An Efficient and Extensible Approach for Compressing Phylogenetic Trees

Suzanne J. Matthews and Tiffani L. Williams
[sjm, tlw]@cse.tamu.edu
Department of Computer Science and Engineering
Texas A&M University, College Station, Texas, USA
Technical Report No: 2011-05-02

Abstract

Biologists require new algorithms to efficiently compress and store their large collections of phylogenetic trees. TreeZip is a novel method for compressing phylogenetic trees. Recently, we extended our TreeZip algorithm to support branch lengths and show how it can be used to extract sets of trees of interest quickly. The key advantage of TreeZip over standard compression methods like 7zip is its ability to interpret and compress tree collections semantically, making it immune to branch rotations and allowing key operations (such calculating a consensus tree) to be performed quickly and without a loss of space savings. On unweighted phylogenetic trees, TreeZip is able to compress Newick files in excess of 98%. On weighted phylogenetic trees, TreeZip is able to compress a Newick file by at least 73%. TreeZip can be combined with 7zip with little overhead, allowing space savings in excess of 99% (unweighted) and 92%(weighted). Unlike TreeZip, 7zip is not immune to branch rotations, and performs worse as the level of variability in the Newick string representation increases. Finally, since the TreeZip compressed text (TRZ) file contains all the semantic information in a collection of trees, we can easily filter and decompress a subset of trees of interest (such as the set of unique trees), or build the resulting consensus tree in a matter of seconds. We also show the ease of which set operations can be performed on TRZ files, at speeds quicker than those performed on Newick or 7zip compressed Newick files, and without loss of space savings. TreeZip is an efficient approach for compressing large collections of phylogenetic trees. The semantic and compact nature of the TRZ file allow it to be operated upon directly and quickly, without a need to decompress the original Newick file. We believe that TreeZip will be vital for compressing and archiving trees in the biological community.

1 Background

In a phylogenetic tree, living organisms occupy the leaves and ancestral organisms are internal nodes, with the edges of the tree denoting evolutionary relationships (see Figure 1). The task of phylogenetics is to infer this tree from observations (e.g., molecular sequences) obtained from existing organisms of interest. To reconstruct a phylogenetic tree, the most popular techniques (such as MrBayes [5] and TNT [4]) often return tens to hundreds of thousands of trees that represent equally-plausible or closely-related hypotheses (or candidate trees) for how the taxa evolved from a common ancestor. Given that phylogenetic searches return tens to hundreds of thousands of candidate evolutionary trees, biologists need new techniques for managing and sharing these large tree collections effectively. As biologists obtain more data to produce evolutionary trees, phylogenetic techniques must reconstruct larger trees, resulting in ever-larger collections of candidate trees. Thus, there is a critical need to develop phylogenetic compression techniques that reduce the requirements of storing large tree collections so that they can be shared easily with colleagues around the world.
Figure 1: A collection of six evolutionary trees on six taxa labeled \(A\) to \(F\). Each edge \(e_i\) represents an evolutionary relationship (or bipartition) along with a value that represents the length of the branch.

\[
T_0 = (((A:0.12,B:0.14):0.32,C:0.81):0.34,(D:0.91,(E:0.63,F:0.40):0.23):0.21):0.21);
T_1 = (((A:0.32,B:0.12):0.42,D:0.04):0.31,(C:0.62,(E:0.82,F:0.92):0.33):0.33):0.15);
T_2 = (((A:0.24,B:0.63):0.31,E:0.72):0.43,(D:0.01,(C:0.22,F:0.33):0.11):0.46);
\]

(a) Newick string representations

\[
T_0 = ((C:0.81,(B:0.14,A:0.12):0.32):0.34,(D:0.91,(E:0.63,F:0.40):0.23):0.21):0.21);
T_1 = ((B:0.12,A:0.32):0.42,D:0.04):0.31,(C:0.62,(E:0.82,F:0.92):0.33):0.15);
T_2 = ((E:0.72,(A:0.24,B:0.63):0.31):0.43,(D:0.01,(C:0.22,F:0.33):0.11):0.46);
\]

