# A Randomized Algorithm for Comparing Sets of Phylogenetic Trees

Seung-Jin Sul and Tiffani L. Williams
Department of Computer Science
Texas A&M University
E-mail: {sulsj,tlw}@cs.tamu.edu

Technical Report TR-2006-9-3

**Abstract**

Phylogenetic analysis often produce a large number of candidate evolutionary trees, each a hypothesis of the "true" tree. Post-processing techniques such as strict consensus trees are widely used to summarize the evolutionary relationships into a single tree. However, valuable information is lost during the summarization process. A more elementary step is produce estimates of the topological differences that exist among all pairs of trees. We design a new randomized algorithm, called *Hash-RF*, that computes the all-to-all Robinson-Foulds (RF) distance—the most common distance metric for comparing two phylogenetic trees. Our approach uses a hash table to organize the bipartitions of a tree, and a universal hashing function makes our algorithm randomized. We compare the performance of our Hash-RF algorithm to PAUP*'s implementation of computing the all-to-all RF distance matrix. Our experiments focus on the algorithmic performance of comparing sets of biological trees, where the size of each tree ranged from 500 to 2,000 taxa and the collection of trees varied from 200 to 1,000 trees. The results clearly show that our Hash-RF algorithm is up to 500 times faster than PAUP*'s approach. Thus, Hash-RF provides an efficient alternative to a single tree summary of a collection of trees and potentially gives researchers the ability to explore their data in new and interesting ways.

## 1 Introduction

The objective of a phylogenetic analysis is to infer the evolutionary tree for a given set of organisms (or taxa). 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. Under these criteria, trees that have better scores are believed to be better approximations of the truth. A typical phylogenetic search results in $t$ trees (i.e., hundreds to thousands of trees can be found), each representing a hypothesis of the "true" tree. Afterwards, post-processing techniques often use consensus methods to transform the rich output of a phylogenetic heuristic into a single summary tree [2]. Yet, much information is lost by summarizing the evolutionary relationships between the $t$ trees into a single consensus tree [7, 14].

Given a set of $t$ input trees, we design a randomized hash-based algorithm, called *Hash-RF*, that outputs a $t \times t$ matrix representing the topological distances between every pair of trees. The $t \times t$ distance matrix provides a more information-rich approach for summarizing $t$ trees. The most popular distance measure used to compare two trees is the Robinson-Foulds (RF) distance [12]. Under RF, the distance between two trees is based on the edges (or bipartitions) they share. It is

a widely used measure and can be computed in $O(n)$ time using Day's algorithm [5], where $n$ is the number of taxa. Very few algorithms have been designed specifically to compute the all-to-all RF distance. Notable exceptions include PAUP* [15], Phylip [6], and Split-Dist [9][1]. Pattengale and Moret provide an approximation algorithm [10], which provides with high probability a $(1 + \epsilon)$ approximation of the true RF distance matrix. Given that Pattengale and Moret's approach provides an approximation of the RF distance, we do not compare our approach to their algorithm. Furthermore, their implementation is only available in Java whereas the implementation of our approach is written in C++.

Our experimental results compare the performance on biological trees of our Hash-RF algorithm to the all-to-all RF algorithm embodied in PAUP*, a widely-used commercial application for inferring and interpreting phylogenetic trees. Here, $n$ ranges from 500 to 2,000 taxa and $t$ varies from 200 to 1,000 trees. The results clearly demonstrate that our approach outperforms PAUP*, where greater performance is achieved with increasing values of $n$ and $t$. On the largest dataset ($n = 2,000$ and $t = 1,000$), our algorithm is 500 times faster than PAUP*. We also compared our approach to Phylip and Split-Dist, but Phylip is tremendously slow even on our smallest dataset. Performance comparisons with Split-Dist followed the same trends as those shown with PAUP*. (Unfortunately, space limitations prevent us from showing these results). Thus, our Hash-RF algorithm provides an efficient alternative to consensus approaches for summarizing a large collection of trees.

