

# An Experimental Analysis of Consensus Tree Algorithms for Large-Scale Tree Collections

Seung-Jin Sul and Tiffani L. Williams

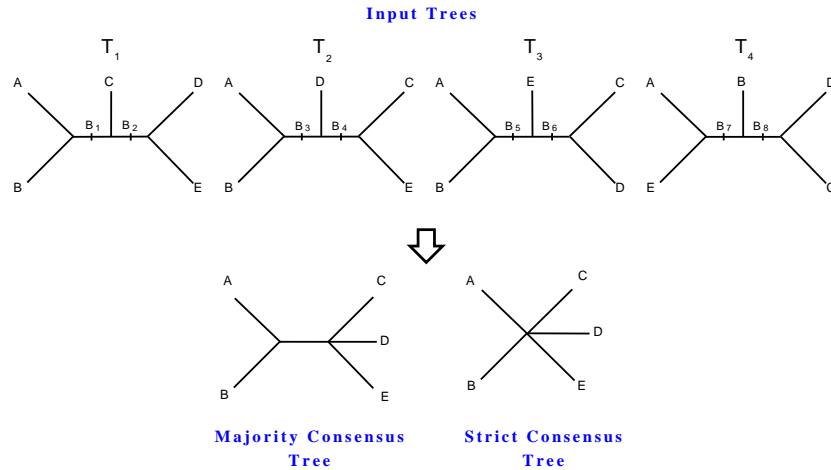
Department of Computer Science and Engineering  
Texas A&M University  
{sulsj,tlw}@cs.tamu.edu

**Abstract.** Consensus trees are a popular approach for summarizing the shared evolutionary relationships in a collection of trees. Many popular techniques such as Bayesian analyses produce results that can contain tens of thousands of trees to summarize. We develop a fast consensus algorithm called HashCS to construct large-scale consensus trees. We perform an extensive empirical study for comparing the performance of several consensus tree algorithms implemented in widely-used, phylogenetic software such as PAUP\* and MrBayes. Our collections of biological and artificial trees range from 128 to 16,384 trees on 128 to 1,024 taxa. Experimental results show that our HashCS approach is up to 100 times faster than MrBayes and up to 9 times faster than PAUP\*. Fast consensus algorithms such as HashCS can be used in a variety of ways, such as in real-time to detect whether a phylogenetic search has converged.

## 1 Introduction

Given a collection of organisms (or taxa), the objective of a phylogenetic analysis is to produce an evolutionary tree describing the genealogical relationships between the taxa. Phylogenetic methods (such as Bayesian analyses) 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. Currently, biologists use popular phylogenetic software packages such as PAUP\* [1] and MrBayes [2] to summarize their large tree collections into a single consensus tree (see Figure 1).

In this paper, we study whether current consensus tree implementations can accommodate the growing requirements of larger phylogenetic analyses, such as those necessary for building the *Tree of Life*, the grand challenge problem in phylogenetics. The *novelty* of our work consists of the following: (1) developing a fast algorithm to construct large-scale consensus trees and (2) performing an extensive, empirical investigation to analyze consensus tree performance. As trees increase in size (number of taxa) and potentially number (size of collection), fast approaches that can handle such sets of trees will be needed. Our experiments



**Fig. 1.** Overview of the consensus techniques on four trees of interest. Bipartitions (or internal edges) in a tree are labeled  $B_i$ , where  $i$  ranges from 1 to 8.

compare the performance of our HashCS algorithm to the consensus tree implementations in MrBayes [2], PAUP\* [1], and Phylip [3]. We also compare our work to Day’s algorithm [4], a theoretically, optimal approach for building strict consensus trees. We obtained a diverse collection of large *biological* and *artificial* trees to assess the performance of the consensus tree implementations. The biological trees were gathered from Bayesian analyses performed by life scientists on two molecular datasets consisting of 150 taxa (desert algae and green plants) [5] and 567 taxa (angiosperms) [6]. To complement our experiments using biological trees, we developed a generator to produce artificial trees to *predict* the performance of the consensus tree implementations on a more diverse collection of trees.

