A Linear-time Majority Tree Algorithm

Nina Amenta[1], Frederick Clarke[2], and Katherine St. John[2,3]

[1] Computer Science Department
University of California, 2063 Engineering II
One Sheilds Ave, Davis, CA 95616.
amenta@cs.ucdavis.edu
[2] Dept. of Mathematics & Computer Science
Lehman College– City University of New York
Bronx, NY 12581
fclarke72@aol.com, stjohn@lehman.cuny.edu
[3] Department of Computer Science
CUNY Graduate Center, New York, NY 10016

Abstract. We give a randomized linear-time algorithm for computing the majority rule consensus tree. The majority rule tree is widely used for summarizing a set of phylogenetic trees, which is usually a post-processing step in constructing a phylogeny. We are implementing the algorithm as part of an interactive visualization system for exploring distributions of trees, where speed is a serious concern for real-time interaction. The linear running time is achieved by using succinct representation of the subtrees and efficient methods for the final tree reconstruction.

1 Introduction

Making sense of large quantities of data is a fundamental challenge in computational biology in general and phylogenetics in particular. With the recent explosion in the amount of genomic data available, and exponential increases in computing power, biologists are now able to consider larger scale problems in phylogeny: that is, the construction of evolutionary trees on hundreds or thousands of taxa, and ultimately of the entire "Tree of Life" which would include millions of taxa. One difficulty with this program is that most programs used for phylogeny reconstruction [8, 9, 17] are based upon heuristics for NP-hard optimization problems, and instead of producing a single optimal tree they generally output hundreds or thousands of likely candidates for the optimal tree. The usual way this large volume of data is summarized is with a consensus tree.

A consensus tree for a set of input trees is a single tree which includes features on which all or most of the input trees agree. There are several kinds of consensus trees. The simplest is the *strict consensus tree*, which includes only nodes that appear in all of the input trees. A node here is identified by the set of taxa in the subtree rooted at the node; the roots of two subtrees with different topologies, but on the same subset of taxa, are considered the same node. For some sets of input trees, the strict consensus tree works well, but for others, it produces

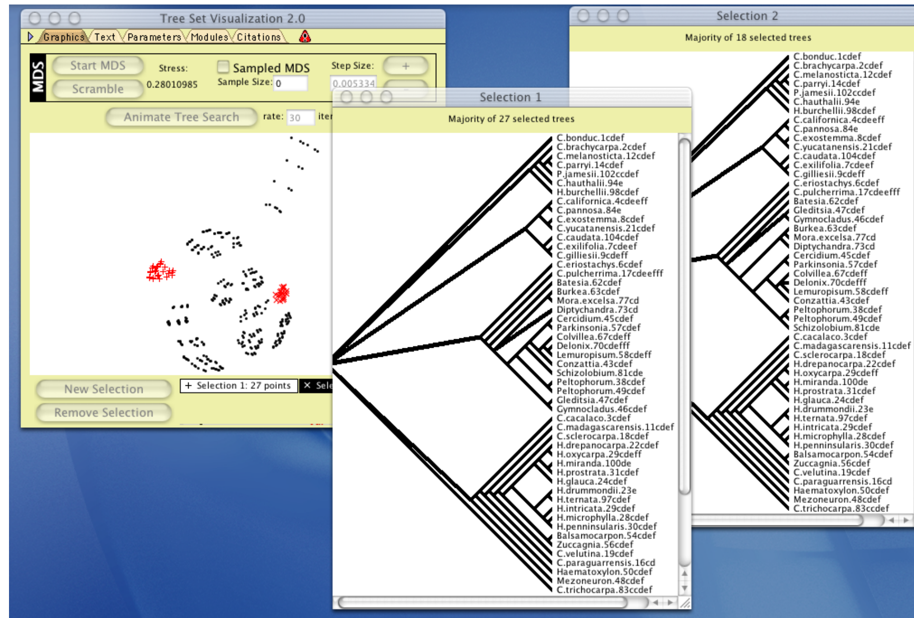


Fig. 1. The tree visualization module in Mesquite. The window on the left shows a projection of the distribution of trees. The user interactively selects subsets of trees with the mouse, and, in response, the consensus tree of the subset is computed on-the-fly and displayed in the window on the right. Two selected subsets and their majority trees are shown.

