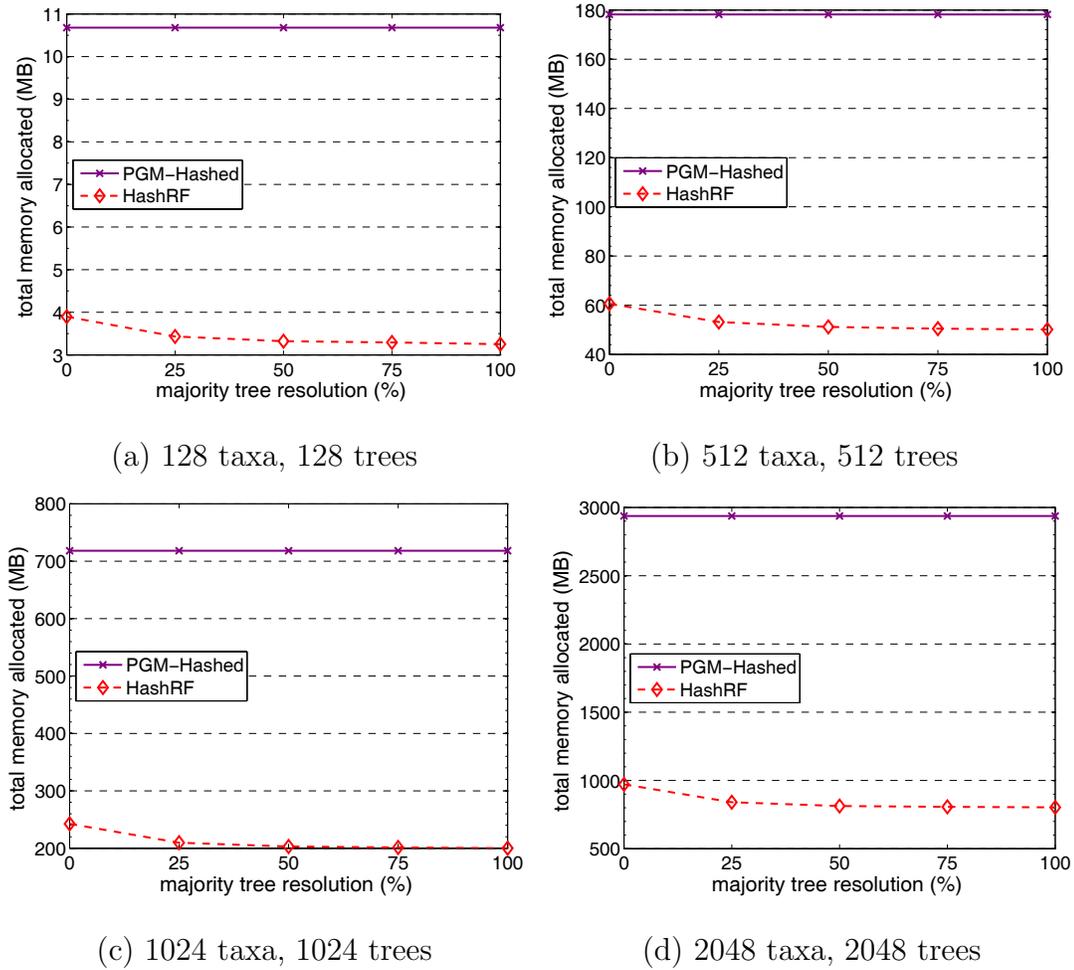


Fig. 38. Memory usage of the HashRF and PGM-Hashed algorithms. For the case of 2,048 taxa and 2,048 trees additional swap space beyond the 2 GB of physical memory on our platform was required. The scale of the y-axis is different for all the plots.

We provided extensive experimentation regarding testing the practical performance of our HashRF algorithm across a diverse collection of biological and artificial trees. Our HashRF algorithm is up to 3 times faster than its closest competitor, PGM-Hashed. HashRF is several orders faster than Day, PAUP*, and Phylip. Finally, we show that there is a performance penalty for computing the weighted versus unweighted RF distance matrix. However, our WHashRF algorithm computes the weighted RF matrix much faster than the other competing RF algorithms (except PGM-Hashed) calculate an unweighted RF distance matrix. Our experimental study shows that HashRF(p,q) is the best performing algorithm for large $t \times t$ matrices, where the number of trees is greater than 2,048. Popular phylogenetic software, such as PAUP*, is up to 200 times slower than HashRF(p,q). Furthermore, HashRF(p,q) is around 40% faster than our HashRF approach for large all-to-all matrices. Such performance results suggests that we could combine our two hash-based approaches into one so that one of them can be chosen based on size of the RF distance matrix of interest.

Next, we showed performance results on artificial trees. For computing RF matrix, the best case for HashRF comes when small amount of bipartitions are shared. PGM-Hashed, on the other hand, gets faster as the amount of bipartition sharing increases. Speedup of HashRF algorithm over PGM-Hashed is 12 times in the best case (with 0% shared bipartitions) and 1.3 times in the worst case (with 100% shared bipartitions).

As our grand challenge is the reconstruction of the *Tree of Life*, faster algorithm can help us analyze more trees in the same amount of time. Furthermore, the resulting RF distance matrix opens a way to apply diverse data-mining techniques to analyze large collections of phylogenetic trees.

CHAPTER VII

HASHCS AND HASHRF APPLICATIONS

We have introduced our novel algorithms to solve two real-world problems: (i) computing consensus trees for large collections of biological trees, and (ii) computing all-to-all RF (Robinson-Foulds) distance matrix for large collections of trees. We introduce our experience and results we earned from applying our implementations in two practical applications in this chapter: (i) comparing two heuristics, and (ii) clustering large collections of evolutionary tree.

A. Application #1: Comparing Phylogenetic Search Heuristics

1. Motivation

Of interest to systematists is the following question: “*What is the best phylogenetic heuristic to reconstruct evolutionary trees accurately?*” This best heuristic could then be used as the benchmark method to construct accurate evolutionary trees. In the realm of algorithms, computer scientists have long analyzed deterministic algorithms in terms of identifying fundamental operations, and counting those operations a function of input size. Asymptotic algorithms shares the benefits of platform and implementation independence, which is ideal for measuring and comparing algorithmic performance. In the interdisciplinary realm of phylogeny reconstruction, however, we face an extreme challenge. Computationally, we are trying to solve a problem whose solution we cannot verify: given a set of organisms or taxa, how did they evolve from a common ancestor? Phylogenetic search heuristics are used to search stochastically for the best trees in tree space and their results often vary across each run of the heuristic. Thus, it is difficult to compare performance among heuristics that produce

different solutions.

We develop new performance measures to compare the efficacy of phylogenetic search heuristics [110, 111]. Improved performance measures ultimately lead to better phylogenetic search heuristics, which will result in better approximations of the true evolutionary history of the organisms of interest. Our work focuses on the performance of two well-known maximum parsimony (MP) search heuristics, Parsimony Ratchet [77] and Recursive-Iterative DCM3 (Rec-I-DCM3) [28] on four molecular datasets of 60, 174, 500 and 567 taxa. The parsimony ratchet algorithm used is called Pauprat since we used a Perl script by Bininda-Emonds [27] to generate a PAUP* [75] batch file to run the parsimony ratchet heuristic.

Our work centers around the following two questions.

1. What value (if any) do slower heuristics provide?
2. How effective are parsimony scores in distinguishing between different tree topologies?

Traditional techniques for comparing phylogenetic search heuristics use convergence plots to show how the best score improves over time since best scores are thought to symbolize more accurate trees. Under this measure, the heuristic that obtains the best score the fastest is desired. Given that different tree topologies may have identical tree scores, preference of good-scoring trees found by fast heuristics may result in overlooking potentially more accurate evolutionary histories that can be found by slower approaches.

Our first observation is that there are benefits to considering different speed heuristic implementations of a MP phylogenetic analysis. In general, Pauprat is a slower heuristic than Rec-I-DCM3. Since we were curious of the merits of a heuristic, time constraints were removed from consideration in this study. However, both

Pauprat and Rec-I-DCM3 find different trees with the same best parsimony scores. These diverse best-scoring trees denote that the heuristics are visiting different areas of the exponentially-sized tree space. We note that although TNT [76] has a faster implementation of parsimony ratchet than Pauprat, TNT does not have the capability to return to the user the set of trees found during each iterative step of the parsimony ratchet algorithm. The Pauprat implementation of parsimony ratchet provides this capability. Moreover, the Rec-I-DCM3 implementation also provides users with the trees found during each step of the algorithm.

Secondly, although different trees are found with the same parsimony score, it's interesting to consider whether maximum parsimony is effectively distinguishing between the trees, which has significant implications for understanding evolution. By using a measure called *relative entropy*, we show for a given collection of trees that parsimony scores have less information content in distinguishing trees than topological distance measures such as the Robinson-Foulds (RF) distance [109]. In other words, for a collection of trees, parsimony scores identify fewer unique trees—which increases the potential of being stuck in a local optimum and producing less accurate phylogenies—than topological distance measures. Thus, more powerful search strategies could be designed that use a combination of score and topological distance to guide the search into fruitful areas of the exponentially-sized tree space.

2. Comparison Methods

a. Maximum Parsimony Heuristics

We study heuristics that use the maximum parsimony (MP) optimization criterion for inferring the evolutionary history between a collection of taxa. Each of the taxa in the input is represented by a molecular sequence such as DNA or RNA. These

sequences are put into a multiple alignment, so that they all have the same length. Maximum parsimony then seeks a tree, along with inferred ancestral sequences, so as to minimize the total number of evolutionary events by counting only point mutations.

Parsimony Ratchet. Parsimony ratchet is a particular kind of phylogenetic search performed with alternating cycles of reweighting and Tree Bisection Recombination (TBR). The approach works as follows: starting with an initial tree, a few of the characters (between 5 – 25%) are sampled, and reweighted. It suffices to say here that reweighting of characters involves duplicating the characters so that each shows up twice (or more) in the resulting dataset. Then, using these reweighted characters, TBR search is performed until a new starting tree is reached using this subset of data. This new starting tree is then used with the original data set to repeat the phylogenetic search. Parsimony ratchet tries to refine the search by generating a tree from a small subset of the data and using it as a new starting point. If the new tree is better than the old one, then the new one is used as the new starting tree. Otherwise, the old one is kept.

Rec-I-DCM3. Recursive-Iteration DCM3 (Rec-I-DCM3) [28] implements a disk-covering method (DCM) [112, 113, 114] to improve the score of the trees it finds. A DCM is a divide-and-conquer technique that consists of four stages: divide, solve, merge, and refine. At a high level, these stages follow directly from DCM being a divide-and-conquer technique.

Rec-I-DCM3, involves all of the above DCM stages, but in addition, is both recursive and iterative. The recursive part concerns the divide stage of the DCM, where overlapping subsets of the input tree's leaf nodes may be further divided into yet smaller subsets (or subproblems). This is an important enhancement to the

DCM approach since for very large datasets, the subproblems remain too large for an immediate solution. Thanks to the recursion, the subproblems are eventually small enough to be solved directly using some chosen base method. At this point, Rec-I-DCM3 uses strict consensus merger to do the work of recombining the overlapping subtrees to form a single tree solution. The iterative part of Rec-I-DCM3 refers to the repetition of the entire process just described. That is, the resulting tree solution becomes the input tree for a subsequent iteration of Rec-I-DCM3.

b. Comparing Collections of Trees

Relative Entropy. Entropy represents the amount of chaos in the system. We use entropy to quantitatively capture the distribution of parsimony scores and RF rates among the collection of trees of interest. In our plots, we show *relative entropy*, which is a normalization of entropy, to allow the comparison of entropy values across different population sizes. Relative entropy ranges from 0% to 100%. Higher entropy values indicated more diversity (heterogeneity) among the population of trees. Lower entropy values indicate less diversity (homogeneity) in the population.

Let λ represent the total number of objects (parsimony scores or RF rates) in the population of trees. For example, suppose we want to partition a population of 10 trees based on their parsimony scores. Then, $\lambda = 10$. However, if we are interested in partitioning the 10 trees based on the upper triangle of the corresponding 10×10 RF matrix, then $\lambda = \frac{10(9)}{2}$ or 45 since the RF matrix is symmetric. Next, we group the λ objects into P total partitions. Each partition i contains n_i individuals with identical values. For RF, each individual in partition i will have the same RF value. An individual in the RF matrix refers to a cell location (p, q) .

We can compute the entropy (E_T) of the collection of parsimony scores as:

$$E_T = - \sum_i^P p_i \log p_i,$$

where $p_i = \frac{n_i}{\lambda}$. The highest entropy value (E_{max}) is $\log \lambda$. Relative entropy (E_{rel}) is defined as the quotient between the entropy E_T and the maximum entropy E_{max} and multiplying by 100 to obtain a percentage. Thus,

$$E_{rel} = \frac{E_T}{E_{max}} \times 100.$$

Resolution Rate. For n taxa, a resolved, unrooted binary tree will have $n - 3$ bipartitions (or internal edges). Trees with less than $n - 3$ bipartitions are considered to have unresolved relationships among the n taxa. In general, binary (or 100% resolved) trees are preferred by life scientists. The *resolution rate* of a tree is the percentage of bipartitions that are resolved. One common use of this measure is related to evaluating consensus trees, which are used to summarize the information from a set of t trees. The strict consensus method returns a tree such that the bipartitions of the tree are only those bipartitions that occur in all of the t trees. The majority consensus tree incorporates those bipartitions that occur in at least 50% of the t trees of interest. Highly resolved consensus trees denote that a high degree of similarity was found among the collection of trees.