Our results clearly show that HashCS is the best algorithm for computing large-scale consensus trees. HashCS is up to 100 and 1.8 times faster than the modules in MrBayes and PAUP\* for computing consensus trees on our biological tree collections, respectively. Given that HashCS and PAUP\* are substantially faster than the other approaches, we look at their performance on a diverse collection of artificial trees with varying degrees of similarity among them. On these collections, the speedup of HashCS over PAUP\* ranges from 2 to 9. Although HashCS outperforms PAUP\*, the gap between the running times widens (or closes) in relation to the similarity of the evolutionary relationships (or bipartitions) among the trees.

Currently, consensus tree algorithms are exclusively used at the end of a phylogenetic analysis, which implies the search has converged in tree space resulting in the search returning highly similar trees. However, it is conceivable given the fast running times of HashCS (e.g., constructing the strict consensus of over 16,000 trees on 567 taxa requires around 40 seconds) that consensus trees could

be constructed in *real-time* repeatedly during a phylogenetic search. Under this scenario, the additional information provided by a consensus tree could be used to detect whether a search has converged. Consensus algorithms whose running times are minimally impacted by the diversity of trees that they may encounter is quite desirable. Thus, fast algorithms such as HashCS allow scientists to process their data in new and exciting ways.

## 2 Background

### 2.1 Evolutionary trees and their bipartitions

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. 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 1, tree  $T_1$  has two bipartitions:  $AB|CDE$  and  $ABC|DE$ . 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  $n - 3$  non-trivial (or internal) bipartitions.

For each tree in the collection of input trees, we find all of its bipartitions (internal edges) by performing a postorder traversal. In order to process the bipartitions, we need some way to store them in the computer's internal memory. 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  $T_1$  in Figure 1. This bipartition is represented as 11000. Taxa on the same side of a bipartition as taxon  $A$  receive a '1'.

### 2.2 Consensus trees

Consensus trees summarize the information of a collection of trees into a single output tree. We consider the most popular consensus approaches: strict and majority 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. 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 to compute 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. The resolution rate of a tree  $T$  varies between 0% (a star) and 100% (completely resolved binary tree).

### 3 Comparison with Previous Work

A number of different techniques have been developed to summarize tree collections by building consensus trees. Traditional approaches include Day’s algorithm [4], MrBayes, PAUP\*, and Phylip, and our results demonstrate that HashCS outperforms them in practice.

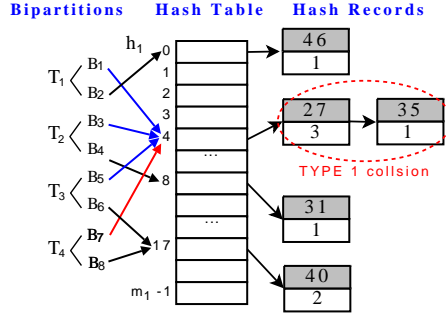
Our HashCS algorithm is motivated by the work of Amenta, Clark, and St. John [7]. They develop an  $O(nt)$  optimal algorithm for computing the majority consensus tree, where  $n$  is the number of taxa and  $t$  is the number of trees. Unfortunately, there appears to be no publicly available implementation that we could obtain of their algorithm. Moreover, in their paper, Amenta et al. do not provide any empirical evidence of their algorithm’s running time on biological or artificial tree collections.

HashCS is a fast implementation of Amenta et al.’s algorithm. However, there are several key differences between the approaches. First, HashCS requires a single traversal of the collection of  $t$  trees to construct a consensus tree. Amenta et al.’s algorithm requires two traversals of the  $t$  trees, where the second traversal of the trees is an involved procedure to construct the majority tree from the nodes in the hash table. Secondly, unlike Amenta et al.’s approach, HashCS does not insert all bipartitions into the hash table. Finally, Amenta et al. explicitly detect double collisions (which can result in an incorrect consensus tree) in the hash table, but our experiments show that this step is unnecessary. Thus, HashCS does not explicitly check for double collisions.