a tree with very few interior (non-terminal) nodes, since if a node is missing in even one input tree it is not in the strict consensus. The *majority rule consensus tree* includes all nodes that appear in a majority of input trees, rather than all of them. The majority rule tree is interesting for a much broader range of inputs than the strict consensus tree. Other kinds of consensus tree, such as Adams consensus, are also used (see [3], §6.2, for an excellent overview of consensus methods). The maximum agreement subtree, which includes a maximal subset of taxa for which the subtrees induced by the input trees agree, gives meaningful results in some cases in which the majority rule tree does not, but the best algorithm has an $O(tn^3+n^d)$ running time [7] (where d is the maximum outdegree of the trees), which is not as practical for large trees as the majority rule tree. Much recent work has been done on the related question of combining trees on overlapping, but not identical, sets of taxa ([2, 13–16]).

In this paper, we present a randomized algorithm to compute the majority rule consensus tree, where the expected running time is linear both in the number t of trees *and* in the number n of taxa. Earlier algorithms were quadratic in n,

which will be problematic for larger phylogenies. Our $O(tn)$ expected running time is optimal, since just reading a set of t trees on n taxa requires $\Omega(tn)$ time. The expectation in the running time is over random choices made during the course of the algorithm, independent of the input; thus, on any input, the running time is linear with high probability.

We were motivated to find an efficient algorithm for the majority rule tree, because we wanted to compute it on-the-fly in an interactive visualization application [1]. The goal of the visualization system is to give the user a more sensitive description of the distribution of a set of trees than can be presented with a single consensus tree. Figure 1 shows a screen shot. The window on the left shows a representation of the distribution of trees, where each point corresponds to a tree. The user interactively selects subsets of trees and, in response, the consensus tree of the subset is computed on-the-fly and displayed. This package is built as a module within Mesquite [10], a framework for phylogenetic computation by Wayne and David Maddison. See Section 4 for more details.

Our original version of the visualization system computed only strict consensus trees. We found in our prototype implementation that a simple $O(tn^2)$ algorithm for the strict consensus tree was unacceptably slow for real-time interaction, and we implemented instead the $O(tn)$ strict consensus algorithm of Day [6]. This inspired our search for a linear-time majority tree algorithm.

Having an algorithm which is efficient in t is essential, and most earlier algorithms focus on this. Large sets of trees arise given any kind of input data on the taxa (e.g. gene sequence, gene order, character) and whatever optimization criterion is used to select the “best” tree. The heuristic searches used for maximizing parsimony often return large sets of trees with equal parsimony scores. Maximum likelihood estimation, also computationally hard, generally produces trees with unique scores. While technically one of these is the optimal tree, there are many others for which the likelihood is only negligibly sub-optimal. So, the output of the computation is again more accurately represented by a consensus tree.

Handling larger sets of taxa is also becoming increasingly important. Maximum parsimony and maximum likelihood have been used on sets of about 500 taxa, while researchers are exploring other methods, including genetic algorithms and super-tree methods, for constructing very large phylogenies, with the ultimate goal of estimating the entire “Tree of Life”. Our visualization system is designed to support both kinds of projects. It is also important for the visualization application to have an algorithm which is efficient when $n > t$, so that when a user selects a small subset of trees on many taxa some efficiency can be realized.

1.1 Notation

Let $\mathcal{S}$ represent a set of taxa, with $|\mathcal{S}| = n$. Let $\mathcal{T} = \{T_1, T_2, \ldots, T_t\}$ be the input set of trees, each with n leaves labeled by $\mathcal{S}$, with $|\mathcal{T}| = t$.

