A Novel Approach for Compressing Phylogenetic Trees.

CONFERENCE PAPER · JANUARY 2010

DOI: 10.1007/978-3-642-13078-6_13 · Source: DBLP

3 AUTHORS, INCLUDING:

Suzanne J Matthews
United States Military Academy
18 PUBLICATIONS **48** CITATIONS
SEE PROFILE

Seung-Jin Sul
Lawrence Berkeley National Laboratory
18 PUBLICATIONS **63** CITATIONS
SEE PROFILE

A Novel Approach for Compressing Phylogenetic Trees

Suzanne J. Matthews, Seung-Jin Sul, and Tiffani L. Williams

Texas A&M University, College Station TX 77843, USA
{sjm,sulsj,tlw}@cse.tamu.edu

Abstract. Phylogenetic trees are tree structures that depict relationships between organisms. Popular analysis techniques often produce large collections of candidate trees, which are expensive to store. We introduce TreeZip, a novel algorithm to compress phylogenetic trees based on their shared evolutionary relationships. We evaluate TreeZip's performance on fourteen tree collections ranging from 2,505 trees on 328 taxa to 150,000 trees on 525 taxa corresponding to 0.6 MB to 434 MB in storage. Our results show that TreeZip is very effective, typically compressing a tree file to less than 2% of its original size. When coupled with standard compression methods such as `7zip`, TreeZip can compress a file to less than 1% of its original size. Our results strongly suggest that TreeZip is very effective at compressing phylogenetic trees, which allows for easier exchange of data with colleagues around the world.

1 Introduction

Phylogenetics is concerned with reconstructing the evolutionary history (or family tree) for a set of organisms. An understanding of evolutionary mechanisms and relationships is at the heart of modern pharmaceutical research for drug discovery. It is also helping researchers understand (and defend against) rapidly mutating viruses such as HIV, and is the basis of genetically enhanced organisms. Typically, the evolutionary history for these organisms (or taxa) is depicted as a binary tree, where the taxa are the leaves of the tree and the edges represent the evolutionary relationships between the taxa (see Figure 1). To reconstruct a phylogenetic tree, the most popular techniques (such as MrBayes [5]) often return tens to hundreds of thousands of trees that represent equally-plausible hypotheses for how the taxa evolved from a common ancestor. We develop a new compression algorithm called *TreeZip* that reduces the requirements over standard compression algorithms for storing large collections of evolutionary trees. Furthermore, our TreeZip algorithm allows large phylogenetic tree collections to be shared easily with colleagues around the world.

The set of all edges (or *bipartitions*) from an evolutionary tree uniquely defines that tree. However, a tree's non-trivial bipartitions (or internal edges) are of most interest. To simplify our discussion, we use the term bipartitions to refer to a tree's set of non-trivial bipartitions. In Figure 1, each tree's bipartitions are represented by vertical lines. A bipartition represents a split on an internal

M. Borodovsky et al. (Eds.): ISBRA 2010, LNBI 6053, pp. 113–124, 2010.

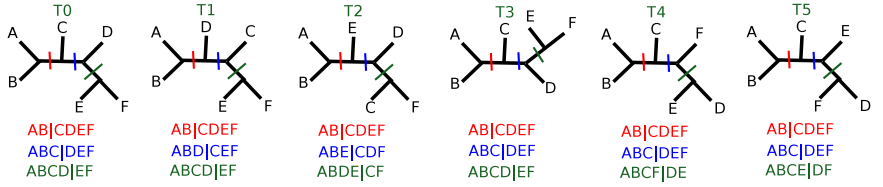


Fig. 1. A collection of six evolutionary trees on six taxa labeled A to F. For each tree, its set of bipartitions are listed.

T_0 = (((A,B),C),(D,(E,F)));
T_1 = (((A,B),D),(C,(E,F)));
T_2 = (((A,B),E),(D,(C,F)));
T_3 = (((A,B),C),((E,F),D));
T_4 = (((A,B),C),(F,(E,D)));
T_5 = (((A,B),C),(E,(F,D)));

(a) Newick strings

T_0 = (D,((A,B),C),(E,F));
T_1 = (C,(D,(B,A)),(E,F));
T_2 = (D,((B,A),E),(F,C));
T_3 = (D,(C,(B,A)),(E,F));
T_4 = (F,((A,B),C),(E,D));
T_5 = (E,((B,A),C),(D,F));

(b) equivalent Newick strings

Fig. 2. Newick representations for the phylogenetic trees shown in Figure 1. Two different, but equivalent, Newick representations are given for each tree.

edge of the evolutionary tree that separates the taxa into two groups. A set of bipartitions uniquely defines an evolutionary tree. For example, tree T_0's bipartitions are $AB|CDEF$, $ABC|DEF$, and $ABCD|EF$ where the symbol '|' acts as a separator. Trees T_0 and T_3 are identical trees since they contain the same set of bipartitions. For a binary tree, the number of bipartitions is $n-3$, where n is the number of taxa.