The Texas Analysis of Symbolic Phylogenetic Information (TASPI) system is a recently proposed technique to compute consensus trees [8], [9]. One of the novelties of TASPI is it incorporates a new format for compactly storing and retrieving phylogenetic trees. Experimental results on several collections of maximum parsimony trees show that the TASPI system outperforms PAUP\* [1] and TNT [10] in constructing consensus trees. An implementation of the TASPI system does not appear to be available to use for experimental comparison in this paper. However, since it was demonstrated clearly that TASPI and PAUP\* are much faster than TNT’s consensus algorithm, we do not explore the performance of TNT in this paper. A comparison of Phylip and MrBayes consensus methods was not included in the experiments with TASPI. Hence, we include those techniques in our study.

### 4 Our HashCS Algorithm

Building consensus trees using a hash table consists of two major steps. First, the hash table is populated with the bipartitions from the collection of trees (see Figure 2). Each tree  $T_i$ ’s bipartitions are fed through two hashing functions,  $h_1$  and  $h_2$  to determine where it should reside in the hash table. The bipartition is stored in the hash table and a frequency counter associated with that bipartition is updated. Once all bipartitions are inserted into the hash table, a consensus tree is constructed based on the values of the bipartition frequency counters.



**Fig. 2.** Overview of the HashCS algorithm. Bipartitions are from Figure 1. That is,  $B_1$  and  $B_2$  define tree  $T_1$ ,  $B_3$  and  $B_4$  are from tree  $T_2$ , 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.

#### 4.1 Step 1: Populating the hash table

*Hashing functions  $h_1$  and  $h_2$ .* Similarly to Amenta et al. [7], we define two universal hashing functions. Our first hash function is defined as  $h_1(B) = \sum b_i r_i \bmod m_1$  and the second one is defined similarly,  $h_2(B) = \sum b_i s_i \bmod m_2$ .  $m_1$  represents the number of entries (or locations) in the hash table.  $m_2$  represent the largest bipartition ID (BID) that we can be given to a bipartition. That is, instead of storing the  $n$ -bitstring, a shortened version of it (represented by the BID) will be stored in the hash table instead. HashCS 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  $B$ .

*Implicit bipartitions.* For faster performance, we avoid sending the  $n$ -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 (instead of a  $n$ -bitstring) 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

$$h_1(B) = \left( \sum_{B_{left}} b_i r_i \bmod m_1 \right) + \left( \sum_{B_{right}} b_i r_i \bmod m_1 \right). \quad (1)$$

We can use Equation 1 to compute implicit bipartitions, which replace the need for bitstrings in our hashing functions. Computing  $h_2$  works similarly.

*Inserting bipartitions into the hash table.* HashCS uses two different policies for inserting bipartitions into the hash table. 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. If these bipartitions are found in the remaining trees, their frequency counters in the hash table are incremented. For the last tree, the  $n$ -bitstring representation for each bipartition is computed along with the implicit representation. The  $n$ -bit representation of a bipartition is stored into an array  $A$  if its frequency count is  $t$ . Let  $x$  represent the number of  $n$ -bitstrings in array  $A$ , where  $x$  is the number of bipartitions that will compose the strict consensus tree. The array  $A$  of  $n$ -bitstring representations will be used to build the consensus tree in Step 2 of the 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. 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  $A$ . Similarly to the strict consensus case, during Step 2 of the algorithm the array  $A$  of majority  $n$ -bitstrings will be used to build the majority tree.

*Collision types and their 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 two types to consider. *Type 1* collisions result from two different bipartitions  $B_i$  and  $B_j$  (i.e.,  $B_i \neq B_j$ ) residing in the same location in the hash table. That is,  $h_1(B_i) = h_1(B_j)$ . *Type 2* (or double) collisions are serious and require a restart of the algorithm. Otherwise, the resulting output will 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 HashCS 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. As a result, HashCS has a theoretical error rate of  $O(\frac{1}{c})$ . In practice, however, the error rate of HashCS 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%.