Without loss of generality, we assume the input trees are rooted at the branch connecting a distinguished taxon s_0, known as the *outgroup*, to the rest of the

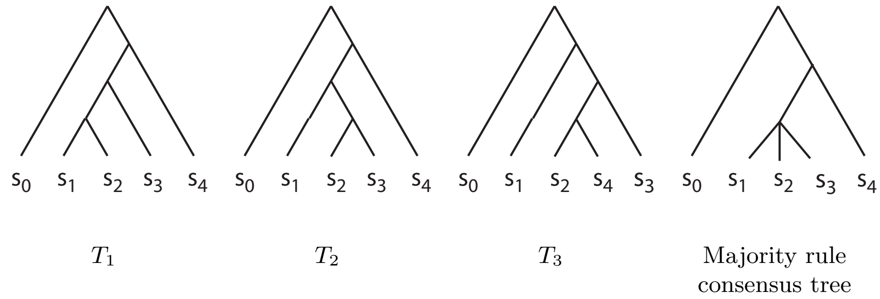


Fig. 2. Three input trees, rooted at the branch connecting s_0, and their majority tree (for a $> 1/2$ majority). The input trees need not be binary.

tree. If $\mathcal{T}$ is given as unrooted trees, or trees rooted arbitrarily, we choose an arbitrary taxon as s_0 and use it to root (or re-root) the trees.

Consider a node i in an input tree T_j. Removing the branch from i towards the root divides T_j into the subtree below i and the remainder of the tree (including s_0). The induced *bipartition* of the taxa set into two subsets identifies the combinatorial type of node i. We can represent the bipartition by the subset of taxa which does *not* include s_0; that is, by the taxa at the leaves of the subtree rooted at i. If B is the bipartition, this set is $S(B)$. We will says that the *cardinality* of B, and of i, is the cardinality of $S(B)$. For example, in Figure 2, $s_1 s_2 \mid s_0 s_3 s_4 s_5$ is a bipartition of tree T_1 and $S(s_1 s_2 \mid s_0 s_3 s_4 s_5) = \{s_1 s_2\}$. The cardinality of this bipartition is 2.

The *majority rule tree*, or M_l tree, includes nodes for exactly those bipartitions which occur in more than half of the input trees, or more generally in more than some fraction l of the input trees. Margush and McMorris [11] showed that this set of bipartitions does indeed constitute a tree for any $1/2 < l \leq 1$. McMorris, Meronk and Neumann [12] called this family of trees the M_l trees (e.g. the M_1 tree is the strict consensus tree); we shall call them all generically majority rule trees, regardless of the size of the majority.

See Figure 2 for a simple example. While this example shows binary trees, the algorithm also works for input trees with polytomies (internal nodes of degree greater than three).

1.2 Prior Work

Our algorithm follows the same intuitive scheme as most previous algorithms. In the first stage, we read through the input trees and count the occurrences of each bipartition, storing the counts in a table. Then, in the second stage, we create nodes for the bipartitions that occur in a majority of input trees - the *majority nodes* - and "hook them together" into a tree.

An algorithm along these lines is implemented in PHYLIP [8] by Felsenstein *et al.*. The overall running time as implemented seems to be $O((n/w)(tn+x \lg x+n^2))$ where x is the number of bipartitions found ($O(tn)$ in the worst case, but often $O(n)$), and w is the number of bits in a machine word. The bipartition B of each of tn input nodes is represented as a *bit-string*: a string of n bits, one per taxon, with a one for every taxon in $S(B)$ set and a zero for every taxon not in $S(B)$. This requires $\lceil n/w \rceil$ machine words per node, and accounts for (n/w) factor in the bound. The first term is for counting the bipartitions. The $x \lg x$ term is for sorting the bipartitions by the number of times each appears; it could be eliminated if the code was intended only to compute majority trees. The n^2 term is the running time for the subroutine for hooking together the majority nodes. For each majority node, every other majority node is tested to see if it is its parent, each in $\lceil n/w \rceil$ time.