3. Experimental Methodology

Datasets. We used the following biological datasets as input to study the behavior of the maximum parsimony heuristics.

1. A 60 taxa dataset (2,000 sites) of ensign wasps composed of three genes (28S ribosomal RNA (rRNA), 16S rRNA, and cytochrome oxidase I (COI)) [115]. The best-known parsimony score is 8,698, which was established by both Pauprat and Rec-I-DCM3.
2. A 174 taxa dataset (1,867 sites) of insects and their close relatives for the nuclear small subunit ribosomal RNA (SSU rRNA) gene (18S). The sequences were manually aligned according to the secondary structure of the molecule [116]. The best-known parsimony score is 7,440, which was established by both Pauprat and Rec-I-DCM3.
3. A set of 500 aligned *rbcL* DNA sequences (759 sites) [78] of seed plants. The best-known parsimony is 16,218, which both Pauprat and Rec-I-DCM3 found.

Starting Trees. All methods used PAUP*'s random sequence addition module to generate the starting trees. First, the ordering of the sequences in the dataset is randomized. Afterwards, the first three taxa are used to create an unrooted binary tree, T . The fourth taxon is added to the internal edge of T that results in the best MP score. This process continues until all taxa have been added to the tree. The resulting tree is then used as the starting tree for a phylogenetic analysis.

Parameter Settings. We set the parameters of the Pauprat and Rec-I-DCM3 algorithms according to the recommended settings in the literature. We use PAUP* [77] to analyze our four datasets using the parsimony ratchet heuristic. The implementation of the parsimony ratchet was developed by Bininda-Emonds [27]). For our analysis, we randomly selected 25% of the sites and doubled their weight; initially, all sites are equally weighted. On each dataset, we ran 5 independent runs of the

parsimony ratchet, each time running the heuristic for 1,000 iterations. For Rec-I-DCM3, it is recommended that the maximum subproblem size is 50% of the number of sequences for datasets with 1,000 or less sequences and 25% of then number of sequences for larger datasets not containing over 10,000 sequences. We used the recommended settings established by Roshan et. al [28] for using TNT as a base method within the Rec-I-DCM3 algorithm.

Implementation and Platform. We used the HashRF algorithm to compute the RF distances between trees. Each heuristic was run five times on each of the biological datasets. All experiments were run on a Linux Beowulf cluster, which consists of four, 64-bit, dual-core processor nodes (16 total CPUs with gigabit switched interconnects). Each node contains four, 2 GHz AMD Opteron processors and they share 4GB of memory. We note that both Rec-I-DCM3 and parsimony ratchet are sequential algorithms. The parallel computing environment was used as a way to execute multiple, independent batch runs concurrently.

4. Results and Discussion

Frequency of the Top-scoring Trees. Table II shows the number of trees found by the Pauprat and Rec-I-DCM3 heuristics in terms of the number of steps they are from the best score, b , we found. Let x represent the parsimony score of a tree T . Then, tree T is $x - b$ steps away from the best score. In Table II, $step_0$, $step_1$, and $step_2$ represents trees that are 0, 1 and 2 steps away from the best score, b , respectively. Hence, $step_0$ trees are the trees with the best-known scores. It is clear that the top-scoring trees from Pauprat comprise a large proportion of the total collection of 5,000 trees for the smaller datasets (60 and 174 taxa). On the other hand, the top trees for Rec-I-DCM3 comprise the majority of its collection of trees for the larger dataset.

So, if one is simply interested in frequency counts, Pauprat finds best-scoring trees more often than Rec-I-DCM3 on the smaller datasets and Rec-I-DCM3 prevails on the 500 taxa dataset.

Topological Comparisons of the Top Trees. Figure 39 shows the topological differences between the top-scoring trees found by the different search heuristics. We use a heatmap representation, where each value (cell) in the two-dimensional $t \times t$ matrix is represented as a color. Darker (lighter) colors represent smaller (higher) RF rates. Our heatmaps are symmetric two-dimensional matrices representing the $t \times t$ RF rates matrix. For each heatmap, the left values are x coordinates and the values on the top are y coordinates. Consider the heatmap represent the collection of 60 taxa trees. Cell (1, 1) represents the set of step_0 trees from the Pauprat heuristic. The 1,508 step_0 trees are compared to each and their RF rates are 0%, which is denoted by a black coloring of the $1,508 \times 1,508$ block of cells. Hence, the best scoring trees found by the Pauprat heuristic are identical. A similar conclusion can be made concerning the 59 step_0 trees found by Rec-I-DCM3 and denoted by cell (2, 2) in the heatmap.

If we look at the step_0 trees from both Pauprat and Rec-I-DCM3, represented by cells (x, y) , where $x, y \leq 2$, the entire block of cells have a RF rate of 0%. Hence, for the 60 taxa dataset, the heuristics found identical best-scoring trees. For the step_2 trees, reflected in cells (3, 3) and (4, 4), there is more variation among the 60 taxa trees. The heatmap also shows comparisons of trees with different number of steps from the best. For example, cell (1, 4) compares step_0 Pauprat trees with step_2 Rec-I-DCM3 trees.

Overall, the heatmaps in Figure 39 show that the Pauprat and Rec-I-DCM3 algorithms find topologically different trees. The best trees are identical in the 60 taxa dataset. However, for the other datasets, the best (step_0) trees found by each

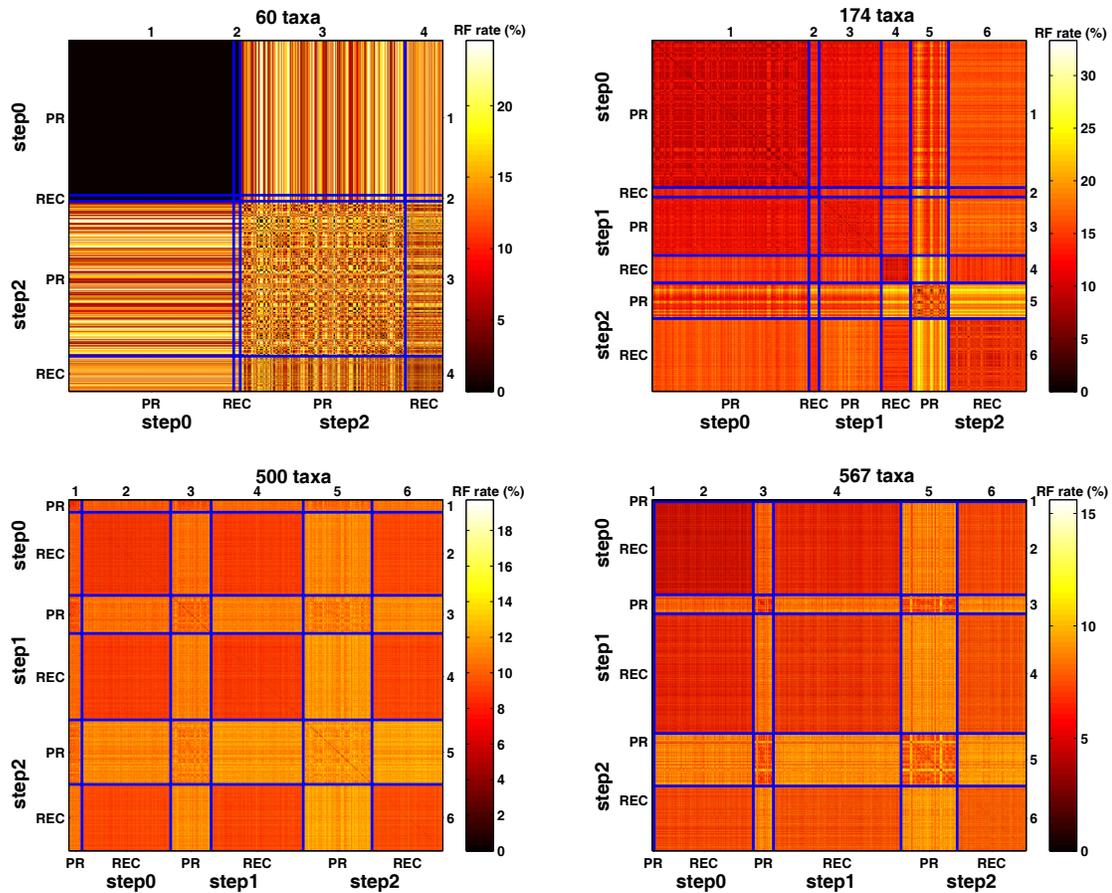


Fig. 39. Comparing the topologies of the top-scoring trees found by the Pauprat and Rec-I-DCM3 heuristics. The RF rate axis varies across the heatmaps.

Table II. The table shows the number of top-scoring trees found by each algorithm. step_i refers to trees that are i steps away from the best score. Hence, step_0 represents the number best-scoring trees, step_1 are trees one step away from the best, and step_2 are trees with parsimony scores two steps greater than the best score. Each algorithm found 5,000 total trees. For Pauprat (Rec-I-DCM3), the step_0 , step_1 , and step_2 trees make up 67.6% (10.7%) of the 5,000 total trees in the collection for the 60 taxa dataset. For the 567 taxa dataset, the top-scoring trees represent 22.5% and 77.5% of the 5,000 trees found by Pauprat and Rec-I-DCM3, respectively.

No. of taxa	best score	Pauprat				Rec-I-DCM3			
		step_0	step_1	step_2	% of total	step_0	step_1	step_2	% of total
60	8,698	1,508	0	1,509	60.3%	59	0	343	8.0%
174	7,440	2,626	1,042	635	86.1%	170	491	1,301	39.2%
500	16,218	184	562	955	34.0%	1,231	1,279	983	69.9%
567	44,165	27	263	735	22.5%	1,299	1,671	903	77.5%

algorithm are topologically distinct. 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. Finally, the heatmaps show that Pauprat finds more topologically dissimilar trees than Rec-I-DCM3. Thus, the Rec-I-DCM3 step_i trees tend to form clusters that are distinct from the step_i Pauprat trees.

Next, consider the heatmaps in Figure 40 which reflect strict consensus resolution rates for a collection of t trees. High resolution rates (e.g., above 85%) reflect high similarity among the trees of interest. This heatmap is read similarly to Figure 39. The difference is the interpretation of cell (i, j) . For example, on the 60 taxa dataset, cell $(1, 1)$ represents the strict consensus resolution of the 1,508 step_0 trees from Pauprat. Cell $(2, 3)$ is the strict resolution rate of the step_0 Rec-I-DCM3 trees with step_2 Pauprat trees. The consensus resolution rate is the highest for the step_0 trees.

This results corroborates the result in Figure 39 that shows that step_0 trees are more topologically similar to each other than higher scoring trees. Furthermore, the strict resolution rate is greater among Pauprat trees than its Rec-I-DCM3 counterparts.

For both Pauprat and Rec-I-DCM3, the majority resolution of comparing the top trees always resulted in a rate greater than 90% (see Figure 41). There was very little variation among the step_i trees when computing the majority tree. In fact, the results indicate that all of the top-scoring trees could be used to create the majority consensus tree with minimal impact on the consensus resolution rate.

Comparisons over Time. Next, we focus on the performance of Pauprat and Rec-I-DCM3 in terms of time using all of the trees returned by each phylogenetic heuristic. Here, time is measured by number of iterations (which is CPU time independent) and not on wall-clock time (e.g., number of hours required). Although number of iterations is an architecture-independent measure, it may not be completely adequate as each algorithm may do more work than the other per iteration. But, given that we are trying to compare heuristics based on solely their input/output behavior, that is the collection of trees returned after 1,000 iterations, we believe that using iterations as a basis of time is adequate for our purposes in this work.