The Newick format [4] is the most widely used format to store a phylogenetic tree in a file. In this format, the topology of the evolutionary tree is represented using a notation based on balanced parentheses. Consider tree T_0 in Figure 1. A Newick representation of the topology of this tree is `(((A,B),C),(D,(E,F)));`, where '`;`' symbolizes the end of the Newick string. Matching pairs of parentheses symbolize internal nodes in the evolutionary tree. The Newick representation of a tree is not unique. For example, another valid Newick string for tree T_0 is `(D,((A,B),C),(E,F));`. Figure 2(a) shows the Newick tree file for the trees in Figure 1, where the Newick representation is based on the lexicographical ordering of the taxa names. Given that trees can have multiple, valid Newick strings, Figure 2(b) shows a different Newick file, where the taxa names are ordered randomly for each tree. For a given tree T_i on n taxa, there are $O(2^{n-1})$ possible Newick strings to represent it.

Our contributions. In this paper, we introduce TreeZip, a new lossless algorithm for compressing large collections of phylogenetic trees. TreeZip requires $O(nt)$ running time for both its compression and decompression phases, where n is the number of taxa and t is the number of trees in the collection of interest.

Given that many of the bipartitions in a collection of phylogenetic trees are shared, the *novelty* of our TreeZip approach is storing such relationships only once in the compressed representation. TreeZip compresses a Newick file based on the *semantic* representation (i.e., tree bipartitions). General-purpose data compression techniques (e.g., `gzip`, `bzip,` and `7zip`) do not know what the data (Newick file) represents beyond the ASCII string representations. Hence, there is great potential for obtaining good compression by utilizing the semantic information in a Newick file describing large collections of evolutionary trees.

TreeZip leverages two phylogenetic tree algorithms, HashCS (constructs consensus trees) [11] and HashRF (computes a topological $t \times t$ distance matrix) [10], which use a hash table to organize the bipartitions from a collection of trees efficiently. We demonstrate the performance of our TreeZip algorithm in comparison to standard compression approaches (i.e., `gzip`, `bzip,` and `7zip`) on 14 different large-scale tree collections. Our largest (smallest) tree collection consists of 150,000 (2,505) trees requiring 434 MB (0.6 MB) of storage space. Overall, our results show that the compressed TreeZip (`.trz`) file occupies from 0.2% to 2% of its original size, which outperforms `gzip` and `bzip` compression algorithms. When TreeZip is coupled with a standard compression algorithm, even greater compression is attained. For the datasets studied here, the best compression occurs when TreeZip compression is followed by `7zip`. Hence, TreeZip is a great alternative for biologists who want to recycle the trees generated from their experiments.

Related work. To the best of our knowledge, the Texas Analysis of Symbolic Phylogenetic Information (TASPI) [2] [3] is the only described approached for compressing evolutionary trees. It is written in the ACL2 formal logic language, but we were unable to find an available implementation of the TASPI algorithm for direct comparison to our approach on all of our tree collections. However, we were able to obtain the collection of trees that TASPI used to evaluate their approach [2]. Section 3.1 compares the compression ratios of TreeZip to TASPI on those set of trees, but without a TASPI implementation, we were unable to compare running times.

Our TreeZip algorithm compliments and extends the work done with TASPI in several ways. While compression storage results are given, the main focus of TASPI is building a single consensus (or summary) tree from a compressed representation of the collection of trees. While TreeZip can build consensus trees (not shown here), our main focus is on compressing large collections of evolutionary trees efficiently. Since a Newick string does not give a unique representation for a phylogenetic tree (there are $O(2^{n-1})$ possible Newick strings), the designers of TASPI note that their algorithm is affected by the ordering of the taxa in the Newick string. TreeZip, on the other hand, has been designed to not be impacted by different Newick strings representing the same tree. Finally, TASPI does not explicitly state if it has a decompression routine in order to rebuild the original Newick tree file containing the t trees. TreeZip has such a routine.