# 2 Background

## 2.1 Phylogenetic trees

The leaves of an evolutionary tree are always labeled with the taxa, and permuting the labels on a tree with fixed topology generally produces a different evolutionary tree. Internal nodes—the hypothetical ancestors—are generally unlabeled. Phylogenies may be rooted or unrooted, and edges may be weighted or unweighted. Order is unimportant. For example, for a node in a rooted tree, swapping the left and the right child does not change the tree.

It is useful to represent evolutionary trees in terms of *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 $B_i$ associated with edge $e_i$. In Figure 1, $T_2$ has two bipartitions: $AB|CDE$ and $ABD|CE$. An evolutionary tree is uniquely and completely defined by its set of $O(n)$ bipartitions. For ease of computation, many algorithms represent each bipartition as a bit-string. At each internal node, those taxa whose subset includes a specified taxon are represented by the bit value '0'. For example, in Figure 1, those taxa that are in the subset of taxon A, are labeled '0'. All other taxa are labeled '1'. Thus, the bipartition, $AB|CDE$ is represented as the bit-string 00111 and $ABD|CE$ is represented as 00101.

## 2.2 Robinson-Foulds distance

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

[1] Actually, these approaches compute the symmetric distance between two trees. Dividing the symmetric distance by two easily converts it into the RF distance.

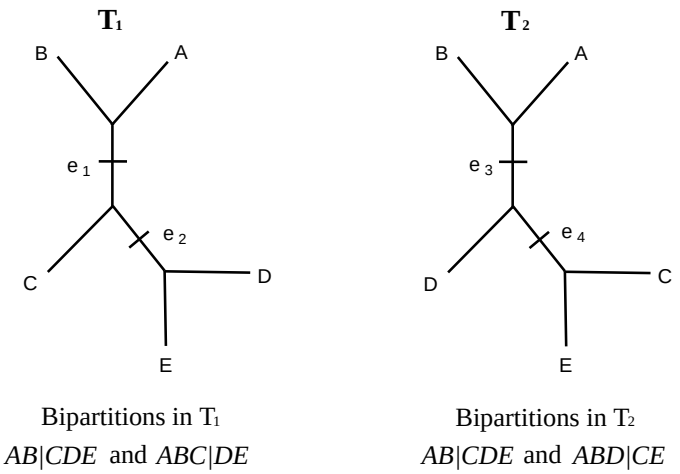



Figure 1: Two phylogenetic trees, $T_1$ and $T_2$, and their respective bipartitions. Internal edges are represented by the labels $e_1$ through $e_4$.

between trees $T_1$ and $T_2$ is defined as:

$$d_{RF}(T_1, T_2) = \frac{|\Sigma(T_1) - \Sigma(T_2)| + |\Sigma(T_2) - \Sigma(T_1)|}{2} \tag{1}$$

Figure 1 depicts how the RF distance between two trees is calculated. Trees $T_1$ and $T_2$ consist of five taxa, and each tree has two non-trivial bipartitions (or internal edges). In this example, the trees are binary and thus you can find $d_{RF}(T_1, T_2)$ is same with $d_{RF}(T_2, T_1)$. By the equation (1), the RF distance between the two trees in Figure 1 is 1. (Note that the numerator of equation (1) computes the symmetric distance between two trees.) An optimal $O(n)$ time complexity algorithm for computing RF distance between two trees given by Day. Day's algorithms first transforms a tree representation into a special cluster representation, which allows computing the RF distance in linear time.

### 2.3 Hashing

A set abstract data type (set ADT) is an abstract data type that maintains a set $S$ under the following three operations:

1. Insert($x$): Add the key $x$ to the set.
2. Delete($x$): Remove the key $x$ from the set.
3. Search($x$): Determine if the key $x$ is contained in the set, and if so, return $x$.

Hash tables are the most practical and widely used methods of implementing the set ADT and perform the three set ADT operations in $O(1)$ expected time.

The main idea behind all *hash table* implementations is to store a set of $n = |S|$ elements in an array (the hash table) of length $m \geq n$. Hence, we require a function 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 $h(x)$ is called the *hash value* of $x$. That is, the element $x$ gets stored at the array location $H[h(x)]$. Given two distinct elements $x_1$ and $x_2$, a *collision* occurs if $h(x_1) = h(x_2)$. 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.