Figures 42 and Figure 43 use relative entropy as a measure for uniformly quantifying the information content of parsimony scores and RF rates. Relative entropy is shown as a percentage of the maximum possible entropy. Higher relative entropy means that there is more diversity (heterogeneity) among the values of interest, and hence higher information content. Lower relative entropy values denote homogeneous values and lower information content. One implication of low entropy values is that the search has reached a local optimum. Higher entropy values signify that more diverse trees are found by a phylogenetic heuristic, which lessen its probability of

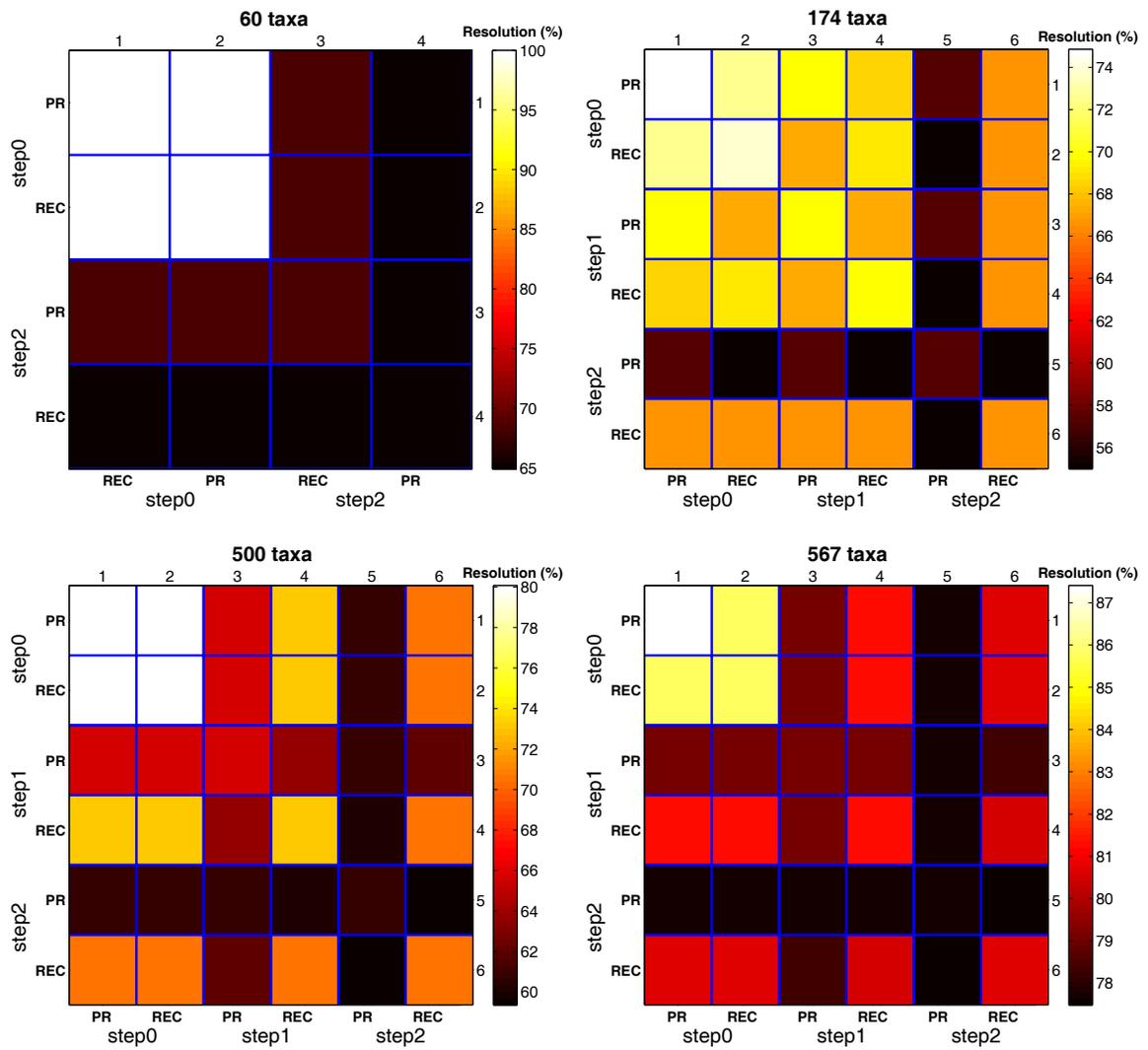


Fig. 40. Comparing the strict resolution rates of the top-scoring trees found by the Pauprat and Rec-I-DCM3 heuristics.

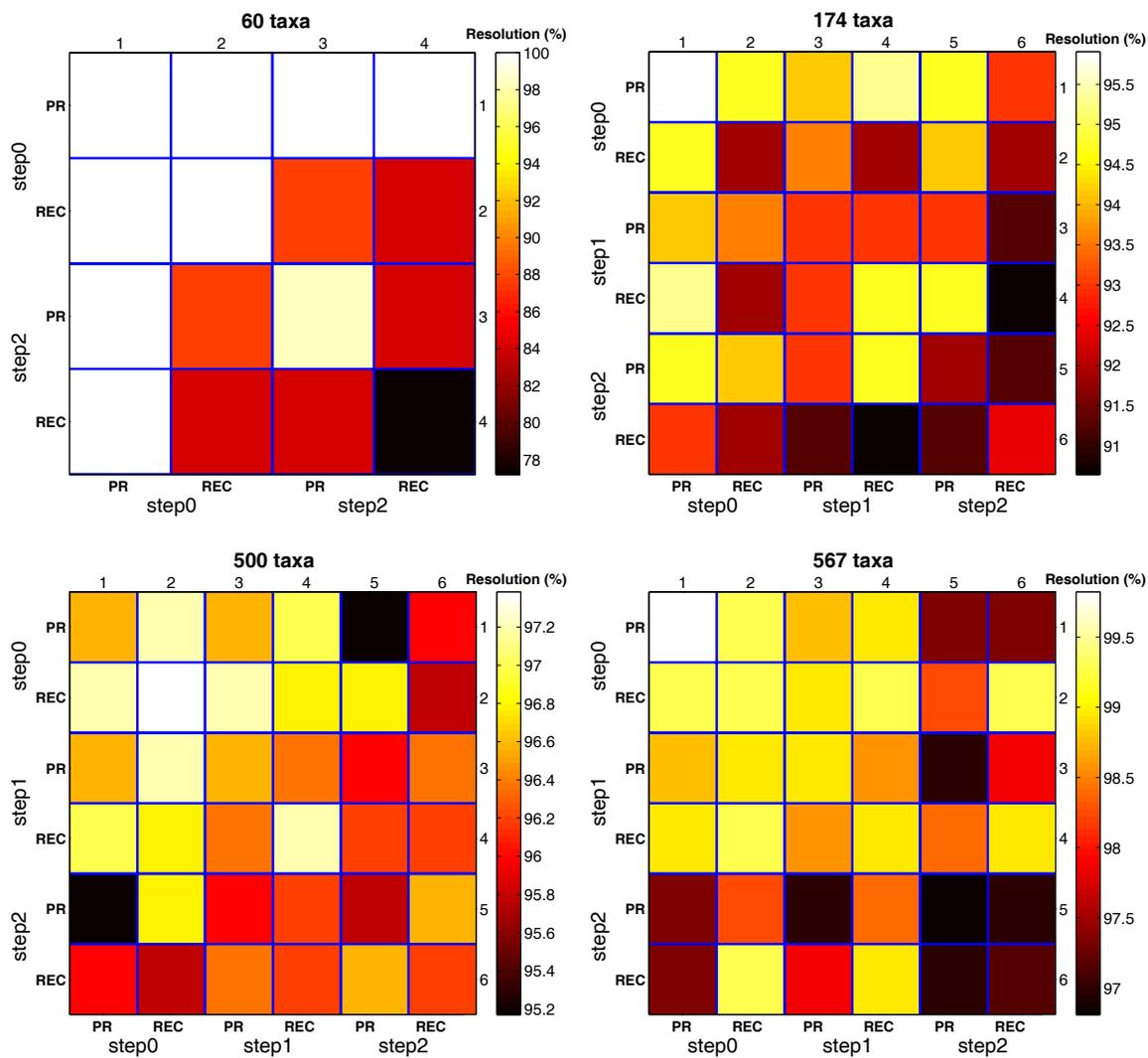


Fig. 41. Comparing the majority resolution rates of the top-scoring trees found by the Pauprat and Rec-I-DCM3 heuristics.

being trapped in local optima.

For the 174 and 500 taxa datasets, Pauprat has a higher relative entropy than Rec-I-DCM3 when comparing parsimony scores and RF distances. That is, Pauprat trees are more diverse than Rec-I-DCM3 trees. For the 60 taxa curves, Rec-I-DCM3 has a much higher relative entropy than Pauprat. Moreover, for Rec-I-DCM3, parsimony score entropy values are much higher than RF rate values for 60 taxa. Such a result implies that the parsimony scores of trees are more diverse than their topologies. In other words, trees with different scores when compared topologically are similar. For Pauprat, the relative entropy values vary quite a bit more than for Rec-I-DCM3, which has relative entropy values that are fairly constant across iterations. Essentially such behavior denotes that the Rec-I-DCM3 search has converged as there is not much change in the parsimony or RF rates among the trees found.

5. Summary

We used novel methods to assess the quality of two Maximum Parsimony-based phylogenetic search algorithms, Pauprat and Rec-I-DCM3. The goal of this work was to a.) ascertain the value (if any) of slower heuristics, and b.) understand if parsimony score is effective in distinguishing between different tree topologies. We designed a new entropy-based measure which we used in tandem with the Robinson Foulds topology distance metric to quantify levels of tree heterogeneity across separate iterations of these algorithms over several datasets. In addition, we used heatmaps to visualize levels of tree diversity within and between Pauprat and Rec-I-DCM3. Our results show that parsimony score masks diversity in large populations of equally parsimonious trees. This suggests that our topology-based methods may be better in quantifying fine-grain differences between different heuristics, especially in larger datasets. Thus, it would be valuable for heuristics to use both parsimony score

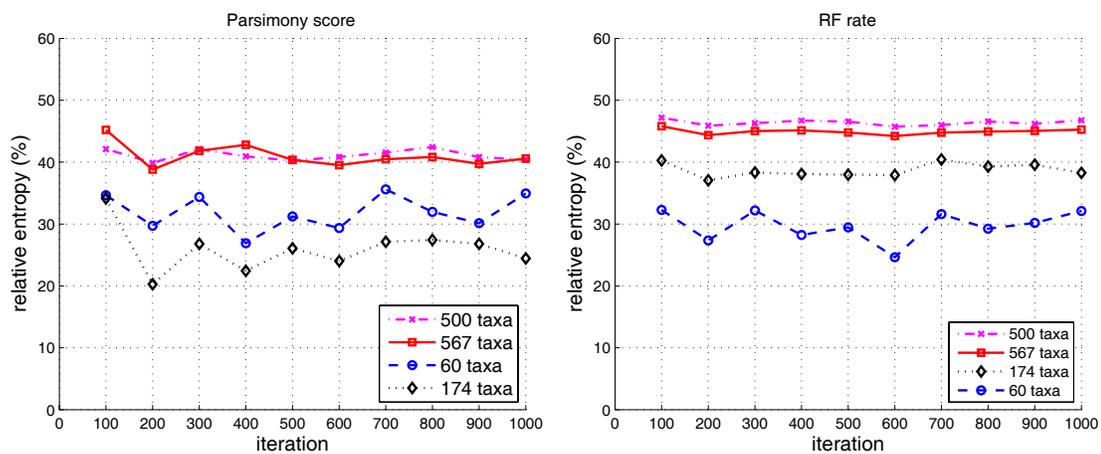


Fig. 42. Comparing the resolution rates of the top-scoring trees found by the Pauprat heuristics. Relative entropy values obtained from Pauprat trees every 100 iterations.

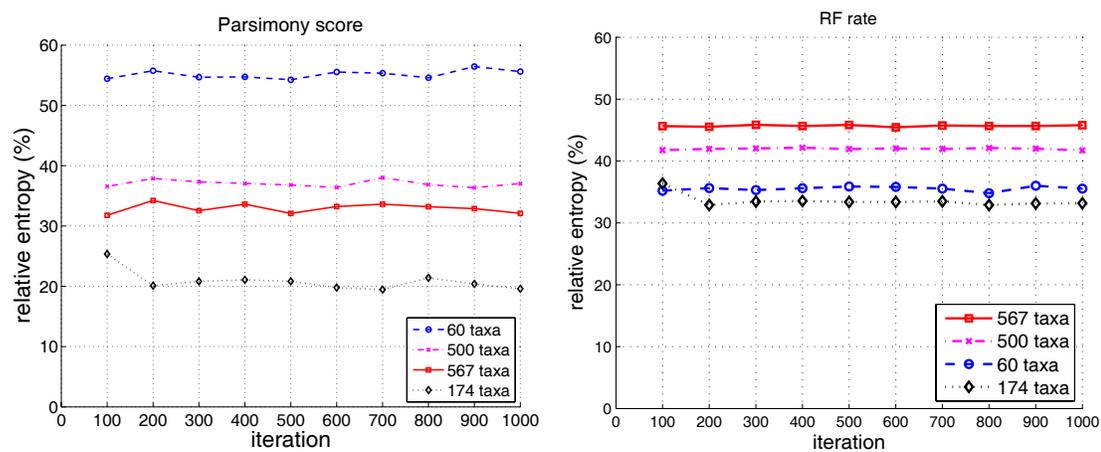


Fig. 43. Comparing the resolution rates of the top-scoring trees found by the Rec-I-DCM3 heuristic. Relative entropy values obtained from Rec-I-DCM3 trees every 100 iterations.

and tree topology as part of their optimization criteria. Furthermore Pauprat, while slower, finds different trees than Rec-I-DCM3. In addition, Pauprat's trees tended to be more diverse, especially as the dataset gets larger. Since Pauprat finds different trees from Rec-I-DCM3, the value in this heuristic lies in its ability to find different equally plausible candidate trees than Rec-I-DCM3. Depending on the evolutionary value of these different hypotheses, it may be worthy to improve *implementations* of Pauprat, rather than discount the algorithm itself.

For the future, we plan to create additional metrics for ascertaining levels of diversity in populations of trees. This work will also be extended to include other popular criteria, such as maximum likelihood. In addition, we plan to develop a heuristic that incorporates tree topology as a way to guide phylogenetic search. Such a search heuristic will be invaluable to the scientific community, and will help us find more accurate trees.

B. Application #2: Effective Techniques for Summarizing Large Collections of Evolutionary Trees

1. Motivation

Currently, biologists use popular phylogenetic software packages such as PAUP* [75], MrBayes [117], and TNT [22] to summarize their large tree collections into a single consensus tree. We develop two techniques to effectively summarize large collections of evolutionary trees utilizing our fast implementations described in the previous chapters. We study our techniques on two collections of Bayesian trees that were obtained from biologists on 150 taxa (desert algae and green plants) [103] and 567 taxa (angiosperms) [104] datasets. The 150 taxa dataset consists of 20,000 trees from two runs of the MrBayes phylogenetic heuristic. The 567 taxa dataset contains 33,306 trees from 12 Bayesian runs. To the best of our knowledge, we are the first to analyze biological tree collections consisting of tens of thousands of trees. Given such large tree collections, our primary research objective is to answer the following question: “*What is the best way to cluster large tree collections?*” The *novelty* of our work is based on developing two techniques to cluster tree collections effectively and quantify the results.