(b) different, but equivalent Newick string representations

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

1.1 Our Contributions

Recently, we introduced TreeZip \([8]\), a novel compression algorithm that reduces the requirements over standard compression algorithms (such as \texttt{7zip}) for storing and sharing large collections of evolutionary trees. Given that many of the evolutionary relationships in a collection of phylogenetic trees are shared, the novelty of the TreeZip approach is storing such relationships only once in the compressed representation. TreeZip compresses a Newick file based on the \textit{semantic} representation (i.e., tree bipartitions and unique topologies). The Newick format \([3]\) is the most widely used file format to represent phylogenetic trees. In this format, the topology of the evolutionary tree is represented using a notation based on balanced parentheses (see Figure 2). A Newick formatted tree uses nested parentheses to represent the evolutionary relationships (or subtrees) within a phylogenetic tree. In a Newick file, each tree is located individually on separate lines. Figure 2 shows two sets of different, but equivalent Newick representations for the three trees shown in Figure 1. Matching pairs of parentheses symbolize internal nodes in the evolutionary tree. However, the Newick representation of a tree is not unique. For a particular tree with \(n\) taxa, there are \(O(2^{n-1})\) different (but equivalent) Newick strings to represent its topology. Consequently, general-purpose data compression techniques (e.g., \texttt{gzip}, \texttt{bzip}, and \texttt{7zip}) cannot recognize (or leverage) domain-specific information regarding the Newick file (data). Thus, there is great potential for obtaining good compression by utilizing the semantic information in a Newick file describing large collections of evolutionary trees.

Previously, we showed that our TreeZip approach is an effective algorithm for compressing the set of evolutionary relationships in a collection of phylogenetic trees \([8]\). In this paper, we improve the TreeZip algorithm in three significant ways. First, we extend TreeZip to handle phylogenetic trees containing \textit{branch lengths} as shown in Figure 1. Next, we also store unique topologies only once in the file, in addition to unique bipartitions. Lastly, we show the \textit{extensibility} of the TreeZip compressed format, when given a Newick file.
representing the collection of \( t \) trees of interest. Our previous work focused on designing a decompressor that takes a TreeZip compressed file and outputs all of its trees into a Newick file. However, the TreeZip format contains valuable phylogenetic information for performing extraction operations (such as the set of unique trees) and constructing consensus trees. Since the TreeZip compressed (TRZ) file is text, we also show that set operations, such as union, intersection and set difference can be performed several times faster on a TRZ file than on a Newick file without loss of space savings.

Besides TreeZip, the only other known phylogenetic tree compressor is the Texas Analysis of Symbolic Phylogenetic Information (TASPI) algorithm [2,6]. The benefits of TASPI include compressing phylogenetic trees and computing their consensus. TreeZip shares these benefits as well as the ability to handle branch lengths and perform set operations (i.e., extracting the set of unique trees) on the trees in its compressed representation. Also, the authors of TASPI state that their approach is not robust to the \( O(2^n - 1) \) different Newick representations of a phylogenetic tree. Our experimental results show that TreeZip’s performance is not impacted by these different Newick representations.