2 Our TreeZip Algorithm

Our TreeZip algorithm compresses and decompresses phylogenetic trees based on their shared evolutionary relationships. Under compression, the input to the algorithm is a Newick file and the output is a TreeZip (or a `.trz`) file. The input to TreeZip's decompression phase is a `.trz` file and the output is a Newick file.

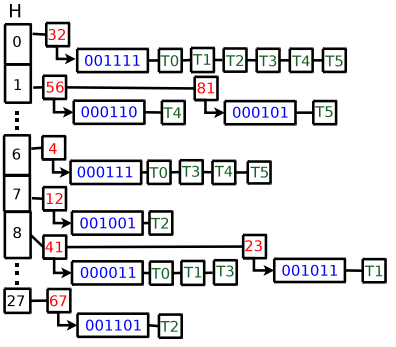
2.1 Compression: Converting the Newick File to a `.trz` File

Building a hash table from the Newick file. In the Newick input file, each string i, which represents tree T_i, is read and stored in a tree data structure. During the depth-first traversal of input tree T_i, each of its bipartitions is fed through two universal hash functions, h_1 and h_2 [1]. Both of these functions require as input a n-bit bitstring representation of each bipartition in tree T_i, where n represents the number of taxa. In the n-bit bitstring, 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'. In our example, taxa on the same side of a bipartition as taxon A receive a '0'. In Figure 1 tree T_1's bipartitions are $AB|CDEF$, $ABD|CEF$, and $ABCD|EF$ which can be described by the bitstrings 001111, 001011, and 000011, respectively.

The hash function h_1 is used to generate the location (index) for storing a bipartition in the hash table. h_2 is responsible for creating a unique and short bipartition identifier (BID) for the bipartition so that the entire n-bit bitstring does not have to be analyzed in order to insert bipartitions into the hash table. Our two universal hash functions are defined as follows: $h_1(B) = \sum b_i a_i \bmod m_1$ and $h_2(B) = \sum b_i a_i \bmod m_2$, where $A = (a_1, ..., a_n)$ is a list of random integers in $(0, ..., m_1\text{-}1)$ and $B = (b_1, ..., b_n)$ is a bipartition represented as an n-bit bitstring. m_1 represents the number of entries (or locations) in the hash table. m_2 represents the largest bipartition ID (BID) given to a bipartition. b_i represents the ith bit of the n-bit bitstring representation of the bipartition B.

Figure 3(a) shows how the bipartitions from Figure 1 are stored in our hash table. Each entry in the hash table consists of BID, a bitstring representation of the bipartition, and a list of trees that contain that bipartition. 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 $\frac{1}{m_1}$. In Figure 3, $H[1]$ and $H[8]$ show two different bipartitions colliding to the same location in the hash table. Bipartitions $ABCF|DE$ and $ABCE|DF$ both reside in $H[1]$ and $ABCD|EF$ and $ABD|CEF$ reside in $H[8]$. However, these colliding bipartitions are differentiated by their h_2 hash value. In location $H[1]$, h_2 values 56 and 81 differentiate bipartitions $ABCF|DE$ and $ABCE|DF$, respectively.

The probability of a double collision ($h_1(B_i) = h_1(B_j)$ and $h_2(B_i) = h_2(B_j)$) is $O(\frac{1}{c})$, where c can be an arbitrarily large number [1]. Double collisions often result in an incorrect result for the underlying application. In our experience with using these hash functions in our phylogenetic applications (HashCS, HashRF,



(a) Hash table

1. TAXA␣A:B:C:D:E:F
2. NTREES␣6
3. NBIPARTITIONS␣8
4. L2K4␣-:0:
5. L3K2A␣+:1:E
6. L3BAB␣+:1:F
7. L3K3␣-:2:K2
8. L2BL2B␣+:1:C
9. L4K2␣+:3:ABC
10. L2BAK2␣+:1:B
11. L2K2AB␣+:1:C

(b) TreeZip (`.trz`) file

Fig. 3. TreeZip compressed file, which was obtained from our hash table data structure, for the phylogenetic trees shown in Figure 1. The symbol ␣ represents a visible space that is in the TreeZip file.

and TreeZip), we have not encountered double collision even when using small c values. For t trees on n taxa, $O(nt)$ time is required to construct the hash table.