## 4.2 Step 2: Constructing the consensus tree

Once all the bipartitions in the tree collection have been processed, we can build the consensus tree. In our approach, the consensus tree is initially a star tree of  $n$  taxa. Bipartitions from the array  $A$ , which contains  $x$   $n$ -bitstrings created in Step 1, are added to refine the consensus tree based on the number of 1’s in their  $n$ -bitstring representation.<sup>1</sup> The more 1’s in the  $n$ -bitstring representation,

<sup>1</sup> The number of 0’s could have been used as well.

the more taxa that are grouped together by this bipartition. A star tree is an  $n$ -bitstring representation consisting 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 the most taxa appears first in the sorted list. The bipartition that groups together the fewest taxa appears last in the sorted list of '1' bit counts.

For each bipartition  $B$  in the sorted list, a new internal node is created to further refine the taxa in the consensus tree created so far. To do this, the bipartition  $B$  is scanned to put the taxa into two groups. Taxa in the  $n$ -bitstring representation of bipartition  $B$  with '0' bits compose one group and those with '1' bits compose the other group. The taxa in the current consensus tree 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.

### 4.3 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. Using our assumption that  $t \gg n$ , then 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  $O(nt)$ , which is optimal since there are  $O(nt)$  total bipartitions in the input trees that must be processed by any consensus algorithm.

## 5 Our Collection of Evolutionary Trees

### 5.1 Biological trees

The large biological tree collections used in this study were obtained from two recent Bayesian analyses.

- 20,000 trees obtained from a Bayesian analysis of an alignment of 150 taxa (23 desert taxa and 127 others from freshwater, marine, and oil habitats) with 1,651 aligned sites [5]. The majority and strict consensus resolution rates for these 20,000 trees are 87% and 34%, respectively.
- 33,306 trees obtained from an analysis of a three-gene, 567 taxa (560 angiosperms, seven outgroups) dataset with 4,621 aligned characters [6]. The majority and strict consensus resolution rates for these 33,306 trees are 93% and 51%, respectively.

In our experiments, for each number of taxa,  $n$ , we created different tree set sizes,  $t$ , to test the scalability of the algorithms. Dividing the original tree collection into smaller tree sets simulates using higher burn-in rates (i.e., higher burn-in rates means the consensus tree is composed of fewer trees) as well as shorter Bayesian runs (fewer generations produce fewer trees). Hence, the entire collection of trees is divided into smaller sets, where  $t$  is 128, 256, 512,  $\dots$ , 16384 trees. Thus, for each  $(n, t)$  pair,  $t$  trees with  $n$  taxa were randomly sampled without replacement from the appropriate tree collection. For each  $(n, t)$  pair, we repeated the above sampling process five times. Our experimental results show the average running time performance for each  $(n, t)$  pair.