For the strict consensus tree, Day's deterministic algorithm uses a clever $O((\lg x)/w)$ representation for bipartitions. If we assume that the size of a machine word is $O(\lg x)$, so that for instance we can compare two bipartitions in $O(1)$ time, then we say that Day's algorithm achieves an optimal $O(tn)$ running time. Day's algorithm does not seem to generalize to other M_l trees, however. Wareham, in his undergraduate thesis at the Memorial University of Newfoundland with Day [18], developed an $O(n^2 + t^2 n)$ algorithm, which only uses $O(n)$ space. It uses Day's data structure to test each bipartition encountered separately against all of the other input trees. Majority trees are also computed by PAUP [17], using an unknown (to us) algorithm.

Our algorithm follows the same general scheme, but we introduce a new representation for each bipartition of size $O((\lg x)/w) \approx O(1)$, giving an $O(tn)$ algorithm for the first counting step, and we also give an $O(tn)$ algorithm for hooking together the majority nodes.

2 Majority Rule Tree Algorithm

Our algorithm has two main stages: scanning the trees to find the majority bipartitions (details in Section 2.1) and then constructing the majority rule tree from these bipartitions (details in Section 2.2). It ends by checking the output tree for errors due to (very unlikely) bad random choices. Figure 3 contains pseudo-code for the algorithm.

2.1 Finding Majority Bipartitions

In the first stage of the algorithm, we traverse each input tree in post-order, determining each bipartition as we complete the traversal of its subtree. We count the number of times each bipartition occurs, storing the counts in a table. With the record containing the count, we also store the cardinality of the bipartition, which turns out to be needed as well.

A first thought might be to use the bit-string representation of a bipartition as an address into the table of counts, but this would be very space-inefficient:

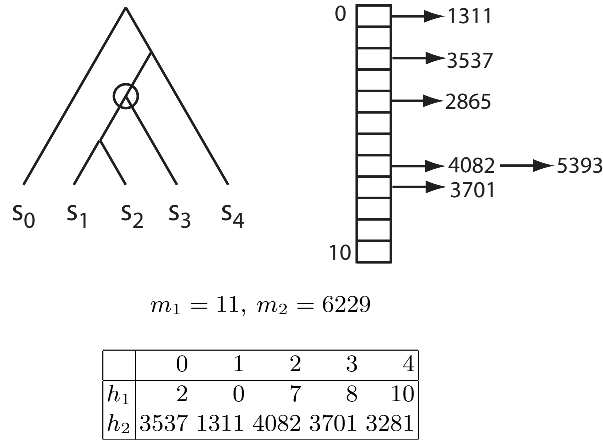
Input:	A set of t trees, $\mathcal{T} = \{T_1, T_2, \ldots, T_t\}$.
Output:	The majority tree, M_l, of $\mathcal{T}$
Algorithm:	
Start-up:	Pick prime numbers m_1 and m_2 and random integers for the hash functions h_1 and h_2.
1.	For each tree $T \in \mathcal{T}$,
2.	Traverse each node, x, in *post order*.
3.	Compute hashes $h_1(x)$ and $h_2(x)$
4.	If no double collision, insert into hash table (details below).
5.	If double collision, restart algorithm from beginning.
6.	For each tree $T \in \mathcal{T}$,
7.	Let c point to the root of T
8.	Traverse each node, x, in *pre order*.
9.	If x is a majority node,
10.	If not existing in M_l, add it, set its parent to c.
11.	Else x already exists in M_l, update its parent (details below).
12.	In recursive calls, pass x as c.
13.	Check correctness of tree M_l.

Fig. 3. The pseudo-code for the majority tree algorithm

there are at most $O(tn)$ distinct bipartitions, but 2^n possible bit-strings. A better idea, used in our algorithm and in `PHYLIP`, is to store the counts in a hash-table.