1.2 Summary of Results

We experimentally study the performance of our TreeZip algorithm in comparison to 7zip on five biological data sets, including freshwater (20,000 trees over 150 taxa), angiosperms (33,306 trees over 567 taxa), fish (90,002 trees over 264 taxa) and insects (150,000 trees over 525 taxa) tree collections. Of these datasets, the first four are weighted (have branch lengths), while the last is unweighted. Our largest (smallest) tree collection consists of 150,000 (20,000) trees requiring 434 MB (67 MB) of storage space. However, due to the storage requirements of weighted tree collections, our fish dataset consisting of 90,002 264-taxa trees has the largest file size of 533 MB. Overall, our results show that the compressed TreeZip (TRZ) file is over 74% smaller than the original Newick file on weighted collections. On unweighted collections, it is 98% smaller. When TreeZip is coupled with 7zip, the resulting TRZ+7zip file is on average 92% smaller on weighted collections. On unweighted collections, the TRZ+7zip file is in excess of 99.8% smaller than the original Newick file. Given that there are \( O(2^n - 1) \) different Newick representations for a phylogenetic tree, we study the impact of these different, but equivalent representations on both the TreeZip and 7zip approaches. The results show that as the number of different Newick representations increases, there is a significant increase in 7zip’s compressed representation. TreeZip, on the other hand, is robust to changes in the Newick string representation of a tree. Furthermore, not only does TreeZip produce a smaller compressed file than 7zip, it often does so in a time that is faster or comparable to 7zip.

Beyond decompressing a TRZ file to its original Newick representation, our experiments provide exciting results related to the flexibility of extracting additional information from the compressed file. Of interest to biologists are the unique set of trees that are contained in their tree collection (or compressed TreeZip file). Moreover, we can output the strict and majority consensus trees from the phylogenetic data in the TRZ file in less than one second on the tree collections studied in this paper. Since the TRZ file is text, various set operations can be quickly and efficiently performed on the TRZ representation of a weighted (unweighted) collection of trees up to 5 (60) times faster than on the Newick representation. Thus, our results show that the TRZ file is an effective and extensible compressed format that biologists can leverage to manage their large tree collections.

1.3 Structure of Paper

The rest of this paper is organized as follows. Section 2 describes the TreeZip algorithm, including the mechanisms behind compression, decompression, and our set operations functionality. We also describe our experimental methodology. Section 3 describes and discusses our experimental results. Lastly, we conclude in Section 4.
2 Methods

The TreeZip algorithm is composed of two main parts: compression and decompression. We discuss each of these in turn in the subsections that follow. Section 2.1 discusses the process of compression, in which an inputted Newick file is transformed into the TreeZip compressed format, or TRZ file. Section 2.2 discusses decompression, in which a TRZ file is used to reconstruct the desired set of phylogenetic trees in Newick format. We note here that since any phylogenetic tree with \( n \) taxa has \( O(2^n-1) \) equivalent Newick string representations, any one of these equivalent Newick string representations can be used as the decompressed version. Section 2.3 describes the algorithm behind the TreeZip set operations. Unlike the compression and decompression functions, the TreeZip set operations take as input two TRZ files, and outputs a single TRZ file. In this manner, set operations are performed in the context of a TRZ file, without any loss of space savings. Lastly, we summarize our experimental methodology in Section 2.4.

2.1 Compression

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]. There are \( 2(n-1) \) total bipartitions contained in each tree \( T_i \), where \( n \) is the number of taxa. Thus, each of the 6-taxa trees in Figure 1 contains 10 bipartitions. Both of the universal hashing functions require as input a \( n \)-bit bitstring representation of each bipartition in tree \( T_i \). Taxa are ordered lexicographically, where \( h_0 \) represents the first bit and the first taxon name in the ordering, \( h_1 \) is the second bit representing the taxon in the ordering, etc. For the phylogenetic trees shown in Figure 1, the taxa ordering is \( A, B, C, D, E, \) and \( F \). The bitstring 100000 represents the bipartition \( A|B|C|D|E|F \), which corresponds to edge \( e_1 \) in each of the three trees. In tree \( T_0 \), edge \( e_5 \) corresponds to the bipartition \( A|B|C|D|E|F \) or bitstring 111000, which is formed by performing an \( \text{OR} \) operation on the bitstrings of its children represented by edges \( e_3 \) and \( e_4 \). The bipartition \( D|E|F|A|B|C \) corresponds to the bitstring 000111, which has a branch length of 0.21 denoted by edge \( e_{10} \) in tree \( T_0 \).