# 3 The Hash-RF Algorithm

We were inspired by the work of Amenta et al. [1] to use a hash table as a mechanism for organizing tree bipartitions. Although their algorithm computes a majority consensus tree, we incorporate many of their ideas into our approach. Our Hash-RF algorithm consists of two major steps. The first step requires collecting all of the bipartitions from the evolutionary trees and hashing them into the hash table. Once all of the bipartitions are hashed into the table, the pairwise RF distances can be computed quite quickly. Algorithm 3 presents our hash-based approach. In the following subsections, we explain each of these major steps in detail.

## 3.1 Populating the hash table

Figure 2 provides an overview of the steps required in placing a tree's bipartitions into the hash table. As each input tree, $T_i$ is traversed in post-order, each of its bipartitions 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, $H$. $h_2$ is responsible for creating a unique identifier for each unique bipartition. For each bipartition, its associated hash table record contains its bipartition ID (BID) along with the index of the tree from where it originated.

For every hash function $h$, there exists bad sets $S$, where all distinct keys hash to the same address. To get around this difficulty, we need a collection of hash functions from which we can choose the one that works well for $S$. Even better would be a collection of hash functions such that, for any given $S$, most of the hash functions work well. Then, we could randomly pick one of the functions and have a good chance of it working well.

Similarly to Amenta et al. [1], we employ the use of universal hash functions, where $A = (a_1, ..., a_n)$ of is a list of random integers in $(0, ..., m_1 - 1)$ and $B = (b_1, ..., b_n)$ is a bipartition. Our $h_1$ and $h_2$ hash functions are defined as follows:

$$h_1(B) = \sum b_i a_i \ mod \ m_1 \tag{2}$$

$$h_2(B) = \sum b_i a_i \ mod \ m_2 \tag{3}$$

Using these universal hash functions, the probability that any two distinct bipartitions $B_i$ and $B_j$ collide (i.e., $h_1(B_i) = h_1(B_j)$) is $\frac{1}{m_1}$ [3, 4]. We call this Type II collision and collisions are described in more detail in the following subsection. If we choose $m_1 > tn$, the expected number of Type II collisions is $O(tn)$. A double collision (i.e., $h_1(B_i) = h_1(B_j)$ and $h_2(B_i) = h_2(B_j)$) occurs with probability $\frac{1}{m_1 m_2}$. Since the size of $m_2$ has no impact on the hash table size, $m_2$ can be made arbitrarily large to avoid double collisions with high probability. We provide more detail on how we detect double collisions (i.e., Type III collisions) below.

**Algorithm 1** The Hash-RF algorithm

**Require:** A set of $T_1, T_2, ..., T_t$ input trees

**for** $i = 1$ to $t$ **do**
  Traverse tree $T_i$ in post order
  **for all** bipartition $B_j \in T_i$ **do**
    Determine collision type at hash table $H[h_1(B_j)]$.
    **if** Type I collision **then**
      Increment count at KeyMap[$B_j$]
      Insert $h_2(B_j)$ and tree index $i$ into $H[h_1(B_j)]$
    **else if** Type III collision **then**
      Terminate and restart algorithm
    **else**
      Insert $h_1(B_j)$, $h_2(B_j)$ and $B_j$ into KeyMap[$B_j$]
      Increment count at KeyMap[$B_j$]
      Insert $h_2(B_j)$ and tree index $i$ into $H[h_1(B_j)]$
    **end if**
  **end for**
**end for**
**for all** KeyMap[$i$].count $\geq 2$ **do**
  Retrieve the linked list $l_i$ of nodes from $H[i]$
  **for all** node pairs $j, k \in l_i$ **do**
    **if** BID($j$) = BID($k$) **then**
      Increment $Sim[TID(\text{j})][TID(\text{k})]$
    **end if**
  **end for**
**end for**
**for all** $i, j \leq t$ **do**
  $RF[i][j] = \frac{((n-3)-Sim[i][j])+((n-3)-Sim[j][i])}{2}$
**end for**