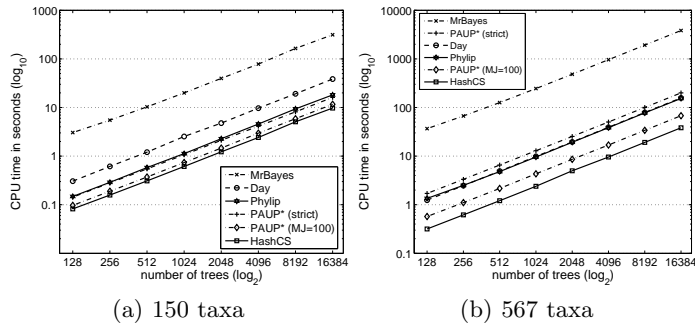
## 5.2 Artificial trees

We use artificial tree collections to *predict* how the performance of the algorithms would scale across a wider range of trees than are available with our biological tree collections. To generate our collection of  $t$  trees, each consisting of  $n$  taxa, we first generate a random, Yule model trees on  $n$  taxa called  $T_{100\%}$ . Each of these trees are binary and have  $(n - 3)$  bipartitions. Next, we remove  $r\%$  of the bipartitions from  $T_{100\%}$  to create a  $r\%$  strict or majority consensus tree,  $C_{r\%}$ . If  $C_{r\%}$  represents a majority tree, then we assign a random number in the interval  $(50, 100]$  to each bipartition in  $C_{r\%}$ . This value represents the percentage of the  $t$  trees that contain that bipartition. For example, consider the 50% resolution majority tree in Figure 1. The bipartition  $AB|CDE$  appears in 75% of the input trees. Hence, three ( $T_1, T_2$ , and  $T_3$ ) of the four trees are selected randomly to contain the bipartition  $AB|CDE$ . If  $C_{r\%}$  represents a strict tree, then each consensus bipartition appears 100% of the time.

Once all of the bipartitions from the consensus tree  $C_{r\%}$  have been distributed, each of the  $t$  trees for the collection is constructed. For each tree  $T_i$ , where  $1 \leq i \leq t$ , we construct tree  $T_i$  with the consensus bipartitions that have been distributed to it. If tree  $T_i$  is a multifurcating tree (i.e., its resolution rate is less than 100%), then it is randomly made into a binary tree. That is, the internal nodes in tree  $T_i$  that are not binary are randomly resolved so that they become binary. These randomly resolved bipartitions (non-consensus bipartitions) are then distributed to the remaining  $\lfloor p \rfloor$  trees, where  $1 < p \leq 0.50t$  and  $i < p \leq t$ , if  $C_{r\%}$  is a majority tree. If  $C_{r\%}$  is a strict tree, then  $1 < p < t$  and  $i < p \leq t$ . We distribute the non-consensus bipartitions to the remaining trees in order to increase the amount of sharing among the non-consensus bipartitions in the tree collection. The above process is repeated until all  $t$  trees are constructed.

## 5.3 Experimental validation, implementations and platform

All consensus implementations produce the same majority and strict consensus trees for the tree collections. Although PAUP\*, Phylip, and MrBayes have methods for inferring evolutionary trees, all experiments were done using only the consensus tree modules in these programs. No phylogenetic searches were conducted in our experiments.



**Fig. 3.** Running time of the strict consensus tree algorithms on our collection of biological trees. The scale of the y-axis is different for all the plots.

All experiments were run on an Intel Pentium platform with a 3.0GHz processor and a total of 2GB of memory. We used the Linux operating system (Red Hat 2.5.22.14-17.fc6). HashCS, and Day were written in C++ and compiled with gcc 4.1.2 with the `-O3` compiler option. PAUP\* is commercially-available software and we used version 4.0b10 in our experiments. We also compared our approach to the consensus tree approaches in Phylip (ver. 3.65) and MrBayes (ver. 3.1.2), which are freely available.

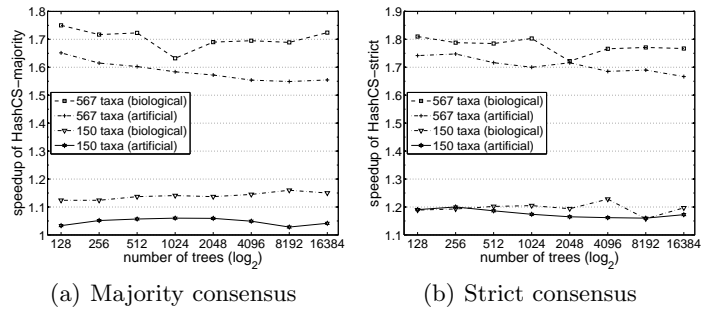
## 6 Results

### 6.1 Biological tree collections

First, we consider the running time of the consensus tree implementations for computing the strict consensus tree. Figure 3 shows the results. Overall, HashCS is the fastest implementation 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). HashCS is 1.8 times faster than PAUP\* on the largest tree collection. In our plots, 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. Other researchers have observed this behavior as well [8], [9]. In any case, PAUP\*(MJ=100) is the second faster performer behind HashCS. The running times for constructing majority trees are similar and are not shown because of space limitations.