Hash Tables: Our algorithm depends on the details of the hash-table implementation, so we briefly include some background material and terminology (or see, for example, [5], Chapter 11). We use a function, the *hash function*, to compute the address in the table at which to store the data. The address is called the *hash code*, and we say an element *hashes* to its address. In our case, the hash function takes the 2^n possible bipartitions into $O(tn)$ hash table addresses; but since at most tn of the possible bipartitions actually occur in the set of input trees, on average we put only a constant number of bipartitions at each address. More than one bipartition hashing to the same table address is called a *collision*. To handle collisions, we use a standard strategy called *chaining*: instead of storing a count at each table address, we store a linked list of counts, one for each bipartition which has hashed to that address. When a new bipartition hashes to the address, we add a new count to the linked list. See Figure 4.

Universal Hash Functions: As the hash function, we use a *universal hash function* ([5], §11.3.3, or [4]), which we call h_1 (the reader expecting an h_2 later will not be disappointed). As our program starts up, it picks a prime number m_1 which will be the size of the table, and a list $a = (a_1, \ldots, a_n)$ of random integers in $(0, \ldots, m_1 - 1)$. We can select m from a selection of stored primes of different sizes; only the a need to be random. Let $B = (b_1, \ldots, b_n)$ be the bit-string representation of a bipartition. The universal hash function is defined



$m_1 = 11,\ m_2 = 6229$

	0	1	2	3	4
h_1	2	0	7	8	10
h_2	3537	1311	4082	3701	3281

Fig. 4. Storing nodes in the hash table: Assume we have the five leaf tree on the left, T_1, and two universal hash functions (and associated prime numbers) given by the table below. The IDs stored in the hash table for the beginning of a post-order traversal of T_1 are shown; the circled node was the last one processed. First s_0 was processed, storing $h_2(s_0) = 3537$ at $h_1(s_0) = 2$. Similarly s_1 and s_2 were processed, and then their parent, storing $h_2 = 1311 + 4082 \bmod 6229$ into $h_1 = 0 + 7 \bmod 11$, and so on.

as

$$h_1(B) = \sum_{i=1}^{n} b_i a_i \bmod m_1$$

Notice that $h_1(B)$ is always a number in $0, \ldots, m_1 - 1$.

Using this universal hash function, the probability that any two bipartitions B_1 and B_2 collide (that is, that $h_1(B_1) = h_1(B_2)$) is $1/m_1$ [4], so that if we choose $m_1 > tn$ the expected number of collisions is $O(tn)$.

Collisions: To detect and handle these collisions, we use a second universal hash function h_2 to produce an ID for each bipartition. We represent a bipartition in the hash table by a record which includes the ID as well as the running count of the number of times the bipartition occurs. To increment the count of a bipartition B, we go to the table address $h_1(B)$ and search the linked list for the record with ID $h_2(B)$. If no such record exists, we create one and give it a count of one.

It is possible that a *double collision* will occur; that is, that for two bipartitions B_1, B_2, we have $h_1(B_1) = h_1(B_2)$ and also $h_2(B_1) = h_2(B_2)$. Since these two events are independent, the probability that B_1 and B_2 have a double collision is $1/(m_1 m_2)$. Notice that although we have to choose $m_1 \approx tn$ to avoid wasting space in the table, we can choose m_2 to be very large so as to minimize the probability of double collisions. If we choose $m_2 > ctn$, for any constant

c, the probability that no double collisions occur, and hence that the algorithm succeeds, is at least $1-O(1/c)$, and the size of the representation for a bipartition remains $O(\lg tn + \lg c)$.

Nonetheless, we need to detect the unlikely event that a double collision occurs. If so, we abort the computation and start over with new random choices for the parameters a of the hash functions.