### 3.2 Handling collisions

Given two bipartitions $B_i$ and $B_j$, there are three types of collisions in the algorithm. Table 1 provides a summary of the different collision types. The first collision type, which we call Type I, occurs as a result of identical bipartitions $B_i$ and $B_j$ appearing in two different trees. Hence, the record for each of these bipartitions at $h_1(B_i)$ will differ in the tree index part of their hash record. In a standard hash implementation, collisions occur between two different keys hashing to the same location. For our implementation, we must keep track of all of the trees that contain bipartition $B_i$ in order to compute the all-to-all RF distance. Thus, all of the trees that contain bipartition $B_i$ are chained together at location $h_1(B_i)$. Therefore, we consider this situation as a collision in our algorithm.

We use an additional data structure, called a *KeyMap*, which is a `map` container from the C++ Standard Template Library, for collision detection. The KeyMap table is used to store key/value pairs, where the keys are logically maintained in sorted order. Each unique bipartition from the set of $t$ trees is given an entry in KeyMap. Our KeyMap table contains four fields for each unique

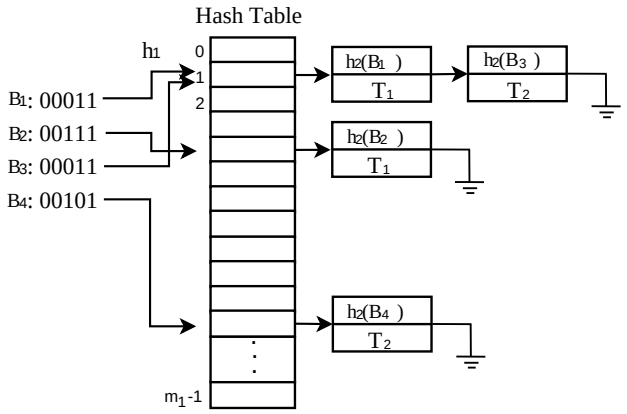



Figure 2: Populating the hash table with the bipartitions from trees $T_1$ and $T_2$, which are shown in Fig. 1. Bipartitions $B_1$ and $B_2$ define $T_1$, and $B_3$ and $B_4$ are from $T_2$. Each bipartition is fed to the hash functions $h_1$ and $h_2$. For each bipartition, its associated hash table record contains its bipartition ID (BID) along with the associated tree index (TID).

| Collision Type | $B_i = B_j$? | $h_1(B_i) = h_1(B_j)$? | $h_2(B_i) = h_2(B_j)$? |
|---|---|---|---|
| Type I | Yes | Yes | Yes |
| Type II | No | Yes | No |
| Type III | No | Yes | Yes |

Table 1: Collision types in the Hash-RF algorithm.

bipartition $B_i$: $h_1(B_i)$, $h_2(B_i)$, $B_i$, and the frequency of $B_i$. To detect if $B_i$ causes a Type I collision at $h_1(B_i)$, we search for $h_1(B_1)$ in the KeyMap table. If an entry is found, a collision has occurred. If the bipartition at this location is equal to $B_i$, we have a Type I collision. Otherwise, if no entry for KeyMap$[B_i]$ is found, $B_i$ is a new bipartition, and a new entry is created in the KeyMap table.

For a Type II collision, $h_1(B_i) = h_1(B_j)$ and $h_2(B_i) \neq h_2(B_j)$. Hash function $h_2$ is used to generate a bipartition identifier (BID), which attempts to distinguish $B_i$ from the other unique bipartitions. Let $B_j$ represent the bipartition field of the entry KeyMap$[h_1(B_i)]$. If $h_2(B_i) \neq h_(B_j)$, then a Type II collision has occurred. Hence, two different bipartitions hash to the same location in the hash table. (We note that this is the standard definition of collision in hash table implementations.) Otherwise, there is a double collision (or Type III collision), that is, the bipartitions $B_i$ and $B_j$ are different, but they have the same $h_1$ and $h_2$ values. In our algorithm, this is a critical collision, and the algorithm must be restarted with a different set of random integers for the set $A$.