Converting the hash table to `.trz` *format.* Once all of the bipartitions are organized in the hash table, we can begin the process of writing the `.trz` compressed file, which is binary. Figure 3(b) shows a compressed version of the hash table in Figure 3(a). The first three lines of the `.trz` file represent the taxa names, the number of trees in the file, and the number of unique bipartitions denoted by lines 1–3 in the `.trz` file in Figure 3(b). The remaining lines in the `.trz` file are related to the bipartitions contained in the t evolutionary trees. Each of the remaining lines is composed of two parts (n-bit bitstring and list of tree ids) separated by a single space.

We run-length encode our bitstrings. Run-length encoding is a form of data compression in which runs of data (i.e, sequences in which the same data value occurs in many consecutive data elements) are stored as a single data value and count, rather than as the original run. For the bitstring 001111 in Figure 3, we would have a run-length encoding of 0:2 1:4, where each $x : y$ element represents the data value (x) and the number of repetitions (y). Since bitstrings can either contain runs of 1s or 0s, we introduce two new symbols. `1:` is encoded as K, while `0:` encoded as L. (We use characters A through J for compressing our list of tree ids described shortly.) Hence, we encode the bitstring 001111 as L2K4. In our experiments, we considered taking every group of 7 bits in our bitstring and translating it to an ASCII character. However, we were able to get better compression by using run-length encoding, which showed significant benefits on our biological tree collections consisting of thousands of taxa.

The set of unique bipartitions comprise the remaining portion of the `.trz` file. Let $\mathcal{T}$ represent the set of evolutionary trees of interest, where $|\mathcal{T}| = t$. For a bipartition B, $\mathcal{B}_{in}$ represents the set of the trees in $\mathcal{T}$ that share that bipartition. $\mathcal{B}_{out}$ is the set of trees that do not share bipartition B. Since these

sets are complements, their union comprises the set $\mathcal{T}$. To minimize the amount of information present in our `.trz` output, we print out the contents of the smaller of these two sets. If $|\mathcal{B}_{in}| \leq |\mathcal{B}_{out}|$, then we output $\mathcal{B}_{in}$. Otherwise, $\mathcal{B}_{out}$ is outputted. In our `.trz` file, we denote $\mathcal{B}_{in}$ and $\mathcal{B}_{out}$ lines with the `'+'`and `'-'` symbol, respectively.

Even with use of the smaller of the $\mathcal{B}_{in}$ or $\mathcal{B}_{out}$ sets, the list of tree ids can get very large. This is due to the fact that as t grows large, the number of bytes necessary to store a single id also grows. Since the trees are inserted into the hash table in their order of appearance in the Newick file, our lists of tree ids will be in increasing order. As a result, we store the differences between adjacent elements in our tree id list. These differences are then run-length encoded. To eliminate the need for spaces between the run-length encoded differences, the first digit of every element is encoded as a character, with $0 \ldots 9$ represented by $\mathrm{A} \ldots \mathrm{J}$. Consider bipartition $ABCD|EF$ (bitstring 000011), which is in $H[8]$ in Figure 3. The $\mathcal{B}_{in}$ set will be used for this bipartition, and its run-length encoded differences will be 0 1 2, which will be encoded as `ABC` on line 9 in the `.trz` file.

Finally, one of the guiding factors for our TreeZip format is not only effective compression, but also readability. We did try several different compression schemes for our TreeZip approach, but the compression algorithm described here gave the best compression along with the best decompression times (not shown).

2.2 Decompression: Converting the `.trz` File to a Newick File

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

To assist in bipartition collection, we maintain two data structures. The first, which we will refer to as V, is a vector of the bipartitions contained in *all* of the t trees. The second, M, is a $t \times k$ matrix, where $k = n - 3$ is the maximum possible number of bipartitions for a phylogenetic tree. The length of the matrix M corresponds to the number of trees specified in the `.trz` file. Each row i in matrix M corresponds to the bipartitions required to rebuild tree T_i. For example, in figure 3, the bipartition 000011 is shared among all the trees. It is therefore added to vector V. On the other hand, the bipartitions on lines 5 and 6 are contained in only trees 4 and 5 respectively, and therefore will be added to $M[4]$ and $M[5]$. The bipartition on line 9 will be added to $M[0], M[1]$, and $M[3]$ since `ABC` decodes to the tree ids T_0, T_1, and T_3. Line 7 in our `.trz` file warrants special attention. Since this line belongs to the set $\mathcal{B}_{out}$, we know upon decoding that this bipartition does *not* belong to trees 1 and 2. Therefore, the bipartition is added to rows $M[0], M[3], M[4]$, and $M[5]$.