If B_1 and B_2 represent bipartitions with different numbers of taxa, we can detect the double collision right away, since we can check to make sure that both the IDs and the cardinality in the record match before incrementing a count. Similarly if B_1 and B_2 are bipartitions corresponding to leaves we can detect the double collision immediately by checking that the two taxa match before incrementing the count. The final remaining case of detecting a double collision for two bipartitions B_1, B_2, both with cardinality $k > 1$, will be done later by a final check of output tree against the hash table.

Implicit Bipartitions: To achieve an $O(tn)$ overall running time, we need to avoid computing the $O(n/w)$-size bit-string representations for each of the $O(tn)$ bipartitions. Instead, we directly compute the hash codes recursively at each node, without explicitly producing the bit-strings.

Fact 1 *Consider a node with a bipartition represented by bit-string B. Let the two children of this node have bipartitions with bit-strings B_L and B_R. Then*

$$h_1(B) = h_1(B_L) + h_1(B_R) \ \mathit{mod}\ m_1$$

This is true because B_l and B_r represent disjoint sets of taxa, so that

$$h_1(B) = \left(\sum_{B_L} b_i a_i \bmod m_1\right) + \left(\sum_{B_R} b_i a_i \bmod m_1\right)$$

where a_i is the prime number assigned to i by the universal hash function. A similar statement of course holds for h_2, and when B has more than two children.

We can use this fact to compute the hash code recursively during the post-order traversal. We store the hash codes in the tree nodes as we compute them. At a leaf, we just look up the hash code in the array a; for the leaf containing taxon i, the hash code is a_i. For an internal node, we will have already computed the hash codes of its children $h_1(B_L)$ and $h_1(B_R)$, so we compute $h_1(B)$ in constant time using Fact 1. The reader may wish to go over the example of the computation in Figure 4.

We compute the cardinality of the bipartition in constant time at each node similarly using recursion.

2.2 Constructing the Majority Tree

Once we have all the counts in the table we are ready to compute the majority rule consensus tree. The counts let us identify which are the majority bipartitions

that appear in more than lt trees. But since the bipartitions are represented only implicitly, by their hash functions, hooking them up correctly to form the majority rule tree is not totally straightforward. We use three more facts.

Fact 2 *The parent of a majority bipartition B in the majority rule tree is the majority bipartition B' of smallest cardinality such that B is a subset of B'.*

Fact 3 *If majority bipartition B' is an ancestor of majority bipartition B in an input tree T_j, then B' is an ancestor of B in the majority rule tree.*

Fact 4 *For any majority bipartition B and its parent B' in the majority rule tree, B and B' both appear in some tree T_j in the input set. In T_j, B' is an ancestor of B, although it may not be B's parent.*

Fact 4 is true because both B and B' appear in more than $l \geq t/2$ trees, so they have to appear in some tree together, by the pigeon-hole principle.

We do a pre-order traversal of each of the input trees in turn. As we traverse each tree, we keep a pointer to c, the last node corresponding to a majority bipartition which is an ancestor of the current node in the traversal. As we start the traversal of a tree T, we can initialize c to be the root, which always corresponds to a majority bipartition. Let C be the bipartition corresponding to c.

At a node i, we use the stored hash codes to find the record for the bipartition B in the hash table. If B is not a majority node, we ignore it. If B is a majority node and a node for B does not yet exist in the output tree, we create a new node for the output tree and with its parent pointer pointing to C. If, on the other hand, a node in the outut tree does exist for B, we look at its current parent P in the output tree. If the cardinality of P (stored in the hash table record for P) is greater than the cardinality of C, we switch the parent pointer of the node for B to point to the node for C. When we are done, each node B in the output tree, interior or leaf, points to the node of smallest cardinality that was an ancestor in any one of the input trees. Assuming there was no double collision, Facts 2, 3, and 4 imply that the output tree is the correct majority rule consensus tree.

2.3 Final check

After constructing the majority rule tree, we check it against the hash table in order to detect any occurrence of the final remaining case of a double collision, when two bipartitions B_1, B_2 of the same cardinality $k > 1$ have the same value for both h_1 and h_2. Recall that, if B_1, B_2 are singletons or have different cardinalities, double collisions would already have been detected when putting the data into the hash table.