### 3.3 Computing the pairwise RF distances

Once all the bipartitions are organized in the hash table, then the RF distance matrix can be calculated. To produce RF distance matrix, we first search the KeyMap table to find all the bipartition which occurred more than two times (i.e.,the bipartition exists in more than two trees). In turn, using the bipartition as a key, the location of the bipartition in the hash table can be easily acquired, and then we can have the list of tree indices in which the bipartition is found. At this time, we have to use the bipartition ID (BID) to select out the same bipartitions.

For example, if we have a list of trees $\{T_1, T_3, T_{11}\}$, it means that the bipartition $B_i$ occurred in trees $T_1$, $T_3$ and $T_{11}$ if all of their BIDs in the linked list is same. However, if the BID is only the same in $T_1, T_3$, then $B_i$ only occurred in trees $T_1$ and $T_3$ and not $T_{11}$. Once we have the list of tree indices (TIDs), the distance matrix is updated by using the tree index as the index in the RF matrix.

### 3.4 Analysis

In our algorithm, the hash table must be populated with $nt$ bipartitions. Hence, this stage of our algorithm requires $O(nt \log nt)$ time since each bipartition must be processed by the KeyMap table to detect the collision type. However, the distribution of the bipartitions in the hash table is responsible for the running time involved in calculating the RF distance matrix. The best case running time of $O(nt(\log nt + 1))$ arises when each hash location has one record, which occur when there are no bipartitions shared among the input trees. The worst case occurs when all of the $nt$ bipartitions hash to the same location $i$. Here, the size of the linked list at location $i$ will be $nt$, which requires $O((nt)^2)$ time to compute the RF distance matrix.

## 4 Our Collection of Biological Trees

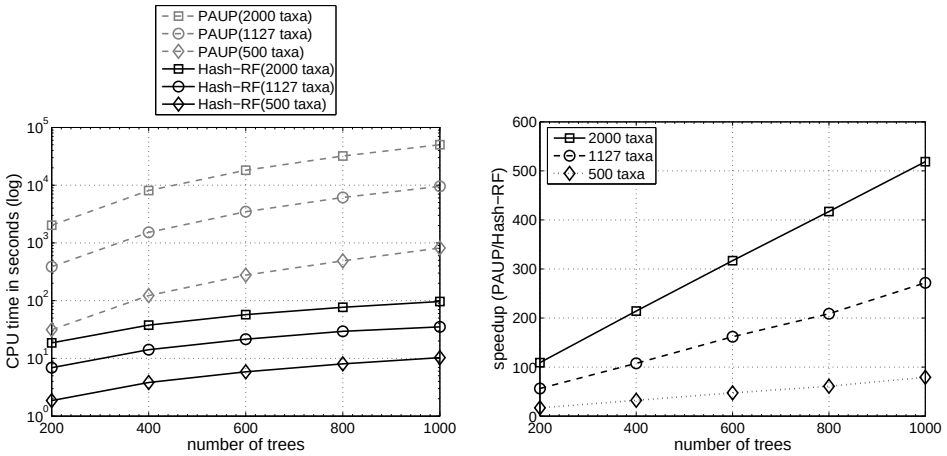
Since the performance of our algorithm is dependent upon the distribution of the $O(nt)$ bipartitions, our experiments consider the behavior of our Hash-RF algorithm between the best and worst running time bounds. Our experimental approach is to explore the performance of the algorithm on biological trees produced from a phylogenetic search. Since phylogenetic search techniques operate within a defined neighborhood of the search space, the resulting output tree tends to share many bipartitions among the $t$ trees.

The biological trees used in this study were obtained by running the *Recursive-Iterative DCM3 (Rec-I-DCM3)* algorithm [13], one of the best algorithms for obtaining maximum parsimony trees. We used the following molecular datasets to obtain phylogenetic trees from a Rec-I-DCM3 search: (1) a set of 500 aligned *rbc*L DNA sequences (1,398 sites) [11]; (2) a set of 1,127 aligned large subunit ribosomal RNA sequences (1,078 sites) obtained from the Ribosomal rRNA database [16]; and (3) a set of 2,000 aligned Eukaryotic sRNA sequences (1,251 sites) obtained from the Gutell Lab at the Institute for Cellular and Molecular Biology, The University of Texas at Austin. Thus, $n$ ranged from 500 to 2,000 taxa.