2.1.1 Step 1: Storing bipartitions in a hash table

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: 

\[
h_1 (B) = \sum b_ir_i \mod m_1 \quad \text{and} \quad h_2 (B) = \sum b_is_i \mod m_2.
\]

\( R = (r_0, \ldots, r_{n-1}) \) is a list of random numbers in the range of \( (0, \ldots, m_1 - 1) \), and \( S = (s_0, \ldots, s_{n-1}) \) is a list of random integers between \( (0, \ldots, m_2 - 1) \). \( B = (b_0, \ldots, b_{n-1}) \) 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 \( i \)th bit of the \( n \)-bit bitstring representation of the bipartition \( B \).

Figure 3 shows how the bipartitions from Figure 1 are stored in our hash table. Each entry represents a unique bipartition and the hash line consists of a BID, its bitstring representation, a list of trees that contain that bipartition, and the respective branch lengths for each of trees. In this figure, \( R = (22, 45, 19, 27, 12, 20) \), \( S = (32, 42, 24, 31, 16, 26) \), \( m_1 = 37 \), and \( m_2 = 3,701 \).

The first three lines of the compressed TRZ file represent the taxa names, the number of trees in the file, and the number of unique bipartitions. Afterwards, we process each hash table row which will represent a line in the compressed file. There are three components (bitstrings, tree ids, branch lengths) to a TRZ line. We also note here that bipartitions stored in the TRZ file are stored in sorted order according to the number of ones they contain. Ties are broken lexicographically. This guarantees that if two tree collections have equivalent one-to-one corresponding sets of trees, the TRZ files of the two collections will be identical despite differences in the Newick string representations. Below, we describe how TreeZip encodes each of these components.
2.1.2 Step 2: Encoding bitstrings

Once all of the bipartitions are organized in the hash table, we begin the process of writing the TRZ compressed file, which is a plain text file. 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 110000 in Figure 3, we would have a run-length encoding of 1:2 0:4, where each x:y element represents the data value (x) and the number of repetitions (y). The \( \_ \) character denotes a space. 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 110000 as K2L4. 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 hundreds of taxa.

2.1.3 Step 3: Identifying and encoding the set of unique tree ids

Let \( T \) represent the set of evolutionary trees of interest, where \( |T| = t \). For a bipartition \( B \), \( B_{in} \) represents the set of the trees in \( T \) that share that bipartition. \( B_{out} \) is the set of trees that do not share bipartition \( B \). Since these sets are complements, their union comprises the set \( T \). To minimize the amount of information present in our TRZ output, we print out the contents of the smaller of these two sets. If \( |B_{in}| \leq |B_{out}| \), then we output \( B_{in} \). Otherwise, \( B_{out} \) is outputted. In our TRZ file, we denote \( B_{in} \) and \( B_{out} \) lines with the ‘+’ and ‘-’ symbol, respectively.

Even with use of the smaller of the \( B_{in} \) or \( B_{out} \) 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. We note first that a tree \( T \) can be represented as a k-bit bitstring, where \( k \) is the number of bipartitions discovered in the collection. If we feed these k-bit bitstring representations into a slightly modified version of the above hash functions, we can obtain the set of unique trees, \( U \), where \( |U| = u \). This set of unique trees are given the corresponding tree ids of 0 . . . u – 1, and will represent the total set of trees in consideration with any bipartition. Duplicate information is encoded and stored at the end of the TRZ file.

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...9 represented by A...J. Consider bitstring 000011, which is in row 30 (its $h_1$ value) in our hash table shown in Figure 3 and has an $h_2$ value of 42. The $B_{out}$ set will be used for this bipartition, and its run-length encoded differences will be 2, which will be encoded as C. Given the large number of shared bipartitions in a collection of trees that result from a phylogenetic search, there will be many more unique trees than unique bipartitions. Hence, encoding the differences in the tree ids leverages the sharing among the trees—especially since 1 is the most common difference between adjacent elements in the tree id lists.