To check the tree, we do a post-order traversal of the completed majority rule tree, recursively computing the cardinality of the bipartition at each node, and checking that these cardinalities match those in the corresponding records

in the hash table. If we find a discrepancy, this indicates a double collision and we discard the majority rule tree and run the algorithm again with new random choices for the parameters a of the hash functions.

Claim. Any remaining double collisions are detected by checking the cardinalities.

Proof: Let us consider the smallest k for which a double collision occurs, and let B_1, B_2 be two of the bipartitions of cardinality k which collide. Since the double collision was undetected in the first stage, there is some record B in the hash table, with cardinality k, representing both B_1 and B_2. Whenever B_1 or B_2 was encountered during the first stage of the algorithm, the count for B was incremented.

In the second reconstruction stage, a node in the output tree is created for B as soon as either B_1 or B_2 is encountered in the traversal of some input tree. Consider a bipartition C which is a child of B_1 (equivalently B_2) in the correct majority rule tree. At the end of stage two, there will be a node for C in the output tree. Its parent pointer will be pointing to B, since in some input tree B_1 (resp. B_2) will be an ancestor of C, with cardinality k, and no majority node of cardinality less than k will be an ancestor of C in any input tree. Thus the nodes for all majority bipartitions which would be children of either B_1 or B_2 in the correct majority rule tree end up as children of B in the incorrect output tree. Notice that since B_1, B_2 have the same cardinality, one cannot be the ancestor of the other; so the two sets $S(B_1), S(B_2)$ are disjoint. Therefore the cardinality of B in the output tree will be $2k$, while the cardinality stored in the hash table for B will be k.

2.4 Analysis summary

The majority rule consensus tree algorithm runs in $O(tn)$ time. It does two traversals of the input set, and every time it visits a node it does a constant number of operations, each of which requires constant expected time (again, assuming that $w = O(\lg x)$). The final check of the majority tree takes $O(n)$ time.

The probability that any double collision occurs is $1/c$, where c is the constant such that $m_2 > ctn$. Thus the probability that the algorithm succeeds on its first try is $1 - 1/c$, the probability that r attempts will be required decreases exponentially with r, and the expected number of attempts is less than two.

3 Weighted trees

The majority rule tree has an interesting characterization as the median of the set of input trees, which is useful for extending the definition to weighted trees. When a bipartition is not present in an input tree we consider it to have weight zero, while when a bipartition is present we associate with it the positive weight of the edge determining the bipartition in the rooted tree. Now consider the

medial weight of each bipartition over all input trees, including those that do not contain the bipartition. A bipartition which is *not* contained in a majority of the input trees has median weight zero. A bipartition in a majority of input trees has some positive median weight.

Note that it is simple, although space-consuming, to compute this median weight for each majority bipartition in $O(nt)$ time. In the second pass through the set of input trees, we store the weights for each majority edge in a linked list, as they are encountered. Since there are $O(n)$ majority bipartitions and t trees the number of weights stored is $O(nt)$. The median weight for each bipartition can then be computed as the final tree is output, using a textbook randomized median algorithm which runs in $O(t)$ time per edge ([5], §9.2).

4 Implementation

Our majority rule consensus tree algorithm is implemented as part of our tree-set visualization system, which in turn is implemented within Mesquite [10]. Mesquite is a framework for phylogenetic analysis written by Wayne and David Maddison, available for download at their Web site [10]. It is designed to be portable and extensible: it is written in Java and runs on a variety of operating systems (Linux, MacIntosh OS 9 and X, and Windows).

Mesquite is organized into cooperating of modules. Our visualization system has been implemented in such a module, TreeSetVisualization, the first published version of which can be downloaded from our webpage [1]. The module was introduced last summer at Evolution 2002 meeting and has since been downloaded by hundreds of researchers. In the communications we get from users, majority trees are frequently requested. The TreeSetVisualization module includes tools for visualizing, clustering, and analyzing large sets of trees.