Our first technique clusters trees based on the MrBayes run that produced it. That is, trees from the same MrBayes run are placed into the same cluster. Then, the average dissimilarity between trees within a cluster and trees outside the cluster is computed. Our second technique uses a clustering algorithm to group the data. The clustering algorithm has no knowledge of the origin of the trees. Overall, both techniques show that summarizing the collection of trees as a single group is not the best representation of the data. Furthermore, both techniques agree that trees are best clustered based on the runs that generated them. Hence, in addition to finding

an effective method to summarize the trees, our results indicate that the Bayesian runs used to generate the trees explored different areas of tree space. Further runs may be desirable for convergence of the Bayesian analysis across runs.

Overall, our work presents systematists with tools and techniques for exploring their tree collections more thoroughly. More importantly, such techniques may help preserve crucial evolutionary relationships with low signal that may have been discarded in a single tree representation. Thus, the exciting benefit of our work is helping systematists understand their collections of trees in new and intriguing ways.

2. Previous Approaches

Our work has a number of distinguishing features. First, we are interested in studying collections containing tens of thousands of trees on large numbers of taxa (150 and 567 taxa). Although most of the previous work study collections with hundreds of trees, there have been studies that analyzed datasets with around 6,000 trees. One of Stockham et al. analysis looks at 5,630 trees on 129 taxa [80]. Hillis et al. study 6,000 trees on 40 taxa [81]. Secondly, we use a combination of clustering and visualization to analyze collections of trees. To cluster large groups of trees, we consider two techniques: grouping trees by run and grouping them using a clustering algorithm. For visualization, we use heatmaps instead of MDS (Multi-Dimensional Scaling) to quantify the similarities and dissimilarities among the clusters of trees. MDS visualizes the points (trees) of the clusters into two-dimensional space. It does not automatically determine whether points should be in the same cluster. Hence, under MDS, it is up to the human observer to group the points into clusters. Finally, the focus of our approach is on identifying similar groups of trees so that they can be summarized more effectively. Multipolar and meta-tree techniques are interested in effectively showing differences such as outlier trees and distinct bipartitions. Thus,

these techniques would be good companions to the methods described in this work.

3. Clustering Methodology

The experimental biological trees were obtained from two recent Bayesian analysis, which we describe in Chapter VI Section 2. All trees in our collections are unique. Table III provides statistics concerning our tree collections.

a. Clustering Large Tree Collections

We use CLUTO [118], a freely-available high-performance software for clustering large high-dimensional data, to cluster our two tree collections. CLUTO was chosen for its ability to cluster large data sets efficiently. We were unable to use Matlab or the R statistical package to cluster our tree collections. For each of our data sets we generated an RF matrix representing the all to all distances between the trees. This matrix was fed into CLUTO which treats the matrix as an array of distance vectors. So the 150 taxa data set with its 20,000 trees was treated as an array of vectors of 20,000 dimensional space. The default settings in CLUTO were used which tells the clustering algorithm to use the cosine between two vectors as the measure of similarity and to minimize the difference in internal similarity for each cluster as the primary metric for quality. We ran each data set of 150 taxa and 567 taxa trees to generate k clusterings, where $2 \leq k \leq 16$.

On our 150 taxa dataset, CLUTO took on average 6.5 minutes and approximately 8GB of memory to cluster the 20,000 trees for each k value. Our 567 taxa tree collection required on average 8.2 hours and over 24GB of memory for each k value. On our experimental platform (see Chapter VII Section 3), CLUTO would not run natively on Mac OS X. Thus, we ran it on our under the Ubuntu operating system (64-bit ver. 8.10) using the Parallels Desktop 4 virtual machine environment. The

Table III. Statistics for the two tree collections of interest. For the 150 and 567 taxa datasets, the total number of trees analyzed is 20,000 and 33,306, respectively. There are no identical trees in these collections.

dataset	total trees	majority tree <i>(resolution rate)</i>	strict tree <i>(resolution rate)</i>
150 taxa	20,000	85.7%	34.0%
567 taxa	33,306	92.6%	51.8%

Parallels environment can only take advantage of 8GB of system memory (even though our machine has 16GB of memory available). So, the 24GB of memory required by CLUTO consisted of 8 GB of system memory and 16GB of virtual memory on the 567 taxa dataset.

b. Computing Tree Distances

One type of input to the CLUTO clustering software is a $t \times t$ matrix representing the dissimilarity between the t objects of interest. In phylogenetics, the Robinson-Foulds distance [119] is a popular distance used to represent the dissimilarity between two trees. To compute a $t \times t$ RF matrix, we use our HashRF algorithm introduced in Chapter V. Other available packages (such as PAUP* [75]) are unable to handle creating such a large tree distance matrices. HashRF required 4.9 and 45.5 minutes to compute the $20,000 \times 20,000$ and $33,306 \times 33,306$ RF matrices, respectively, on our Mac Pro platform described in Chapter VII Section 3. The $20,000 \times 20,000$ matrix requires 1.2GB of disk space and the $33,306 \times 33,306$ RF matrix occupies 3.4GB of storage space. Although the RF matrix is symmetric, we store the full RF matrix since CLUTO requires it.

We use HashRF(p,q) [106] to compute the average RF rate between trees across (i) MrBayes runs and (ii) CLUTO clusters. For example, suppose that we want to

compute the average RF distance between trees in clusters C_i and C_j from CLUTO, where $|C_i|$ and $|C_j|$ are p and q , respectively. HashRF(p,q) would compute a $p \times q$ RF matrix based on Clusters C_i and C_j . Since the matrix is symmetric, the entries in the upper triangle are totaled to compute the average RF distance. However, in our plots, we show the average RF rate, which is the average RF distance divided by $n - 3$, the number of internal edges in a n taxa binary tree. Thus, the RF rate ranges from 0% to 100%, where higher RF rates represent greater dissimilarity among the trees.

c. Computational Platform and Implementations

All experiments were run on a two processor, eight-core (4 cores per processor) Apple Mac Pro platform with a total of 16GB of memory. Each 64-bit processor is 3.0GHz. We used both the Mac OS X (ver. 10.5.5) and the Linux (Ubuntu 64-bit ver. 8.10) operating systems to run our experiments on the Mac Pro platform. HashCS, HashRF and HashRF(p,q) were written in C++ and compiled with gcc 4.2.4 with the `-O3` compiler option. For clustering, we used CLUTO (64-bit ver. 2.1.2a). Hit-MDS version 2 was used for our multidimensional scaling analysis.

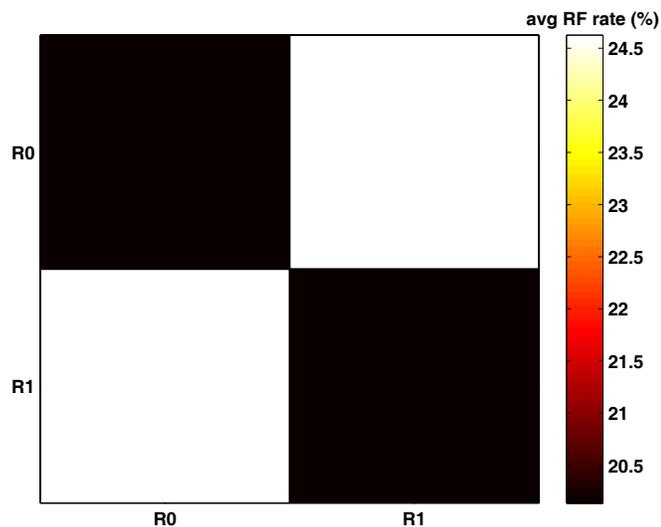
4. Results and Discussion

We use two different techniques to address the following question: “*What is the best way to group a collection of trees?*” Ideally, trees within a group are more similar to each other (smaller RF rate) than to trees outside the group (higher RF rate). The first technique partitions the trees by the Bayesian run that produced them, and computes the average RF rate between the trees from run R_i and R_j . The second technique uses a clustering algorithm to group the trees.

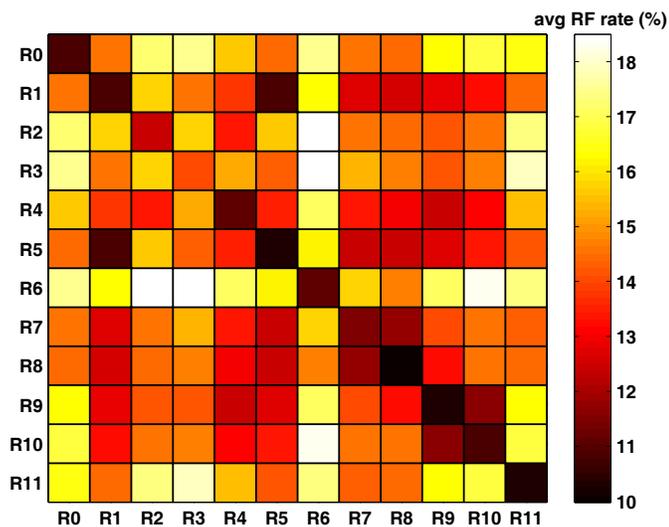
a. Technique #1: Using MrBayes Run Labels to Cluster Trees

Figure 44(a) shows that the average RF rate between trees from the same MrBayes run (cells $(R0, R0)$ and $(R1, R1)$) is approximately 20%. However, trees from different MrBayes runs (cells $(R0, R1)$ and $(R1, R0)$) have a higher RF rate of 25%. Hence, trees from the same run are more similar to each other than to trees from other runs. For the 567 taxa trees, Figure 44(b) shows that trees from run $R0$ (cell $(R0, R0)$) have an average RF rate of 11%, which is much lower than the average RF rate across different runs (cells $(R0, Ri)$, where $i > 0$). Hence, $R0$ trees should be grouped by themselves. The heatmap also suggests that runs $R6$ and $R11$ should also be grouped by themselves. Cells $((R1, R1)$ and $(R1, R5))$ imply that trees from runs $R1$ and $R5$ are equidistant from each other. So, they should be clustered together. Similar observations can be made regarding grouping trees from runs $R7$ and $R8$ as well as runs $R9$ and $R10$.

Overall, our plots clearly demonstrate that there are distinct groups of trees within our tree collections. They should not be summarized as a single group with a single consensus tree. However, the above plots also show another interesting trend. That is, many of the runs do not converge to the same set of trees. In the case of the 150 taxa trees, the two runs are distinct from each other. For the 567 taxa trees, some of the runs converge (e.g., $R1$ and $R5$), but other runs are non-overlapping (e.g., $R0$ and $R6$). In addition to determining how trees should be grouped, this technique can be used to determine how trees from a run relate to each other—especially as it relates to the convergence of multiple runs of a phylogenetic heuristic.



(a) 150 taxa



(b) 567 taxa

Fig. 44. Average RF rate between runs for 150 and 567 taxa tree collections. Cell (R_i, R_j) represents the average RF rate between trees from runs R_i and R_j . The heatmap is symmetric. The range of average RF rate values differs in each plot.

b. Technique #2: Using CLUTO to Cluster Trees

Figure 44 suggests that trees should be grouped according to the MrBayes run that generated them. Next, we use CLUTO to cluster the tree collections. CLUTO has no knowledge of the origin of our phylogenetic trees. It uses its own optimization criteria to find the best clustering of the data. Hence, it is more automated than Technique #1, which requires human intervention to infer patterns to group the trees.

Before performing clustering of trees based on RF rates, MDS (Multidimensional Scaling, See Section 2) visualizations are shown in Figure 45 and Figure 46. MDS takes $20,000 \times 20,000$ and $33,306 \times 33,306$ RF distance matrices for 150 taxa and 567 taxa, respectively and regards the columns as the attributes explaining features of each row (tree). Then it collapses 20,000- and 33,306-dimension feature spaces into 2D spaces. Note that Figure 45 and Figure 46 show only specified numbers of sampled points (trees) for making the plots easy to read (1,000 and 1,665 points for 150 taxa and 567 taxa datasets, respectively). The figures clearly show distinct groups in both 150 taxa and 567 taxa trees. For example, we can definitely find two distinct groups in trees from Figure 45. Again, MDS only provides a visual representation of the pattern of similarities of trees in two-dimensional space. It cannot be used to determine whether points should be in the same cluster or not. Thus, we perform clustering analysis. The first step of our clustering analysis requires selecting the appropriate number of clusters, k , that maximizes the similarity among the trees in the collection. Selecting an appropriate k value is a nontrivial problem, and new techniques are being developed by researchers to automate the selection process. CLUTO's internal similarity measures ($ISim$ and $ESim$) were our primary measure for determining the quality of a clustering. Figure 47 shows the distribution of the $\frac{ISim_{ave}}{ESim_{ave}}$ ratio values across k in our data sets. For the 150 taxa tree collection, two

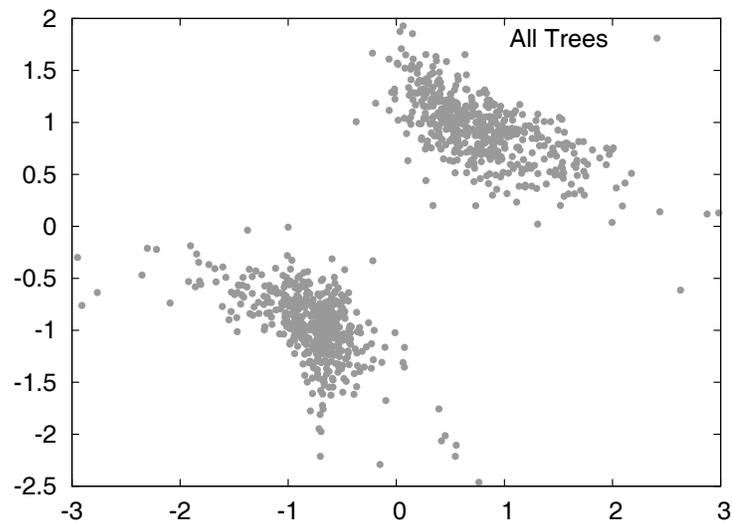


Fig. 45. Visualizing the 150 taxa trees with MDS based on RF rates. 1,000 trees are sampled and shown in this figure.