For each of the above datasets, a single run of the Rec-I-DCM3 algorithm produced 1,000 trees (i.e., the Rec-I-DCM3 was run for 1,000 iterations). From these 1,000 trees, we created five sets consisting of 200, 400, 600, 800, and 1,000 trees. Hence, $t$ ranged from 200 to 1,000 trees. Overall, we performed five runs of the Rec-I-DCM3 algorithm on each of the biomolecular datasets leading to 75 different collections of biological trees. Since there are five sets of trees for each pairing of $n$ and $t$, our experimental results show the average performance of the algorithms on the five tree collections for each pair of $n$ and $t$.

## 5 Experimental Results

We ran a series of experiments to study the performance of Hash-RF and PAUP* on the collection of biological trees described in the previous section. All experiments were run on an Intel Pentium




*(a) CPU time* *(b) Speedup*

Figure 3: Performance of Hash-RF and PAUP* on the collection of biological trees. (a) provides the running time required by the algorithms to compute the all-to-all RF distance matrix for various values of $n$ and $t$. (b) shows the speedup of the Hash-RF approach over PAUP*.

platform with 3.0GHz dual-core processors and a total of 2GB of memory.

Figure 3 compares the performance of the Hash-RF algorithm with PAUP* in terms of running time and speedup. The Hash-RF algorithm requires a minimum of 1.86 seconds on the smallest dataset ($n = 500$, $t = 200$) and a maximum of 96.72 seconds on the largest dataset ($n = 2,000$ and $t = 1,000$). For the same datasets, PAUP* requires 31.09 seconds and 13.94 hours, respectively. Hence, on our largest dataset, the Hash-RF algorithm is over 500 times faster than PAUP*'s all-to-all RF distance algorithm. Moreover, the results demonstrate that even greater speedups can be expected with larger collections of biological trees.

For $n = 2,000$, Table 2 provides information on the number of hash locations where $|l_i| = t$, which implies that bipartition $B_i$ is shared among the $t$ trees. A large number of locations with $|l_i| = t$ in the hash table will result in a slowdown in the performance of the Hash-RF algorithm since it will require $O(|l_i|^2)$ time to process the linked list at location $i$. The number of edges in an unrooted binary tree with $n$ taxa is $n - 3$, which also represents the maximum number of distinct bipartitions that can be shared across the $t$ trees. Shared bipartitions among the $t$ trees compose the strict consensus tree. Table 2 shows that the average number of bipartitions shared among the trees range from 43.77% to 45.62%, when $n = 2,000$. When $n = 500$, the resolution of the strict consensus tree ranges from 68.21% to 71.23% (not shown). So, even under diverse conditions of overlap among the bipartitions, the Hash-RF algorithm performs quite well in comparison to PAUP*.

# 6 Conclusions and Future Work

Phylogenetic search methods can produce large numbers of candidate trees as approximations to the "true" evolutionary tree. Such trees provide a powerful data-mining opportunity for examining the evolutionary relationships that exist among them. Post-processing methods such as strict consensus trees are the most common methods for providing a single tree that summarizes the

| $t$ | # hash locations where $\|l_i\| = t$ | strict consensus tree resolution (%) |
|---|---|---|
| 200 | 911 | 45.62 |
| 400 | 908 | 45.47 |
| 600 | 884 | 44.27 |
| 800 | 882 | 44.17 |
| 1,000 | 874 | 43.77 |

Table 2: For $n = 2{,}000$, the number of identical bipartitions shared between the $t$ trees and resulting resolution of the strict consensus tree.

information contained in the candidate trees. We advocate a more information-rich approach to analyzing the trees returned from a phylogenetic search. As a step in this direction, we present a fast randomized algorithm that calculates the Robinson-Foulds (RF) distance between each pair of evolutionary trees that in the best and worst case require $O(nt(\log nt + 1))$ and $O((nt)^2)$ running times, respectively.