The decoded bitstrings are the basic units for building trees. Once the bitstrings and the associated tree ids are decoded, we can build the original trees one by one. In order to build tree x, the tree building function receives as input

the vector V containing bipartitions shared among all of the trees and matrix row $M[x]$ which contains the bipartitions encoded as bitstrings for tree x. Since vector V contains the bitstrings common to all the trees, it is always passed to the tree building function.

Each of the t trees is built starting from tree T_0 and ending with tree T_{t-1}, whose bipartitions are stored in $M[0]$ and $M[t-1]$, respectively. The trees are reconstructed in the same order that they were in the original Newick file. However, given $O(2^{n-1})$ possible Newick strings for a tree T_i, the Newick representation that TreeZip outputs for tree T_i will probably differ from the Newick string in the original file. However, this is not a problem semantically since the different strings represent the same tree.

In order to build tree T_i, the bitstrings in matrix $M[i]$ and vector V are merged into a single array of bitstrings. Initially, tree T_i is represented as a star tree on n taxa. Bipartitions from $M[i]$ are added to refine tree T_i based on the number of 1's in its bitstring representation. (The number of 0's could have been used as well.) The more 1's in the bitstring representation, the more taxa that are grouped together by this bipartition. A star tree is an bitstring representation consisting of all 1's. For each of T_i's bitstrings, we count the number of 1's it contains. Bipartitions are then sorted in increasing order of their bitstrings, which means that bipartitions that group together the most taxa appear first. The bipartition that groups together the fewest taxa appears last in the sorted list of '1' bit counts. For each bipartition, a new internal node in tree T_i is created. Hence, the bipartition is scanned to put the taxa into two groups—taxa with '0' bits compose one group and those with '1' bits compose the other group. The taxa indicated by the '1' bits become children of the new internal node. The above process repeats until all bipartitions are added to tree T_i.

3 Experimental Results

Our implementation of TreeZip used in the following experiments can be found at `http://treezip.googlecode.com`. Experiments were conducted on a 2.5Ghz Intel Core 2 quad-core machine with 4GB of RAM running Ubuntu Linux 8.10. We ran our experiments on fourteen sets of trees which are described in Table 1. We use the *compression ratio* measure to evaluate the performance of TreeZip in comparison to general-purpose compression algorithms. The compression ratio C is calculated as $C = \frac{|\text{compressed file}|}{|\text{original file}|}$. This result is multiplied by 100 to achieve a percentage. A lower compression ratio denotes better compression of the original file.

3.1 Performance on the TASPI Tree Collection

In Figure 4, we compare the compression ratio achieved by TreeZip and TASPI on the 9 tree collections used in Collection 3 of [2], which is denoted by datasets 6 through 14 in Table 1. We also show the compression ratio of standard compression approaches (`gzip`, `bzip,` and `7zip`) achieved on this set of trees, along with

Table 1. Characteristics of our biological tree files. The `mammals`, `freshwater`, `angiosperms`, `fish`, and `insects` datasets were given to us by biologists. The remaining tree collections are the same ones used by Boyer et al. to evaluate their TASPI approach.

	Datasets	Description	Taxa	Trees	File size (MB)	Bipartitions
1	`mammals`	Mammalian trees [6]	16	8,000	0.6	13
2	`freshwater`	Organisms from freshwater, marine, and oil habitats [7]	150	20,000	16.0	1,168
3	`angiosperms`	Flowering plants [9]	567	33,306	105.0	2,444
4	`fish`	Fish trees (unpublished collection from M. Glasner's lab at Texas A&M)	264	90,000	127.0	12,115
5	`insects`	Insect trees [8]	525	150,000	434.0	574
6	`aster328`	Tree Collection 3 from Boyer et al. [2]	328	2,505	5.3	788
7	`eern476`		476	2,505	7.7	3,019
8	`john921`		921	2,505	16.0	15,448
9	`lipsc439`		439	2,505	7.1	903
10	`mari2594`		2,594	2,505	47.0	8,628
11	`ocho854`		854	2,505	15.0	3,232
12	`rbcl500`		500	2,505	8.2 (8.1 in [2])	1,579
13	`three567`		567	2,505	9.3	1,588
14	`will2000`		2,000	2,505	36.0	13,257

the ratio of TreeZip coupled with each of these standard approaches. Since an implementation of TASPI is not available publicly, the compression ratio numbers for TASPI were calculated directly from [2]. Since TASPI coupled its approach with the `bzip` algorithm, we highlight the compression ratio achieved by TASPI and TASPI+`bzip` (blue), as well as TreeZip and TreeZip+`bzip` (red). TASPI did not couple its approach with either `7zip` or `gzip`.