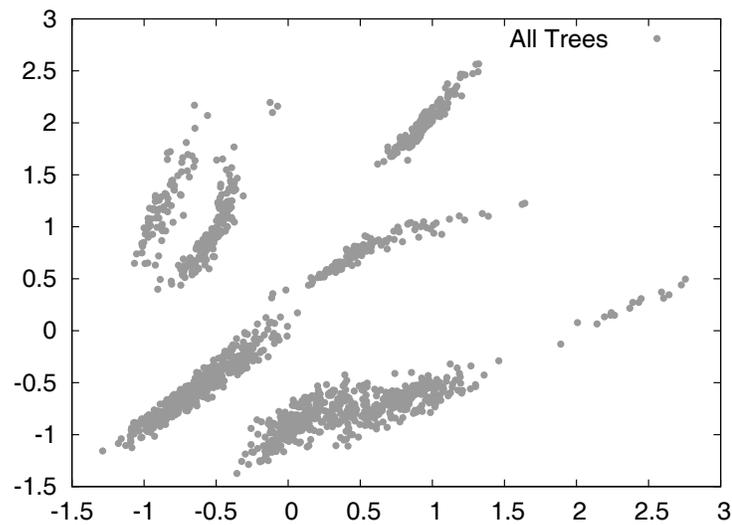


Fig. 46. Visualizing the 567 taxa trees with MDS based on RF rates. 1,665 trees are sampled and shown in this figure.

clusters effectively summarize the 20,000 trees. However, for the 567 taxa dataset, a range of k values of 8, 11, and 14 appear to give equal quality clusterings for the 33,306 trees. We study $k = 8$ thoroughly for the 567 taxa dataset. We choose $k = 8$ since the ratio values for $k = 11$ and $k = 14$ are not that significantly different. Moreover, the average RF rate between clusters is the highest for $k = 8$, when compared to $k = 11$ and $k = 14$ (not shown).

Figure 48 shows the MDS plot for 150 taxa trees and the points (trees) are colored by the cluster labels. The figure shows two distinct groups in the MDS plot are exactly matched with two different clusters of trees. Figure 49 shows our heatmap representation of the average RF rate between the two clusters of trees found by CLUTO for the 150 taxa dataset. Clearly, the trees within a cluster are more similar to each other than trees outside the cluster. Figure 44(a) and Figure 49 agree that the collection of 20,000 trees can be partitioned into two groups. However, do they agree on the composition of the clusters? Table IV examines the composition of Clusters $C0$ and $C1$. Both clusters are composed of half of the 20,000 total trees, and each cluster is composed of trees from a single run. Cluster $C0$ is composed of trees from run $R1$ from the Bayesian analysis. Cluster $C1$ consists of trees from run $R0$. These results correspond exactly to our inferences from Figure 44(a). By clustering, the average majority and strict consensus rates for the two clusters is 89.8% and 36.8%, respectively, which represents a 4.1% and 2.8% increase in the resolution rate over using a single consensus tree. Thus, by clustering the 150 taxa trees into groups, we can better represent the data and construct more resolved consensus trees.

Figure 50 shows the MDS plot for 567 taxa trees when $k = 8$ and demonstrates how cluster labels for trees are dispersed among the groups in MDS plot. For example, the trees of "Cluster 7" are dispersed in almost all groups in the plot which means the trees in "Cluster 7" are more similar with trees in other clusters. Our heatmap

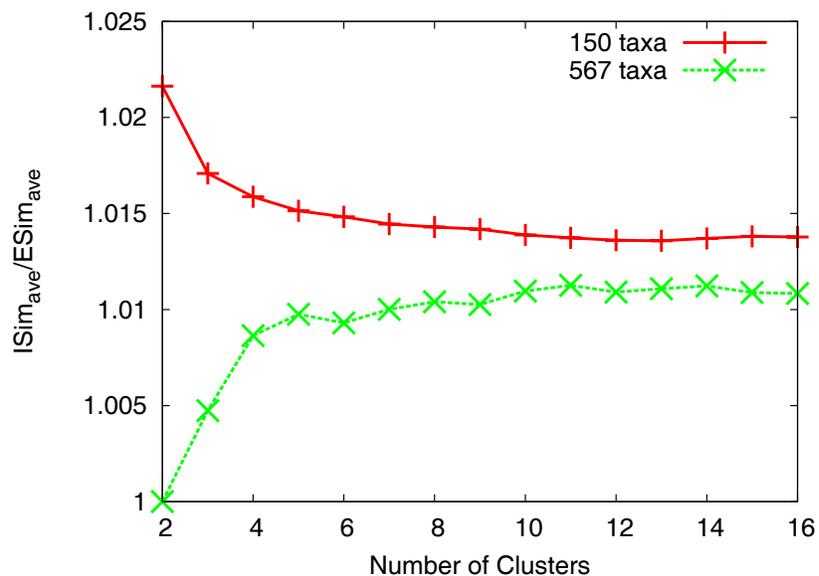


Fig. 47. Selecting the appropriate number of clusters, k . Larger $\frac{ISim_{ave}}{ESim_{ave}}$ values are preferred since they indicate a better clustering of the data.

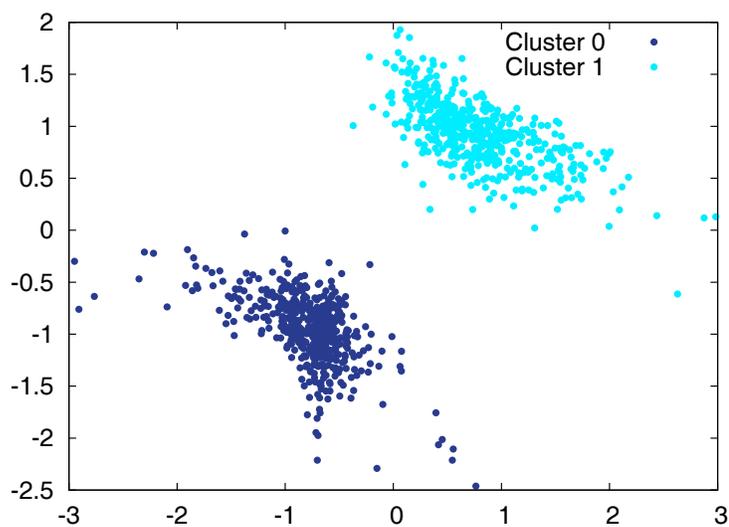


Fig. 48. Visualizing the 150 taxa trees with MDS based on RF rates. Trees are colored with the cluster labels when $k = 2$. 1,000 trees are sampled and shown in this figure.

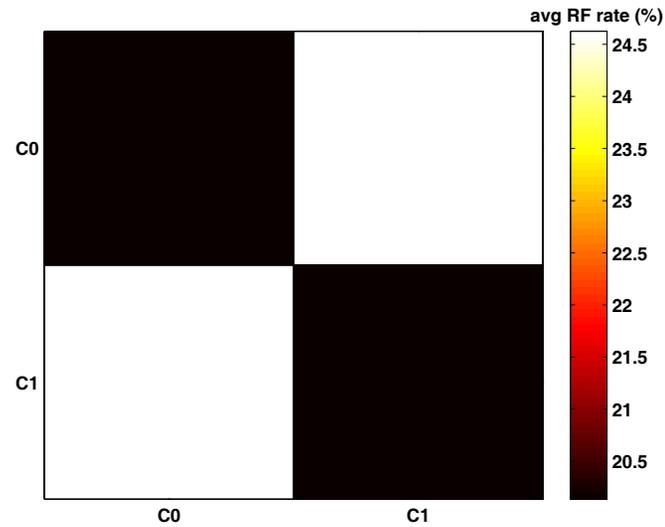


Fig. 49. Average RF rate between the two clusters for the 150 taxa dataset.

Table IV. Detailed information for the $k = 2$ clustering of the 150 taxa trees. For each cluster, we list the number of trees, resolution rates of the majority and strict consensus trees, and run labels of the trees in each cluster.

cluster	number of trees	resolution rate		runs (%)
		majority	strict	
C_0	10,000	90.5	35.4	R1 (100%)
C_1	10,000	89.1	38.1	R0 (100%)

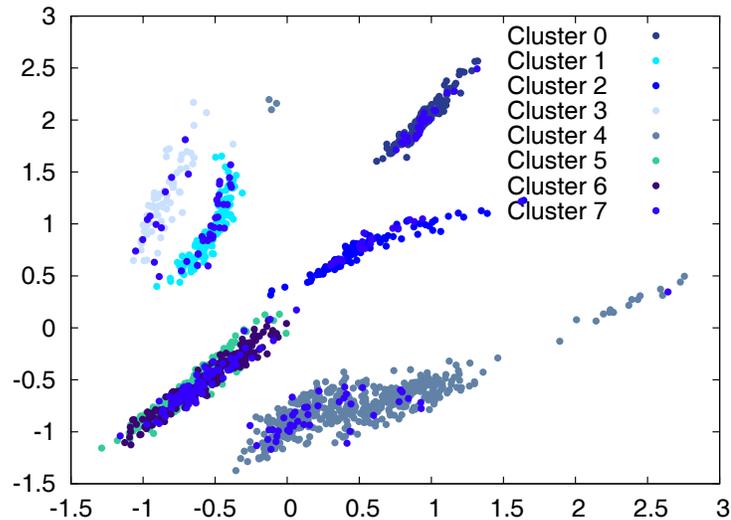


Fig. 50. Visualizing the 567 taxa trees with MDS based on RF rates. Trees are colored with the cluster labels when $k = 8$. 1,665 trees are sampled and shown in this figure.

representation shows the similarity relationship among trees in different clusters more precisely. Figure 51 shows the average RF rate between the clusters of trees found by CLUTO for the 567 taxa dataset. We show the results for 8 and 11 clusters. First, we consider the case when the number of clusters is 8. The trees in clusters C_0 , C_1 , C_2 , C_3 , C_5 , and C_6 are more similar among themselves than to trees outside the cluster. The average RF rate between the trees within these clusters varies from 10% to 12%, which is lower than the average RF rate to trees outside their cluster. For clusters C_4 and C_7 , the dissimilarity between trees within and outside their cluster is not as distinct. For comparison, Figure 51(b) shows the average RF rate heatmap for 11 clusters. In this case, some clusters can be merged together such as C_0 and C_1 , C_2 and C_3 , and C_6 , C_8 , and C_{10} . Hence, CLUTO forces the partitioning of the 33,306 trees into 11 clusters since some of the clusters should be merged together as they are not distinct enough to be in their own cluster. Thus, from our perspective,

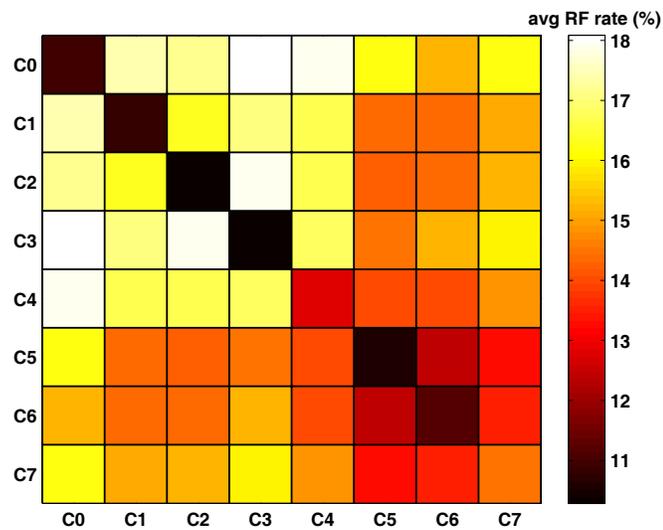
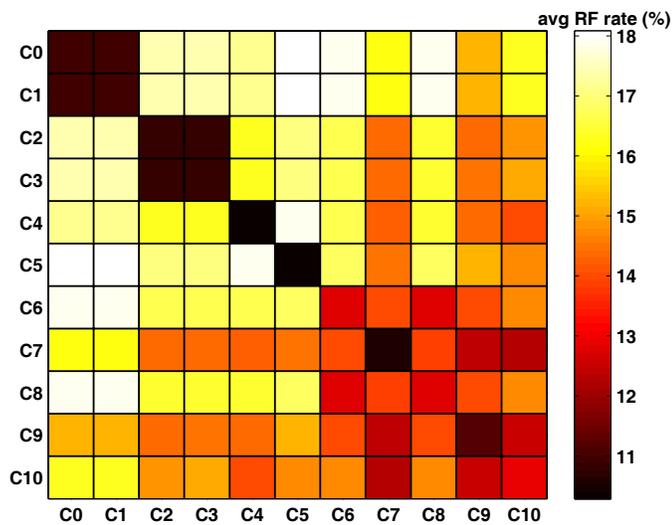
(a) $k = 8$ (b) $k = 11$

Fig. 51. Average RF rate between the clusters for 567 taxa dataset.

$k = 8$ provides a better clustering of the data than $k = 11$.