Our experiments explore the behavior of our approach within these two boundaries. We compared the performance of our Hash-RF algorithm to PAUP*—a popular, commercially-available software package for inferring and interpreting phylogenetic trees—on large collections of biological trees. Our Hash-RF algorithm is up to 500 times faster than PAUP*'s approach. We also compared our approach to Phylip and Split-Dist, but Phylip is extremely slow and Split-Dist performed similarly to PAUP* (not shown). The experiments with biological trees share between 43.77% and 71.23% of their bipartitions among the $t$ trees. Given the diverse distributions of bipartition sharing among the biological trees, the results clearly demonstrate the performance gain achieved by using a hash-based approach for computing the RF distance between each pair of trees. Moreover, fast algorithms such as Hash-RF will enable users to perform interactive analyses of large tree collections in such applications as Mesquite [8].

Our work can be extended in many different directions. One immediate source of improvement would be a better mechanism for detecting collisions in our Hash-RF algorithm. Additional experiments will include randomly-generated trees (i.e, to control the degree of bipartition sharing among the $t$ trees) and larger tree collections. Using Day's $O(n)$ algorithm to compute the RF distance between two trees, it is possible theoretically to compute the all-pairs RF distance in $O(nt^2)$ time. We plan on implementing Day's algorithm and comparing its performance in practice to our Hash-RF approach. Finally, we are investigating the use of our hashing approach to design more efficient implementations of constructing consensus trees.

## References


[1] N. Amenta, F. Clarke, and K. S. John. A linear-time majority tree algorithm. In *Workshop on Algorithms in Bioinformatics*, volume 2168 of *Lecture Notes in Computer Science*, pages 216–227, 2003.

[2] D. Bryant. A classification of consensus methods for phylogenetics. In M. Janowitz, F. Lapointe, F. McMorris, B. Mirkin, and F. Roberts, editors, *Bioconsensus*, volume 61 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, pages 163–184. American Mathematical Society, DIMACS, 2003.

[3] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and Systems Sciences*, 18(2):143–154, 1979.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, Inc., 2001.

[5] W. H. E. Day. Optimal algorithms for comparing trees with labeled leaves. *Journal Of Classification*, 2:7–28, 1985.

[6] J. Felsenstein. Phylogenetic inference package (PHYLIP), version 3.2. *Cladistics*, 5:164–166, 89.

[7] D. M. Hillis, T. A. Heath, and K. S. John. Analysis and visualization of tree space. *Syst. Biol*, 54(3):471–482, 2004.

[8] W. P. Maddison and D. R. Maddison. Mesquite: a modular system for evolutionary analyses. Version 1.11, 2006. http://mesquiteproject.org.

[9] T. Mailund. SplitDist—calculating split-distances for sets of trees. Available from http://www.daimi.au.dk/ mailund/split-dist.html.

[10] N. D. Pattengale and B. M. E. Moret. A sublinear-time randomized approximation scheme for the robinson-foulds metric". In *Proc. 10th Int'l Conf. on Research in Comput. Molecular Biol. (RECOMB'06)*, volume 3909 of *Lecture Notes in Computer Science*, pages 221–230, 2006.

[11] K. Rice, M. Donoghue, and R. Olmstead. Analyzing large datasets: *rbcL* 500 revisited. *Systematic Biology*, 46(3):554–563, 1997.

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

[13] 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)*, pages 98–109. IEEE Press, 2004.

[14] C. Stockham, L. S. Wang, and T. Warnow. Statistically based postprocessing of phylogenetic analysis by clustering. In *Proceedings of 10th Int'l Conf. on Intelligent Systems for Molecular Biology (ISMB'02)*, pages 285–293, 2002.

[15] D. L. Swofford. PAUP*: Phylogenetic analysis using parsimony (and other methods), 2002. Sinauer Associates, Underland, Massachusetts, Version 4.0.

[16] J. Wuyts, Y. V. de Peer, T. Winkelmans, and R. D. Wachter. The European database on small subunit ribosomal RNA. *Nucleic Acids Research*, 30:183–185, 2002.