### 6.2 Artificial tree collections

The previous figures clearly demonstrate that MrBayes, Phylip, and Day's algorithm are not competitive as HashCS and PAUP\*. So, we don't consider them

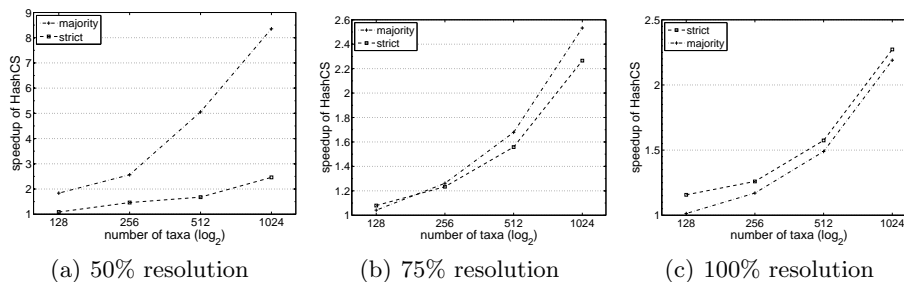


**Fig. 4.** Speedup comparison of HashCS over PAUP\* on artificial and biological trees consisting of 150 and 567 taxa. The scale of the y-axis is different for all the plots.

any further in this paper. For our artificial tree collections, the number of taxa,  $n$ , varies from 128 to 1,024. The number of trees,  $t$  is 16,386. Our results with biological trees show that the speedup of HashCS is not impacted by  $t$ . Instead, it is impacted by the number of taxa,  $n$ , so we fix  $t$  to be 16,386 trees.

To test the predictive capability of our model, Figure 4 shows the speedup of HashCS over PAUP\* on biological and artificial trees of 150 and 567 taxa. The resolution rates of the consensus trees used to create the artificial trees are the same rates that were given in Section 5.1 for the biological trees. For example, in Figure 4(a), the majority resolution rate for both the biological and artificial trees for 567 taxa is 93%. The speedup obtained on our artificial trees is slightly lower than the speedup obtained with our biological trees. By using artificial trees for our experiments, we are not overestimating the performance of the algorithms. Thus, Figure 4 provides evidence that our artificial tree collections are valid for predicting the performance of HashCS and PAUP\* on larger and more diverse collections of trees.

Figure 5 shows the resulting speedup of HashCS over PAUP\*. Since PAUP\*'s running time performance varies as a function of the distribution of the input trees, the speedup varies as well. PAUP\* performs the worst when constructing a 0% or 25% consensus tree resulting in HashCS being up to 300 and 60 times faster than PAUP\* for  $n = 1,024$ , respectively (not shown). At 50% consensus resolution, HashCS is up to 9 times faster than PAUP\*. 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\*. Furthermore, the plot shows that as the resolution decreases and the number of taxa increases, PAUP\*'s majority algorithm is much slower than its strict consensus algorithm. In HashCS, constructing strict and majority trees require about the same time.



**Fig. 5.** Speedup of the consensus tree algorithms on 16,384 artificial trees of varying taxa sizes and consensus tree resolution rates. The scale of the y-axis is different for all the plots.

## 7 Conclusions

In this paper, we performed an extensive empirical analysis of five different consensus tree implementations (HashCS, PAUP\*, Day, Phylip, and MrBayes) on a diverse collection of phylogenetic trees. From life scientists, we obtained two collections consisting of 20,000 and 33,306 biological trees, which were the result of Bayesian analyses on 150 and 567 taxa molecular datasets, respectively. To predict how the consensus tree implementations would perform on even larger trees with greater numbers of taxa, we developed a model to generate collections of large-scale artificial trees.