The majority tree implementation will be part of the next version of the module. Figure 1 shows our current prototype. We expect to release the new version this summer, including the majority tree code and other new features.

5 Acknowledgments

This project was supported by NSF-ITR 0121651/0121682 and computational support by NSF-MRI 0215942. The first author was also supported by an Alfred P. Sloan Foundation Research Fellowship. We thank Jeff Klingner for the tree set visualization module and Wayne and David Maddison for Mesquite, and for encouraging us to consider the majority tree. The second and third authors would like to thank the Department of Computer Sciences and the Center for Computational Biology and Bioinformatics at University of Texas, and the Computer Science Department at the University of California, Davis for hosting them for several visits during 2002 and 2003.

References

1. Nina Amenta and Jeff Klingner. Case study: Visualizing sets of evolutionary trees. In *8th IEEE Symposium on Information Visualization (InfoVis 2002)*, pages 71–74, 2002. Software available at `www.cs.utexas.edu/users/phylo/`.
2. B.R. Baum. Combining trees as a way of combining data sets for phylogenetic inference, and the desirability of combining gene trees. *Taxon*, 41:3–10, 1992.
3. David Bryant. *Hunting for trees, building trees and comparing trees: theory and method in phylogenetic analysis.* PhD thesis, Dept. of Mathematics, University of Canterbury, 1997.
4. J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and Systems Sciences*, 18(2):143–154, 1979.
5. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms.* MIT Press, Cambridge, MA, second edition, 2001.
6. William H.E. Day. Optimal algorithms for comparing trees with labeled leaves. *J. Classification*, 2(1):7–28, 1985.
7. Martin Farach, Teresa M. Przytycka, and Mikkel Thorup. On the agreement of many trees. *Information Processing Letters*, 55(6):297–301, 1995.
8. J. Felsenstein. Phylip (phylogeny inference package) version 3.6, 2002. Distributed by the author. Department of Genetics, University of Washington, Seattle. The consensus tree code is in `consense.c` and is co-authored by Hisashi Horino, Akiko Fuseki, Sean Lamont and Andrew Keeffe.
9. John P. Huelsenbeck and Fredrik Ronquist. Mrbayes: Bayesian inference of phylogeny, 2001.
10. W.P. Maddison and D.R. Maddison. Mesquite: a modular system for evolutionary analysis. version 0.992, 2002. Available from `http://mesquiteproject.org`.
11. T. Margush and F.R. McMorris. Consensus n-trees. *Bulletin of Mathematical Biology*, 43:239–244, 1981.
12. F.R. McMorris, D.B. Meronk, and D.A. Neumann. A view of some consensus methods for trees. In *Numerical Taxonomy: Proceedings of the NATO Advanced Study Institute on Numerical Taxonomy*. Springer-Verlag, 1983.
13. R.D.M. Page. Modified mincut supertrees. *Lecture Notes in Computer Science (WABI 2002)*, 2452:537–551, 2002.
14. M.A. Ragan. Phylogenetic inference based on matrix representation of trees. *Molecular Phylogenetics and Evolution*, 1:53–58, 1992.
15. M.J. Sanderson, A. Purvis, and C. Henze. Phylogenetic supertrees: assembling the trees of life. *Trends in Ecology and Evolution*, 13:105–109, 1998.
16. Charles Semple and Mike Steel. A supertree method for rooted trees. *Discrete Applied Mathematics*, 105(1-3):147–158, 2000.
17. D.L. Swofford. *PAUP*. Phylogenetic Analysis Using Parsimony (*and Other Methods). Version 4*. Sinauer Associates, Sunderland, Massachusetts, 2002.
18. H. Todd Wareham. An efficient algorithm for computing M_l consensus trees, 1985. BS honors thesis, CS, Memorial University Newfoundland.