TreeZip achieves a better (lower) compression ratio than TASPI across all the listed datasets. For example, on the `lipsc439` dataset, TreeZip achieves a compression ratio of 1.592%, while TASPI achieves a compression ratio of 5.57%. This corresponds to a file size of 116 kilobytes and 406 kilobytes respectively. On the `mari2594` dataset, TreeZip achieves a compression ratio of 2.34%, compared to TASPI's 7.02%. This corresponds to compressed file sizes of 1.1 MB and 3.3 MB respectively.

When coupled with `bzip`, TASPI achieves a slightly better compression ratio than TreeZip+`bzip` on most of the datasets. However, these differences are often negligible. For example, on the `three567` dataset, TreeZip+`bzip` has a compression ratio of 0.63% compared to TASPI+`bzip`'s 0.47%. This corresponds to 60 and 45 kilobytes respectively, a difference of 15 kilobytes. On the `lipsc439` dataset, TreeZip+`bzip` achieves a compression ratio of 0.55%, compared to TASPI+`bzip`'s 0.48%. This corresponds to compressed files of 40 and

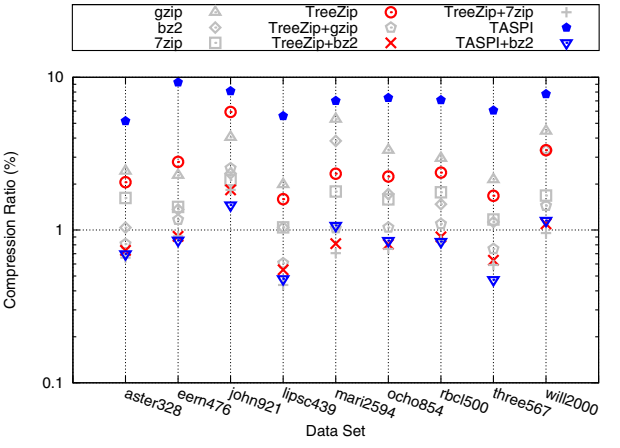


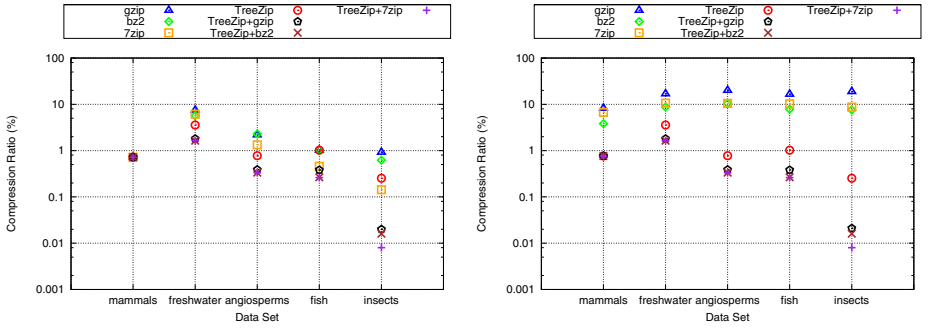
Fig. 4. Compression ratios for various algorithms on Newick string representations of evolutionary trees. TASPI and TASPI+bz2 numbers come from [2].

34.8 kilobytes in size, respectively. On the largest dataset of this set, `mari2594`, TreeZip+`bzip` outperforms TASPI, achieving a compression ratio of 0.81% compared to TASPI+`bzip`'s 1.07%. This corresponds to file sizes of and 392 and 515 kilobytes, respectively, a difference of 123 kilobytes.