### 2.1.4 Step 4: Encoding branch lengths.

The last item to process on a hash line are the branch lengths associated with a unique bipartition. Branch lengths take the form $x.y$, where $x$ is the integral and $y$ the mantissa. For this domain, branch lengths tend to be very small ($x = 0$). Hence, we use this property to our advantage by only encoding the integral in special cases ($x > 0$). For these special cases, we store the integral separately along with its related tree id. On the datasets studied here, at least 99.6% of the branch lengths begin with 0. The mantissa corresponds to a fixed number, $k$, of digits. For our tree collections, $k = 6$. To encode the mantissa, we take two digits at a time (since we can guarantee this value fits into a byte) and translate it into a readable ASCII character. For example if we have a value of 99 as input, we add 33 to create the corresponding Extended ASCII readable character á. It is necessary to add 33 to any inputted value since the first 32 characters in ASCII are non-printable, control characters. We tried different universal integer encodings (e.g., variable byte encoding) [12], but given the range of integers represented by $k$ digits, the various integer encodings did not result in a smaller compressed file. This is due to the fact that when $k = 6$, universal codes become less effective than straight binary as the size of the integers themselves increase [12]. Furthermore, we achieve better compression by feeding the resulting TRZ file to a general-purpose compressor such as 7zip. However, when using variable byte encoding for the branch lengths, 7zip’s algorithm could not reduce the file size any further and resulted in a much larger compressed file.

Lastly, we note that branch lengths are not compacted like tree ids since the branch lengths originate from an infinite set of real numbers. Tree ids, on the other hand, are drawn from a finite set of $t$ tree ids ranging from 0 to $t - 1$.

### 2.2 Decompression

The 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 bipartition 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.

### 2.2.1 Decompression data structures

To assist in bipartition collection, we maintain two data structures $M$ and $N$, both which are $t \times k$ matrices, where $k = 2n$ is the maximum number of bipartitions for a phylogenetic tree. The length of each matrix 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 $T_i$. The corresponding row in matrix $N$ is the list of associated branch lengths. For example, in Figure 3, the bipartition at row 32 of our hash table (it’s $h_1$ value) is shared among all the trees. It is therefore added to every row in $M$. To $N$, we add the value 0.32 to $N[0]$, 0.42 to $N[1]$ and 0.31 to $N[2]$, signifying that these are the associated branch lengths for the corresponding bipartition in $M$. On the other hand, the bipartitions 11100 and 001001 are contained only in trees $T_0$ and $T_2$ respectively, and therefore will be added to $M[0]$ and $M[2]$. Thus, we also add branch lengths 0.34 and 0.11 to $N[0]$ and $N[2]$. Since each bipartition is processed in order in our TRZ file, we are able to guarantee a one to one
correspondence between the values in $M$ and $N$. We also maintain a separate data structure that stores duplicate tree information to assist in the construction for $M$ and $N$.

### 2.2.2 Flexible decompression

The decoded bitstrings are the basic units for building trees. Once the bitstrings, associated tree ids and branch lengths are decoded, we can build the original trees one by one. In order to build tree $T_i$, the tree building function receives as input matrix row $M[i]$ which contains the bipartitions encoded as bitstrings for tree $T_i$, and matrix row $N[i]$ which contain the associated branch lengths for each bipartition in $M[i]$. Each of the $t$ trees is built starting from tree $T_0$ and ending with tree $T_{t-1}$, whose bipartitions (branch lengths) are stored in $M[0]$ ($N[0]$) and $M[t-1]$ ($N[t-1]$), respectively. The trees are reconstructed in the same order that they were in the original Newick file. However, given $O(2^n)$ 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. This is not a problem semantically since the different strings represent the same tree.

To build tree $T_i$, it is initially represented as a star tree on $n$ taxa. A star tree is an bitstring representation consisting of all 1’s. In the TRZ file, bipartitions are stored in decreasing order of their bitstrings. This means when it is time to rebuild trees, the 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 $i$, a new internal node in tree $T_i$ is created using the bitstring in $M[i]$, and the associated weight is added using the value in $N[i]$. Hence, 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 $T_i$.