Table V provides statistics regarding the composition of the eight clusters for the 567 taxa trees. Clusters $C0$, $C1$, $C2$, and $C3$ contain trees from a single run. Clusters $C5$ and $C6$ contain trees from two runs. The trees in cluster $C5$ are essentially equally divided between the runs $R1$ and $R5$. For the cluster $C6$, trees from $R7$ have the largest representation at 58.2%. Cluster $C7$ consists of a hodgepodge of trees from all of the runs. Figure 51(a) shows that the average RF rate between trees in Cluster $C7$ and trees from other clusters is not very distinctive. Hence, the clustering algorithm did not have a good way for dealing with the trees that it collected into cluster $C7$. It may be possible that such trees could be considered outliers as there is not a good way to cluster them effectively. Of course, the results from clusters $C4$ and $C7$ potentially suggest that we should investigate the CLUTO settings we used. For this study, we used the default settings. So, a better clustering could possibly be achieved if we were to tune the clustering settings for our tree collection.

Overall, the clustering results show that the 33,306 trees over 567 taxa can be partitioned into several distinct groups. With eight clusters, the average majority and strict consensus resolution rates improve to 94.4% and 62.9%, respectively, which represents a 1.8% and 11.1% increase in the resolution rate over using a single consensus tree.

5. Summary

Phylogenetic search algorithms (such as Bayesian analysis) often return a large number of trees that require effective summarization techniques to analyze. We develop two techniques to group effectively tree collections consisting of tens of thousands of trees. For t trees of interest, both techniques require a $t \times t$ topological distance matrix to be computed. We apply the Robinson-Foulds distance to compute the

Table V. Detailed information for the $k = 8$ clustering of the 567 taxa trees. For each cluster, we list the number of trees, resolution rates of the majority and strict consensus trees, and run labels of the trees in each cluster.

cluster	number trees	resolution rate		runs (%)
		majority	strict	
<i>C0</i>	2,520	95.2	64.9	<i>R6</i> (100%)
<i>C1</i>	2,674	95.4	65.1	<i>R0</i> (100%)
<i>C2</i>	1,817	95.6	66.8	<i>R11</i> (100%)
<i>C3</i>	1,276	94.3	68.4	<i>R3</i> (100%)
<i>C4</i>	11,258	92.4	58.3	<i>R2</i> (26.4%), <i>R3</i> (13.5%), <i>R4</i> (24.6%), <i>R9</i> (17.7%), <i>R10</i> (17.7%)
<i>C5</i>	5,246	95.2	62.9	<i>R1</i> (51.3%), <i>R5</i> (48.7%)
<i>C6</i>	4,385	94.7	63.7	<i>R7</i> (58.2%), <i>R8</i> (41.8%)
<i>C7</i>	4,130	92.7	53.2	<i>R0</i> (12.2%), <i>R1</i> (11.8%), <i>R2</i> (5.1%), <i>R3</i> (9.2%), <i>R4</i> (5.2%), <i>R5</i> (10.4%), <i>R6</i> (11.3%), <i>R7</i> (10.5%), <i>R8</i> (8.0%), <i>R9</i> (4.0%), <i>R10</i> (4.0%), <i>R11</i> (8.4%)

dissimilarity between two trees. Our first technique groups trees based on the Mr-Bayes run that produced it. This exploratory technique clearly shows that there are distinct groups within the trees and they should not be treated as a single group. Our second technique uses CLUTO, a clustering algorithm designed for large scale data, to cluster the data without any information regarding how the trees were produced. Overall, CLUTO's clustering agreed with the implied clustering from our first technique. Given that the tree collections have distinct groups of trees, multiple consensus trees are needed to more effectively summarize the tree collections. Multiple consensus trees represent trends that would have been hidden by a single consensus tree, but were strong enough to have appeared in a subset of the trees.

Clustering large tree collections shows underlying trends that are not made visible by constructing a single consensus tree. Moreover, clusters can help identify whether a phylogenetic search has converged for a particular run or across different runs.

For example, our techniques showed that several of the MrBayes runs converged to different trees and thus to different areas of tree space for our 150 taxa trees. For the 567 taxa case, some of runs overlapped with others. Thus, by looking at the data from a run perspective, it is interesting to see that different runs have different behaviors. Depending upon the goals of the search, multiple runs can be used to navigate different areas of the exponentially-sized tree space. Or, multiple runs can be used to further explore a fruitful area of tree space. By grouping all of the data across the runs into a single cluster, such trends in the data are lost.

Future research directions include using weighted RF distance for clustering and analyzing phylogenetic trees. Other distance metrics such as quartet distance [16] can also be used to perform a similar analysis of trees. Such an approach will also be useful in terms of comparing RF and quartet distance clusterings.

CHAPTER VIII

CONCLUSIONS AND FUTURE WORK

A. Conclusions

An evolutionary tree is a model of the evolutionary history for a set of species. A single phylogenetic reconstruction method often produces many different trees for the same set of organisms. Furthermore, different phylogenetic reconstruction methods potentially generate different trees. A clearer picture of the similarity or dissimilarity among the collections of trees would help researchers better understand the evolutionary history of the species of interest.

Basic notions on evolutionary trees and hash technique are introduced in Chapter II and Chapter IV. Chapter IV also describes the hash-based method to store evolutionary trees and to compute consensus trees and RF distance matrix based on the hash tables. In Chapter V, we design and implement a fast hash-based consensus algorithm called HashCS for computing strict and majority trees, and evaluate its performance theoretically and experimentally. Consensus trees are commonly used for analyzing phylogenetic trees. However, but summarizing large number of trees (which are believed to be equally plausible) into a single consensus tree loses valuable evolutionary information resides in the trees. All-to-all relationships among the trees in a form of distance matrix can represent the relationships between trees and also provides researchers with tremendous data-mining opportunity for understanding the relationships depicted by the collection of trees. In Chapter VI, we introduce the Robinson-Foulds (RF) distance metric to measure the topological distances between phylogenetic trees and implemented a series of algorithms (HashRF, WHashRF, and HashRF(p,q)) for computing the RF distance matrix.

Based on our biological and artificial sets of phylogenetic trees, we evaluate the performance of our implementations in the experimental analysis sections in Chapter V and Chapter VI. Our results show HashCS is the fastest available consensus tree algorithm. HashCS is over 100 times faster than MrBayes and approximately 4 and 1.8 times faster than Phylip and PAUP*, respectively on 567 taxa, 16,384 biological trees. With 150 taxa trees, the speedup over PAUP* is 1.2 times. On the artificial trees with 1,024 taxa, HashCS is over 300 times faster than PAUP* when constructing 0% consensus tree and 2.3 times faster than PAUP* when the consensus tree is 100% resolved. Furthermore, we also analyze the behavior of our HashCS algorithm by using artificial tree collections with a varying number of shared bipartitions. We found that PAUP*, the most popular commercial software package and second-fast consensus approach, is seriously impacted by the amount of bipartition sharing in the collection of trees. PAUP*'s performance improves as the trees in the collection become more similar. HashCS's performance, on the other hand, is not impacted by the number of shared bipartitions.

For computing RF distances, our experimental results prove our approaches are the most efficient in running time and memory space requirement. To compute RF distance of 567 taxa, 4,096 biological trees, HashRF is 200 times faster than PAUP* and 2.1 times faster than PGM-Hashed. With 150 taxa trees, HashRF is 48 and 2.2 times faster than PAUP* and PGM-Hashed, respectively to compute RF distance of 16,384 trees. On the artificial trees of 2,048 taxa, 2,048 trees, the speedup of HashRF algorithm over PGM-Hashed is 12 times in the best case (with 0% bipartition sharing) and 1.3 times in the worst case (with 100% bipartition sharing). Using the artificial trees, we show the performance of PGM-Hashed algorithm, the closest competitor of HashRF, decreases when trees shares less amount of shared bipartitions. Even when the worst case for HashRF, our algorithm is faster than PGM-Hashed.

In Chapter VII introduces two applications for our HashCS and HashRF implementations. First, we use novel methods to assess the quality of two Maximum Parsimony-based phylogenetic search algorithms, Pauprat and Rec-I-DCM3. We show that topological distance methods such as Robinson-Foulds are more effective than parsimony scores at identifying heterogeneity within a collection of trees. Our entropy-based methods and heatmap visualizations show that Pauprat identifies more diverse trees than Rec-I-DCM3 in larger datasets. This ability suggests that Pauprat, while a slower heuristic, has more opportunities to escape local optima. Tree topology is useful in identifying diversity in a collection of equally parsimonious trees. Furthermore, slower heuristics can produce different candidate trees in the course of their search. Lastly, our results imply that more powerful heuristics can be designed by combining tree topology with scores in the optimization criteria. Our entropy-based methods shed light on the behavior of individual heuristics, thus allowing for the design of better heuristics.

Second, we develop two techniques to group effectively tree collections consisting of tens of thousands of trees. Instead of using a single consensus trees, we develop techniques to analyze two very large tree collections obtained from biologists. We show that there are distinct clusters of trees in our tree collections. We use two different techniques to identify distinct tree groups. Our first technique partitions the trees by the Bayesian run that produced them. The second technique uses a clustering algorithm to group the trees. Both techniques show that considering each tree collection as a single entity is a significant under-representation of the data. Partitioning the trees into distinct groups and summarizing each group separately is a better representation of the data. Furthermore, our results show that the benefits of our approach are better consensus trees as well as insightful information regarding the convergence behavior of the multiple Bayesian runs.

B. Future Work

Our research goal is to provide researchers with a set of efficient algorithms for analyzing large collections of phylogenetic trees. Based on our fast implementations, we'll develop a technique to visually represent the information among the collection of trees. We expect that our approaches will produce deeper insights into the relationships contained in phylogenetic trees and thus provide new ways for scientist to understand the overall evolutionary history of a collection of organisms. We also plan to incorporate our implementations into popular heuristics such as MrBayes so that we can analyze not only on the final phylogenetic estimation, but also on the trees in the middle of inference in real-time. Often times, phylogenetic heuristics do not converge to a robust estimation. Incorporating tree topology as a way to guide phylogenetic search will greatly help researchers to find more accurate trees.

Future research directions include computing other distance metrics. Given a set of phylogenetic trees, different distance metrics could reveal valuable additional information from the trees. Among other metrics, NNI (Nearest Neighbor Interchange), SPR (Subtree Pruning and Regrafting) and TBR (Tree Bisection and Reconnection) distances will be in the scope of our future work. Other distance metrics such as quartet distance [16] can also be used to perform a similar analysis of trees.

Our hash technique can also be applied for the problem of approximating the NNI distance. The NNI distance was first introduced by Robinson [120] and Moore [121] and extensively studied by others [66, 122, 123]. The NNI distance counts the minimum number of NNI operations required to transform one phylogenetic tree to the other. The metric provide different measure than RF distance to compare phylogenetic trees. The NNI distance metric is one of the most widely used tree rearrangement operations in phylogenetic heuristic and particularly useful for studying islands

of trees [64].

From the algorithmic point of view, computing the NNI distance is quite different from computing RF distance. Computing the NNI distance has remained very challenging, especially since it is an NP-Hard problem [65, 124, 125]. Thus, we have to depend upon approximation algorithms to compute the NNI distance. However, most approaches to approximate the NNI distance first need to distinguish *non-shared* (or *shared*) edges between input trees. We've already found the efficiency of our hash-based method to deal with the edge information in trees. Our future work lies in using our hashing technique to find the *non-shared* (or *shared*) edge information. We're working on an efficient approximation algorithm, called HashNNI and compute the approximated NNI distance. Using HashNNI and HashRF, we could compute both NNI and RF distances for sets of trees and study the relationship between the metrics. Finally, in addition to NNI, we will also consider SPR and TBR distances.

Generation of MRP (Matrix Representation with Parsimony) [126, 127, 128, 129] must be another application of our hashing techniques. MRP is a method that takes as input a collection of source trees, recodes them as binary matrices, and returns a tree that is closest to the source trees using a parsimony criterion. The MRP is the most popular method to construct supertrees from input trees which have nonidentical but overlapping sets of leaves [5, 129, 130]. Generating MRP to construct supertree involves determining which bipartitions are shared between the trees. Our hash table can also be applied for fast generation of MRP for constructing supertrees.

Additionally, we'll use other distance metrics such as weighted RF, NNI, SPR, TBR, and quartet distances for clustering and analyzing phylogenetic trees. Nowadays, dimensionality reduction techniques are widely used for the area of phylogenetic tree analysis [132, 134]. As our clustering approach is based on large-scale ($t \times t$) dis-

tance matrices which includes t size vectors, clustering through dimensionality reduction could increase the quality of cluster and dramatically shorten the running time for clustering. We believe that such an approach should disclose more evolutionary information from the phylogenetic trees.

REFERENCES

- [1] A. Dress, K. T. Huber, and V. Moulton, “Metric spaces in pure and applied mathematics,” *Documenta Mathematica LSU*, pp. 121–139, 2001.
- [2] J. Felsenstein, *Inferring Phylogenies*, Sunderland, Massachusetts: Sinauer Associates, 2005.
- [3] D. Penny, M. D. Hendy, and M. A. Steel, “Progress with methods for constructing evolutionary trees,” *Trends in Ecology and Evolution*, vol. 7, pp. 73–79, 1992.
- [4] E. N. Adams, “Consensus techniques and the comparison of taxonomic trees,” *Systematic Zoology*, vol. 21, pp. 390–397, 1972.
- [5] B. R. Baum, “Combining trees as a way of combining data sets for phylogenetic inference, and the desirability of combining gene trees,” *Taxon*, vol. 41, no. 1, pp. 3–10, 1992.
- [6] T. Berger-Wolf and T. L. Williams, “An experimental evaluation of phylogenetic consensus methods,” Department of Computer Science, University of New Mexico, Albuquerque, NM, Tech. Rep. TR-CS-2003-19, 2003.
- [7] R. S. Boyer, W. A. Hunt Jr., and S. Nelesen, “A compressed format for collections of phylogenetic trees and improved consensus performance,” in *Proc. 5th Int’l Workshop Algorithms in Bioinformatics (WABI’05)*. vol. 3692, pp. 353–364, 2005.
- [8] G. S. Brodal, R. Fagerberg, and C. N. S. Pedersen, “Computing the quartet distance between evolutionary trees in time $O(n \log^2 n)$,” *Lecture Notes in Computer Science*, vol. 2223, pp. 731–742, 2001.