3.2 Performance on Tree Sets Provided by Biologists

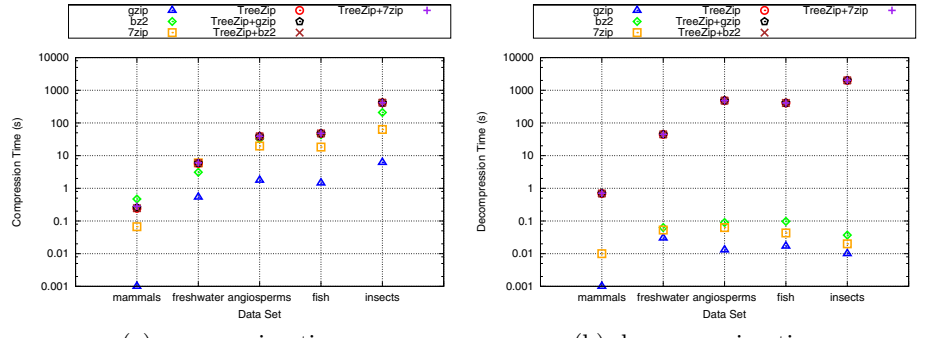
Figure 5(a) shows the performance of TreeZip on the large tree collections (Datasets 1 through 5 in Table 1) given to us by biologists. By itself, TreeZip achieves similar storage to the standard compression algorithms on our biological tree sets. Since all of the trees in the `mammals` dataset are identical, all approaches achieve the same compression ratio and storage size of 4 kilobytes. For our `fish` dataset, `7zip` outperformed TreeZip and the other standard compression approaches, achieving a ratio of 0.46%. TreeZip, on the other hand, had a compression ratio of 1.02%. This corresponds to a size of 596 kilobytes compared to TreeZip's 1.3 megabytes. Coupling TreeZip with standard compression techniques results in improved performance. Returning back to our `fish` dataset, TreeZip+`7zip` achieves a compression ratio of 0.261%, which corresponds to 340 kilobytes. This is most evident for our `insects` dataset, where TreeZip+`7zip` achieves a compression ratio of 0.008%, or roughly 36 kilobytes. On this same dataset, `7zip` has a compression ratio of 0.14% resulting in a compressed file of 636 kilobytes. Our results suggest that the greater the level of bipartition sharing and the number of trees, the better TreeZip will perform, especially when coupled with the `7zip` approach.

One critical advantage of TreeZip is that it collapses the topologies of the phylogenetic trees into a set of common bipartitions, ensuring that each bipartition appears at most once in the compressed form. Both standard compression techniques and TASPI compress trees at the *string* level. If the Newick string for



(a) original Newick file (b) different, but equivalent, Newick file

Fig. 5. Compression ratios of two different Newick files representing the same set of evolutionary trees



(a) compression time (b) decompression time

Fig. 6. Compression and decompression times for the algorithms under study

a particular tree is rearranged denoting a different, but equivalent, Newick string representation of the same tree, text-based compression approaches will have difficulty identifying shared bipartitions among the t trees. Figure 5(b) shows the impact of using different, but equivalent Newick representations (see Figure 2) in our biological tree collections. While TreeZip's performance remains the same, the compression ratio and storage requirements for the standard compression methods explode. For example, for the `fish` dataset, `7zip`'s compression ratio increases from 0.46% to 10.24%. This corresponds to an increase from 596 kilobytes to 13 megabytes. TASPI's storage requirements would also increase under different, but equivalent, Newick strings. In contrast, TreeZip and TreeZip+`7zip` still requires only 1.3 megabytes and 340 kilobytes of storage, respectively.

While TreeZip competes against standard compression algorithms in terms of storage size, it does so at the cost of running time (see Figure 6). While TreeZip's compression speed is about twice as slow as `bzip` and `7zip` (`gzip` runs extremely fast requiring less than 10 seconds on our datasets), its decompression speed is

very slow. All of the methods require less than a second to decompress, while TreeZip can take anywhere from a second to 3,000 seconds. Obviously, this is a place that needs optimization. However, if the user is merely interested in compressing tree files as part of an archive with very little chance for decompression, then TreeZip is a desirable alternative to standard compression techniques. Furthermore, if there is no predefined ordering of the taxa in the Newick file, then using TreeZip will result in a very small file compared to the alternatives given the robustness of the TreeZip approach.

4 Conclusions and Future Work

Phylogenetic heuristics often produces tens to hundreds of thousands of equally-plausible trees, which are usually stored in a Newick-formatted text file. Due to the number of trees, the size of the input file is easily over hundreds of megabytes making it difficult to store, maintain, and exchange the tree files. In this paper, we introduce our *TreeZip* algorithm, a novel approach that leverages the semantic information among trees to compress the tree files. The advantage of TreeZip over current methods is its ability to uniquely identify shared bipartitions and store this information in a compressed TreeZip (`.trz`) file, which consumes considerably less storage space than the original Newick file.