Our experimental results show that our HashCS implementation is the fastest approach for building large-scale consensus trees. The performance of HashCS in comparison to other consensus tree implementations is not impacted by the number of trees in a collection. Instead, the biggest speedup gains occur with increasing number of taxa. Hence, as trees increase in size, even more performance gains can be expected from HashCS. Furthermore, our artificial tree collections predict that HashCS will not be impacted by amount of bipartition sharing among the trees in the collection.

Fast algorithms improve the speed of exploration to facilitate new discoveries. Hence, fast algorithms can be used in all aspects of reconstructing a phylogenetic tree, even if that level is currently not a bottleneck. For the tree collections studied here, constructing the resulting consensus trees requires very little time—especially when compared to the phylogenetic analyses that produced our biological trees. Depending on the needs and patience of a phylogenetic researcher, a typical phylogenetic search could take a few hours to several months. However, the time required to prepare the phylogenetic data (e.g., collecting organisms in the field, extracting molecular data in the lab, performing a multiple sequence alignment on the data) can take years, which dominates the time required for a typical phylogenetic search.

Finally, our results indicate that HashCS could be used for more than post-processing trees. For example, it is difficult to determine whether a phyloge-

netic heuristic such as MrBayes 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. Finally, faster algorithms make it feasible to study the increasing size of tree collections from large-scale phylogenetic analyses (such as those geared toward building the *Tree of Life*) in a reasonable amount of time.

## 8 Acknowledgments

This work was funded by the National Science Foundation under grants DEB-0629849 and IIS-0713618. The authors would like to thank Nick Pattengale, Eric Gottlieb, and Bernard Moret for providing the code for Day's algorithm. Matthew Gitzendanner, Paul Lewis, and David Soltis for providing us with the Bayesian tree collections used in this paper.

## References

1. Swofford, D.L.: PAUP\*: Phylogenetic analysis using parsimony (and other methods) (2002) Sinauer Associates, Underland, Massachusetts, Version 4.0.
2. Ronquist, F., Huelsenbeck, J.P.: Mrbayes 3: Bayesian phylogenetic inference under mixed models. *Bioinformatics* **19**(12) (August 2003) 1572–1574
3. Felsenstein, J.: Phylogenetic inference package (PHYLIP), version 3.2. *Cladistics* **5** (1989) 164–166
4. Day, W.H.E.: Optimal algorithms for comparing trees with labeled leaves. *Journal Of Classification* **2** (1985) 7–28
5. Lewis, L.A., Lewis, P.O.: Unearthing the molecular phylodiversity of desert soil green algae (chlorophyta). *Syst. Bio.* **54**(6) (2005) 936–947
6. Soltis, D.E., Gitzendanner, M.A., Soltis, P.S.: A 567-taxon data set for angiosperms: The challenges posed by bayesian analyses of large data sets. *Int. J. Plant Sci.* **168**(2) (2007) 137–157
7. Amenta, N., Clarke, F., John, K.S.: A linear-time majority tree algorithm. In: Workshop on Algorithms in Bioinformatics. Volume 2168 of Lecture Notes in Computer Science. (2003) 216–227
8. Boyer, R.S., Hunt Jr., W.A., Nelesen, S.: A compressed format for collections of phylogenetic trees and improved consensus performance. In: Proc. 5th Int'l Workshop Algorithms in Bioinformatics (WABI'05). Volume 3692 of Lecture Notes in Computer Science., Springer-Verlag (2005) 353–364
9. Hunt Jr., W.A., Nelesen, S.M.: Phylogenetic trees in ACL2. In: Proc. 6th Int'l Conf. on ACL2 Theorem Prover and its Applications (ACL2'06), New York, NY, USA, ACM (2006) 99–102
10. Goloboff, P.: Analyzing large data sets in reasonable times: solutions for composite optima. *Cladistics* **15** (1999) 415–428