- [9] J. Hein, T. Jiang, L. Wang, and K. Zhang, “On the complexity of comparing evolutionary trees,” *Discrete Applied Mathematics*, vol. 71, no. 1-3, pp. 153–169, 1996.
- [10] B. DasGupta, X. He, T. Jiang, M. Li, J. Tromp, and L. Zhang, “On distances between phylogenetic trees,” in *Proc. the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA’97)*, Philadelphia, PA, USA, 1997, pp. 427–436, 1997.
- [11] L. R. Foulds and R. L. Graham, “The Steiner problem in phylogeny is NP-complete,” *Advances in Applied Mathematics*, vol. 3, pp. 43–49, 1982.
- [12] P. A. Goloboff, “Calculating spr distances between trees,” *Cladistics*, vol. 23, pp. 1–7, 2007.
- [13] T. Mailund, (2007, November) “SplitDist—calculatin split-distances for sets of trees,” [online]. Available: <http://www.daimi.au.dk/mailund/split-dist.html>.
- [14] T. Mailund and C. N. S. Pedersen, “QDist—quartet distance between evolutionary trees,” *Bioinformatics*, vol. 20, no. 10, pp. 1636–1637, 2004.
- [15] N. D. Pattengale, E. J. Gottlieb, and B. M. E. Moret, “Efficiently computing the Robinson-Foulds metric,” *Journal of Computational Biology*, vol. 14, no. 6, pp. 724–735, 2007.
- [16] 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 *Proc. Fifth Asia Pacific Bioinformatics Conference (APBC’07)*, pp. 91–100, 2007.

- [17] M. J. Brauer, M. T. Holder, L. A. Pries, D. J. Zwickl, P. O. Lewis, and D. M. Hillis, "Genetic algorithms and parallel processing in maximum-likelihood phylogeny inference," *Molecular Biology and Evolution*, vol. 19, no. 10, pp. 1717–1726, 2002.
- [18] M. A. Charleston, "Hitch-hiking: A parallel heuristic search strategy, applied to the phylogeny problem," *Journal of Computational Biology*, vol. 8, no. 1, pp. 79–91, 2001.
- [19] C. Coarfa, Y. Dotsenko, J. M. Mellor-Crummey, L. Nakhleh, and U. Roshan, "Prec-i-dcm3: A parallel framework for fast and accurate large scale phylogeny reconstruction," in *Proc. 11th International Conference on Parallel and Distributed Systems (ICPADS 2005)*. pp. 346–350, 2005.
- [20] Y. Dotsenko, C. Coarfa, L. Nakhleh, J. Mellor-Crummey, and U. Roshan, "PRec-I-DCM3: A parallel framework for fast and accurate large scale phylogeny reconstruction," *International Journal on Bioinformatics Research and Applications (IJBRA)*, 2006, in press.
- [21] Z. Du, A. Stamatakis, F. Lin, U. Roshan, and L. Nakhleh, "Parallel divide-and-conquer phylogeny reconstruction by maximum likelihood," in *Proc. 2005 International Conference on High-Performance Computing and Communications (HPCC'05)*, pp. 346–350, 2005.
- [22] P. A. Goloboff, S. Farris, and K. Nixon, (2008, October) "TNT (tree analysis using new technology)," [online]. Available: <http://www.cladistics.com/aboutTNT.html>.
- [23] P. A. Goloboff, "Methods for faster parsimony analysis," *Cladistics*, vol. 12, no. 3, pp. 199–220, 1996.

- [24] G. Savva, J. L. Dicks, and I. N. Roberts, “Current approaches to whole genome phylogenetic analysis,” *Briefings in Bioinformatics*, vol. 4, no. 1, pp. 63–74, 2003.
- [25] A. P. Stamatakis, “Parallel inference of a 10,000-taxon phylogeny with maximum likelihood,” in *Proc. 10th International Euro-Par Conference (Euro-Par 2004)*. Lecture Notes in Computer Science, vol. 3149, pp. 997–1004, 2004.
- [26] D. Sikes and D. O. Lewis, (2009, November) “PAUPRat: PAUP* implementation of the parsimony ratchet,” [online]. Available: <http://www.ucalgary.ca/dsikes/software2.htm>.
- [27] O. Bininda-Emonds, (2009, November) “Ratchet implementation in PAUP*4.0b10,” [online]. Available: <http://www.tierzucht.tum.de:8080/WWW/Homepages/Bininda-Emonds>.
- [28] 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 *Proc. IEEE Computer Society Bioinformatics Conference (CSB 2004)*. pp. 98–109, 2004.
- [29] S. P. Brooks and A. Gelman, “General methods for monitoring convergence of iterative simulations,” *Journal of Computational and Graphical Statistics*, vol. 7, pp. 434 – 455, 1998.
- [30] A. Gelman and D. B. Rubin, “Inference from iterative simulation using multiple sequences,” *Statistical Science*, vol. 7, no. 4, pp. 457 – 472, 1992.
- [31] W. Hennig, *Phylogenetic Systematics*, Champaign, IL: University of Illinois Press, 1999.

- [32] D. A. Bader, B. M. E. Moret, and L. Vawter, “Industrial applications of high-performance computing for phylogeny reconstruction,” in *Proc. SPIE Commercial Applications for High-Performance Computing*, H. Siegel, Ed., Denver, CO, Aug. 2001, SPIE, vol. 4528, pp. 159–168, Aug. 2001.
- [33] 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,” *PNAS*, vol. 99, no. 2, pp. 14292–14297, 2002.
- [34] M. Ruvolo, “Molecular phylogeny of the hominoids: Inferences from multiple independent dna sequence data sets,” *Molecular Biology and Evolution*, vol. 14, pp. 248–265, 1997.
- [35] P. O. Lewis, “A genetic algorithm for maximum-likelihood phylogeny inference using nucleotide sequence data,” *Molecular Biology and Evolution*, vol. 15, no. 3, pp. 277–283, 1998.
- [36] A. Goeffon, J.-M. Richer, and J.-K. Hao, “Progressive tree neighborhood applied to the maximum parsimony problem,” *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, vol. 5, no. 1, pp. 136–145, 2008.
- [37] W. Hordijk and O. Gascuel, “Improving the efficiency of spr moves in phylogenetic tree search methods based on maximum likelihood,” *Bioinformatics*, vol. 21, no. 24, pp. 4338–4347, 2005.
- [38] G. J. Olsen, H. Matsuda, R. Hagstrom, and R. Overbeek, “fastDNAml: A tool for construction of phylogenetic trees of DNA sequences using maximum likelihood,” *Computer Applications in the Biosciences*, vol. 10, pp. 41–48, 1994.

- [39] U. Roshan, (2005, November) “Detailed experimental results on the performance of Rec-I-DCM3 as presented in CSB’04,” [online]. Available: http://www.cs.njit.edu/usman/dcm3/recidcm3_csb04_data.html.
- [40] M. A. Steel, “The maximum likelihood point for a phylogenetic tree is not unique,” *Systematic Biology*, vol. 43, no. 4, pp. 560–564, 1994.
- [41] T. L. Williams, B. M. E. M. T. Berger-Wolf, U. Roshan, and T. Warnow, “The relationship between maximum parsimony scores and phylogenetic tree topologies,” Department of Computer Science, The University of New Mexico, Albuquerque, NM, Tech. Rep. TR-CS-2004-04, 2004.
- [42] D. Hillis, “Inferring complex phylogenies,” *Nature*, vol. 383, pp. 130–131, 1996.
- [43] D. Hillis, C. Moritz, and B. Mable, *Molecular Systematics*, Boston: Sinauer Pub., 1996.
- [44] R. D. M. Page and E. C. Holmes, *Molecular Evolution: A Phylogenetic Approach*, Hoboken, NJ: Wiley-Blackwell, 1998.
- [45] M. Holder and P. O. Lewis, “Phylogeny estimation: Traditional and bayesian approaches,” *Nature Reviews Genetics*, vol. 4, pp. 275–284, 2003.
- [46] J. P. Huelsenbeck and F. Ronquist, “MrBayes: Bayesian inference of phylogenetic trees,” *Bioinformatics*, vol. 17, no. 8, pp. 754–755, 2001.
- [47] J. P. Huelsenbeck, F. Ronquist, R. Nielsen, and J. P. Bollback, “Bayesian inference of phylogeny and its impact on evolutionary biology,” *Science*, vol. 294, pp. 2310–2314, 2001.

- [48] J. Felsenstein, (2008, November) “Confidence limits on phylogenies: An approach using the bootstrap,” *Evolution*, vol. 39, pp. 783–791, [online]. Available: <http://dx.doi.org/10.2307/2408678>.
- [49] D. Hillis and J. J. Bull, “An empirical-test of bootstrapping as a method for assessing confidence in phylogenetic analysis,” *Systematic Biology*, vol. 42, pp. 182–192, 1996.
- [50] S. B. Holmes, “Bootstrapping phylogenetic trees: Theory and methods,” *Statistical Science*, vol. 18, no. 2, pp. 241–255, 2003.
- [51] C. Notredame, “Recent rogresses in multiple sequence alignment: A survey,” *Pharmacogenomics*, vol. 3, no. 1, pp. 1–14, 2002.
- [52] A. Phillips, D. Janies, and W. Wheeler, “Multiple sequence alignment in phylogenetic analysis,” *Molecular Biology and Evolution*, vol. 16, no. 3, pp. 317–330, 2000.
- [53] J. D. Thompson, F. Plewniak, and O. Poch, “A comprehensive comparison of multiple sequence alignment programs,” *Nucleic Acids Research*, vol. 27, no. 13, pp. 2682–2690, 1999.
- [54] J. D. Thompson, D. Higgins, and T. Gibson, “CLUSTAL-W: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice,” *Nucleic Acids Research*, vol. 22, pp. 4673–4690, 1994.
- [55] C. Notredame, D. G. Higgins, and J. Heringa, “T-Coffee: A novel method for fast and accurate multiple sequence alignment programs,” *FEMS Microbiology Letters*, vol. 302, pp. 205–217, 2000.

- [56] C. Notredame, L. Holm, and D. G. Higgins, “COFFEE: An objective function for multiple sequence alignments,” *Bioinformatics*, vol. 14, pp. 407–422, 1998.
- [57] T. Hall, (2009, July) “Bioedit: Biological sequence alignment editor,” *Ibis BioSciences*, 2009, [online]. Available: <http://www.mbio.ncsu.edu/BioEdit/bioedit.html>.
- [58] W. J. Bruno, N. Socci, and A. L. Halpern, “Weighted neighbor joining: A likelihood-based approach to distance-based phylogeny reconstruction,” *Molecular Biology and Evolution*, vol. 17, no. 1, pp. 189–197, 2000.
- [59] N. Saitou and M. Nei, “The neighbor-joining method: A new method for reconstructing phylogenetic trees,” *Molecular Biology and Evolution*, vol. 4, no. 406–425, 1987.
- [60] J. A. Studier and K. J. Keppler, “A note on the neighbor-joining algorithm of Saitou and Nei,” *Molecular Biology and Evolution*, vol. 5, pp. 729–731, 1988.
- [61] J. P. Huelsenbeck, B. Rannala, and J. P. Masly, “Accommodating phylogenetic uncertainty in evolutionary studies,” *Science*, vol. 288, pp. 2349–2350, 2000.
- [62] J. P. Huelsenbeck, B. Larget, R. E. Miller, and F. Ronquist, “Potential applications and pitfalls of bayesian inference of phylogeny,” *Systematic Biology*, vol. 51, pp. 673–688, 2002.
- [63] D. L. Swofford, G. K. Olsen, P. J. Waddell, and D. M. Hillis, “Phylogeny reconstruction,” in *Molecular Systematics*, pp. 407–514. 1996.
- [64] D. R. Maddison, “The discovery and importance of multiple islands of most parsimonious trees,” *Systematic Biology*, vol. 42, no. 2, pp. 200–210, 1991.