Our TreeZip algorithm outperforms standard compression methods by achieving a better compression ratio. For example, our results show that our `.trz` file occupies from 0.2% to 2% of the original Newick file, which outperforms `gzip` and `bzip` algorithms. When TreeZip is coupled with standard compression algorithms, it is the best compression technique for phylogenetic trees. Thus, TreeZip can work on two levels. It can work at the `.trz` file level, where the file can be used as input for other phylogenetic tree algorithms. The benefit of the `.trz` file is that it is readable and can be queried more easily (without decompression) than the Newick file regarding the evolutionary relationships contained in the collection of t trees. Coupling TreeZip with text compression algorithms such as `7zip` produces the best storage savings. In addition, a phylogenetic tree can be represented using several different (yet equivalent) Newick string representations. This proves disastrous for standard compression methods, which perform poorly in the absence of any available redundancy at the Newick string level. TreeZip, on the other hand, performs well on such datasets.

Overall, TreeZip's efficient method for compressing trees allows large phylogenetic tree collections to be easily exchanged with others, an essential component for successful scientific collaborations. Without compression, sharing data can become quite tedious, especially across long distances. As biologists obtain more data and use phylogenetic heuristics to build large-scale evolutionary trees, the size of their tree collections will continue to grow in size. Thus, compression algorithms such as TreeZip will become critical tools for helping biologists manage their rapidly expanding phylogenetic tree collections.

In the future, we will optimize TreeZip for speed since the focus in this work was the quality of the compression achieved. Due to the inherent flexibility of

the compressed format, we plan to add in functionality to incorporate new, additional trees into an existing compressed collection.

Acknowledgments

Funding for this project was supported by NSF under grants DEB-0629849 and IIS-0713618.

References

1. Amenta, N., Clarke, F., John, K.S.: A linear-time majority tree algorithm. In: Benson, G., Page, R.D.M. (eds.) WABI 2003. LNCS (LNBI), vol. 2812, pp. 216–227. Springer, Heidelberg (2003)
2. Boyer, R.S., Hunt Jr., W.A., Nelesen, S.: A compressed format for collections of phylogenetic trees and improved consensus performance. Technical Report TR-05-12, Department of Computer Sciences, The University of Texas at Austin (2005)
3. Boyer, R.S., Hunt Jr., W.A., Nelesen, S.: A compressed format for collections of phylogenetic trees and improved consensus performance. In: Casadio, R., Myers, G. (eds.) WABI 2005. LNCS (LNBI), vol. 3692, pp. 353–364. Springer, Heidelberg (2005)
4. Felsenstein, J.: The Newick tree format. Internet Website (last accessed January 2010), Newick, `http://evolution.genetics.washington.edu/phylip/newicktree.html`
5. Huelsenbeck, J.P., Ronquist, F.: MRBAYES: Bayesian inference of phylogenetic trees. Bioinformatics 17(8), 754–755 (2001)
6. Janecka, J.E., Miller, W., Pringle, T.H., Wiens, F., Zitzmann, A., Helgen, K.M., Springer, M.S., Murphy, W.J.: Molecular and genomic data identify the closest living relative of primates. Science 318, 792–794 (2007)
7. Lewis, L.A., Lewis, P.O.: Unearthing the molecular phylodiversity of desert soil green algae (chlorophyta). Syst. Bio. 54(6), 936–947 (2005)
8. Molin, A.D., Matthews, S., Sul, S.-J., Munro, J., Woolley, J.B., Heraty, J.M., Williams, T.L.: Large data sets, large sets of trees, and how many brains? – Visualization and comparison of phylogenetic hypotheses inferred from rdna in chalcidoidea (hymenoptera) (poster December 2009), `http://esa.confex.com/esa/2009/webprogram/Session11584.html`
9. 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), 137–157 (2007)
10. Sul, S.-J., Williams, T.L.: An experimental analysis of robinson-foulds distance matrix algorithms. In: Halperin, D., Mehlhorn, K. (eds.) ESA 2008. LNCS, vol. 5193, pp. 793–804. Springer, Heidelberg (2008)
11. Sul, S.-J., Williams, T.L.: An experimental analysis of consensus tree algorithms for large-scale tree collections. In: Măndoiu, I., Narasimhan, G., Zhang, Y. (eds.) ISBRA 2009. LNCS, vol. 5542, pp. 100–111. Springer, Heidelberg (2009)