- [65] B. DasGupta, X. He, T. Jiang, M. Li, and J. Tromp, “On the linear-cost subtree-transfer distance between phylogenetic trees,” Rutgers, New Jersey, Tech. Rep. 97-18, May 9 1997.
- [66] W.-K. Hon, M.-Y. Kao, and T.-W. Lam, “Improved phylogeny comparisons: Non-shared edges, nearest neighbor interchanges, and subtree transfers,” in *Proc. the 11th International Conference on Algorithms and Computation (ISAAC '00)*, London, UK, 2000, pp. 527–538, 2000.
- [67] O. R. P. Bininda-Emonds, “MRP supertree construction in the consensus setting,” in *Bioconsensus*, M. Janowitz, F. Lapointe, F. McMorris, B. Mirkin, and F. Roberts, Eds., vol. 61 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pp. 231–240. 2001.
- [68] D. Bryant, “A classification of consensus methods for phylogenetics,” *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, vol. 61, pp. 163–184, 2003.
- [69] W. H. E. Day, “Optimal algorithms for comparing trees with labeled leaves,” *Journal of Classification*, vol. 2, pp. 7–28, 1985.
- [70] T. A. Standish, *Data Structures Techniques*, Reading, MA: Addison-Wesley, 1980.
- [71] N. Amenta, F. Clarke, and K. S. John, “A linear-time majority tree algorithm,” *Workshop on Algorithms in Bioinformatics*, vol. 2168, pp. 216–227, 2003.
- [72] R. S. 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] W. A. Hunt Jr. and S. M. Nelesen, “Phylogenetic trees in ACL2,” in *Proc. 6th Int’l Conf. on ACL2 Theorem Prover and its Applications (ACL2’06)*, New York, USA, 2006, pp. 99–102, 2006.
- [74] J. Allen, *Anatomy of Lisp.*, Columbus, OH: McGraw Hill, 1978.
- [75] D. L. Swofford, (2009, July) “PAUP*: Phylogenetic analysis using parsimony (and other methods),” [online]. Available: <http://paup.csit.fsu.edu/>.
- [76] P. A. Goloboff, “Analyzing large data sets in reasonable times: Solutions for composite optima,” *Cladistics*, vol. 15, pp. 415–428, 1999.
- [77] K. C. Nixon, “The parsimony ratchet, a new method for rapid parsimony analysis,” *Cladistics*, vol. 15, pp. 407–414, 1999.
- [78] K. Rice, M. Donoghue, and R. Olmstead, “Analyzing large datasets: *rbcL* 500 revisited,” *Systematic Biology*, vol. 46(3), pp. 554–563, 1997.
- [79] J. Felsenstein, (2008, July) “The newick ttree format,” [online]. Available: <http://evolution.genetics.washington.edu/phylip/newicktree.html>.
- [80] C. Stockham, L. S. Wang, and T. Warnow, “Statistically based postprocessing of phylogenetic analysis by cluste ring,” in *Proc. 10th Int’l Conf. on Intelligent Systems for Molecular Biology (ISMB’02)*, pp. 285–293, 2002.
- [81] 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.

- [82] W. J. Murphy, E. Eizirik, S. J. O'Brien, O. Madsen, M. Scally, C. J. Douady, E. Teeling, O. A. Ryder, M. J. Stanhope, W. W. de Jong, and M. S. Springer, "Resolution of the early placental mammal radiation using Bayesian phylogenetics," *Science*, vol. 294, pp. 2348–2351, 2001.
- [83] T. M. W. Nye, "Trees of trees: An approach to comparing multiple alternative phylogenies," *Systematic Biology*, vol. 57, no. 5, pp. 785–794, 2008.
- [84] C. Bonnard, V. Berry, and N. Lartillot, "Multipolar consensus for phylogenetic trees," *Systematic Biology*, vol. 55, pp. 837–843, 2006.
- [85] D. Hofheinz and E. Kiltz, "Programmable hash functions and their applications," in *Advances in Cryptology – CRYPTO 2008*, pp. 21–38. 2008.
- [86] Y. Hirai, T. Kurokawa, S. Matsuo, H. Tanaka, and A. Yamamura, "Classification of hash functions suitable for real-life systems," *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, vol. E91-A, no. 1, pp. 64–73, 2008.
- [87] S. K. Debray, W. Evans, R. Muth, and B. De Sutter, "Compiler techniques for code compaction," *ACM Transactions on Programming Languages and Systems*, vol. 22, no. 2, pp. 378–415, 2000.
- [88] S. Muchnick, *Advanced Compiler Design and Implementation*, San Francisco, CA: Morgan Kaufmann, August 1997.
- [89] D. Bovet and M. Cesati, *Understanding the Linux Kernel*, Sebastopol, CA: O'Reilly Media, Inc., 2005.
- [90] R. Love, *Linux Kernel Development*, Toronto, Ontario, Canada: Sams Publishing, 2003.

- [91] X. Lai and J. L. Massey, “Hash functions based on block ciphers,” *Lecture Notes in Computer Science*, vol. 658, pp. 55–70, 1993.
- [92] D. R. Stinson, “Some observations on the theory of cryptographic hash functions,” *Des. Codes Cryptography*, vol. 38, no. 2, pp. 259–277, 2006.
- [93] D. E. Knuth, *The Art of Computer Programming: Sorting and Searching*, vol. 3, Reading, MA: Addison-Wesley Publishing Company, 1973.
- [94] P. Buneman, “The recovery of trees from measures of dissimilarity,” in *Mathematics in Archeological and Historical Sciences*, F. R. Hodson, D. G. Kendall, and P. Tautu, Eds., pp. 387–395. Edinburgh: Edinburgh University Press, 1971.
- [95] D. Gusfield, “Efficient algorithms for inferring evolutionary trees,” *Networks*, vol. 21, pp. 19–28, 1991.
- [96] S.-J. Sul and T. L. Williams, “An experimental analysis of consensus tree algorithms for large-scale tree collections,” in *Proc. the 5th International Symposium on Bioinformatics Research and Applications (ISBRA '09)*, vol. 5542, pp. 100–111, 2009.
- [97] S.-J. Sul and T. L. Williams, “Fast hashing algorithms to summarize large collections of evolutionary trees,” College Station: Department of Computer Science, Texas A&M University, Tech. Rep. TR-CS-2008-6-1, 2008.
- [98] D. S. Johnson, “A theoretician’s guide to the experimental analysis of algorithms,” *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges*, vol. 59, pp. 215–250, 2002.
- [99] B. M. E. Moret, “Towards a discipline of experimental algorithmics,” *Data*

- Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges*, vol. 59, pp. 197–213, 2002.
- [100] B. M. E. Moret and H. D. Shapiro, “Algorithms and experiments: the new (and the old) methodology,” *Journal of Universal Computer Science*, vol. 7, no. 5, pp. 434–446, 2001.
- [101] B. M. E. Moret and T. Warnow, “Reconstructing optimal phylogenetic trees: A challenge in experimental algorithmics,” *Experimental Algorithmics*, vol. 2547, pp. 163–180, 2002.
- [102] 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.
- [103] 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.
- [104] 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.
- [105] N. Bortolussi, E. Durand, M. Blum, and O. François, “apTreeshape: Statistical analysis of phylogenetic tree shape,” *Bioinformatics*, vol. 22, no. 3, pp. 363–364, Feb 2006.
- [106] 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)*, vol. 5251, pp. 123–134, 2008.

- [107] S.-J. Sul and T. L. Williams, “A randomized algorithm for comparing sets of phylogenetic trees,” in *Proc. Fifth Asia Pacific Bioinformatics Conference (APBC'07)*, pp. 121–130, 2007.
- [108] S.-J. Sul and T. L. Williams, “An experimental analysis of robinson-foulds distance matrix algorithms,” in *ESA*, D. Halperin and K. Mehlhorn, Eds. vol. 5193 of *Lecture Notes in Computer Science*, pp. 793–804, 2008.
- [109] D. F. Robinson and L. R. Foulds, “Comparison of weighted labelled trees,” in *Proc. Sixth Austral. Conf.*, vol. 748, pp. 119–126, 1979.
- [110] S.-J. Sul, S. Matthews, and T. L. Williams, “New approaches to compare phylogenetic search heuristics,” in *Proc. IEEE International Conference on Bioinformatics and Biomedicine (BIBM'08)*, pp. 239–245, 2008.
- [111] S.-J. Sul, S. Matthews, and T. L. Williams, “Using tree diversity to compare phylogenetic heuristics,” *BMC Bioinformatics*, vol. 10, no. S-4, 2009.
- [112] D. Huson, S. Nettles, and T. Warnow, “Disk-covering, a fast-converging method for phylogenetic tree reconstruction,” *Journal of Computational Biology*, vol. 6, pp. 369–386, 1999.
- [113] D. Huson, L. Vawter, and T. Warnow, “Solving large scale phylogenetic problems using DCM2,” in *Proc. 7th Int'l Conf. on Intelligent Systems for Molecular Biology (ISMB'99)*. pp. 118–129, 1999.
- [114] L. Nakhleh, U. Roshan, K. St. John, J. Sun, and T. Warnow, “Designing fast converging phylogenetic methods,” in *Proc. 9th Int'l Conf. on Intelligent Systems for Molecular Biology (ISMB'01)*. vol. 17, pp. S190–S198, 2001.

- [115] A. R. Deans, J. J. Gillespie, and M. J. Yoder, “An evaluation of ensign wasp classification (Hymenoptera: Evanildae) based on molecular data and insights from ribosomal rna secondary structure,” *Systematic Entomology*, vol. 31, pp. 517–528, 2006.
- [116] J. J. Gillespie, C. H. McKenna, M. J. Yoder, R. R. Gutell, J. S. Johnston, J. Kathirithamby, and A. I. Cognato, “Assessing the odd secondary structural properties of nuclear small subunit ribosomal rna sequences (18s) of the twisted-wing parasites (Insecta: Strepsiptera),” *Insect Molecular Biology*, vol. 15, pp. 625–643, 2005.
- [117] F. Ronquist and J. P. Huelsenbeck, “Mrbayes 3: Bayesian phylogenetic inference under mixed models,” *Bioinformatics*, vol. 19, no. 12, pp. 1572–1574, August 2003.
- [118] G. Karypis, (2008, June) “CLUTO—software for clustering high-dimensional datasets,” [online]. Available: <http://glaros.dtc.umn.edu/gkhome/cluto/cluto/overview>.
- [119] D. F. Robinson and L. R. Foulds, “Comparison of phylogenetic trees,” *Mathematical Biosciences*, vol. 53, pp. 131–147, 1981.
- [120] D. F. Robinson, “Comparison of labeled trees with valency three,” *Journal of Combinatorial Theory*, vol. 11, pp. 105–119, 1971.
- [121] G. W. Moore, M. Goodman, and J. Barnabas, “An iterative approach from the standpoint of the additive hypothesis to the dendrogram problem posed by molecular data sets,” *Journal of Theoretical Biology*, vol. 38, pp. 423–457, 1973.

- [122] B. Dasgupta, X. He, T. Jiang, M. Li, J. Tromp, and L. Zhang, “On computing the nearest neighbor interchange distance,” in *In: Proc. DIMACS Workshop on Discrete Problems with Medical Applications*. pp. 125–143, 1997.
- [123] K. C. II and D. Wood, “A note on some tree similarity measures,” *Information Processing Letters*, vol. 15, pp. 39–42, 1982.
- [124] W. H. E. Day, “Properties of the nearest neighbor interchange metric for trees of small size,” *Journal of Theoretical Biology*, vol. 101, pp. 275–288, 1983.
- [125] M. Krivanek, “Computing the nearest neighbor interchange metric for unlabeled binary trees is np-complete,” *Journal of Classification*, vol. 3, no. 1, pp. 55–60, March 1986.
- [126] F.-J. Lapointe, M. Wilkinson, and D. Bryant, “Matrix representations with parsimony or with distances: Two sides of the same coin?,” *Systematic Biology*, vol. 52, no. 6, pp. 865–868, 2003.
- [127] D. Pisani and M. Wilkinson, “Matrix representation with parsimony, taxonomic congruence, and total evidence,” *Systematic Biology*, vol. 51, no. 1, pp. 151–155, 2000.
- [128] S. Qiao and W. S.-Y. Wang, “A matrix representation of phylogenetic trees,” *Lecture Notes in Computer Science*, vol. 1276/1997, pp. 274–283, 1997.
- [129] M. A. Ragan, “Phylogenetic inference based on matrix representation of trees,” *Molecular Phylogenetics and Evolution*, vol. 1, pp. 53–58, 1992.
- [130] O. R. P. Bininda-Emonds, “The evolution of supertrees,” in *Trends Ecol Evol.*, vol. 19, pp. 315–322. 2004.

- [131] B. M. E. Moret, U. Roshan, T. Warnow, and T. L. Williams, “Performance of supertree methods on various dataset decompositions,” in *Phylogenetic Supertrees*, O. R. P. Bininda-Emonds, Ed., pp. 301–328. Kluwer Academic Publishers, 2004.
- [132] Z. Bai and J. W. Demmel, “Computing the generalized singular value decomposition,” *SIAM Journal on Scientific Computing*, vol. 14, pp. 1464–1486, 1993.
- [133] J. S. Hourigan and L. V. Mcindoo, (2005, December) “The singular value decomposition,” in *College of the Redwoods*. <http://online.redwoods.cc.ca.us/instruct/darnold/LAPROJ/Fall98/JodLynn/report2.pdf>, 1998.
- [134] D. Skillicorn, *Understanding Complex Datasets: Data Mining with Matrix Decompositions*, Boca Raton, FL: Chapman & Hall/CRC, 2007.

VITA

Seung Jin Sul received his B.S. degree in computer engineering from Dongguk University, Korea, in 1994 and his M.S. degree from Dongguk University, Korea, in 1996. During 2000-2004, he worked as a research manager at an IT company in Korea. Also he worked at Georgia Institute of Technology from 2003 to 2004 as a visiting scholar funded by the Korean government. He graduated with the Ph.D. in computer science and engineering from Texas A&M University in December 2009. His research interests lie primarily in bioinformatics and computational biology, especially in designing and analyzing algorithms for processing large-scale data. He is also interested in experimental and empirical performance studies, data-mining and visualization. He may be contacted at:

SEUNG JIN SUL

Department of Computer Science and Engineering

Texas A&M University

TAMU 3112

College Station, TX 77843-3112

U.S.A.

Phone: (979) 739-5033

Email: sulsj0270@gmail.com