The decompressor can also output sub-collections of trees of interest to the user. For example, if the user was interested in the set of unique trees in the collection (rather than the entire collection), TreeZip can return this set of trees of interest to the user. In addition, TreeZip has built-in functionality to return the strict and majority-rule consensus trees of an encoded collection of trees in a couple of seconds to the user. The strict and majority-rule consensus trees are especially of interest to biologists, since this is the summary tree structure that commonly appears in publications. Furthermore, these subcollections of trees can be produced directly from the TRZ file, without a need to decompress the original collection. In other words, operations can be performed directly on the TRZ file without requiring a loss of space savings. This is not the case with standard compression approaches which produce unreadable binary output. In these cases, the original file must always be fully decompressed in order for any operations to be performed, resulting in zero space savings.

### 2.3 TreeZip Set Operations

One of our goals is to show that the TRZ file represents a viable alternative archive format to the Newick file for representing large collections of trees. If the same set of operations can be performed on a TRZ file that can be done on a Newick file, then we can argue that the two file types are equivalent. In order to accomplish this goal, we implemented a series of set operations that exploits the textual structure of the TRZ file to produce sets of trees of semantic interest. The set operation functions in TreeZip takes as input two TRZ files and outputs a single TRZ file that represents the results of a particular set operation. Here, we implement three set operations in total: union, intersection, and set difference. The union between two collections of trees is defined as the set of unique trees that exist over both collections. The intersection between two collections is defined as the set of unique trees that exist in the first collection and the second collection. The set difference between two collections is defined as the set of unique trees that exist in the first collection and not in the second. For example, consider the collections stored in Files 1 and 2 in Table 1. In this example, each file contains three trees. For simplicity, we label these 1…6. We note that the first tree in both files (trees 1 and 4) are identical; they just have a different Newick string representations. The rest of the trees are distinct from each other. The union between Files 1 and 2 consists of of 5 trees (trees 1,2,3,5, and 6). The intersection consists of 1 tree (tree 1), and the set difference consists of 2 trees (trees 2 and 3).
Table 1: Two sample tree files of weighted trees.

| File 1 | File 2 |
|--------|--------|
| 1. (((A:0.12,B:0.13):0.14,C:0.15):0.16,(D:0.17,(E:0.18,F:0.19):0.20):0.21); | 4. (((E:0.18,F:0.19):0.20,D:0.17):0.21,(C:0.15,(A:0.12,B:0.13):0.14):0.16); |
| 2. (((A:0.11,B:0.34):0.29,D:0.23):0.22,(C:0.24,(E:0.25,F:0.26):0.27):0.28); | 5. (((A:0.34,B:0.23):0.21,C:0.53):0.24,(F:0.41,(E:0.13,D:0.51):0.21):0.33); |
| 3. (((A:0.29,B:0.11):0.31,E:0.33):0.15,(D:0.38,(C:0.36,F:0.37):0.32):0.31); | 6. (((A:0.12,B:0.43):0.21,C:0.06):0.20,(E:0.04,(D:0.28,F:0.33):0.02):0.41); |

TreeZip is able to perform these set operations (and other operations) quickly since the set of unique bipartitions and trees are known and already encoded into the TRZ file. TreeZip then uses this encoded information to create a new TRZ file with the set of desired trees without needing to rebuild the tree structures. On the other hand, if one were to attempt to perform these operations on a Newick file, the bipartitions from each tree will have to be extracted and the relationships between the set of trees will have to be discovered every single time. As tree collections grow large, this can pose a significant overhead. Lastly, we stress again that these set operations can all be performed on the input TRZ files without any loss of space savings. This is of critical interest, as it shows the viability of using the TRZ file as an alternative format for storing trees. With standard compression methods, the resulting binary file must always be decompressed in order for any type of manipulation on the data to be performed. As a result, these could not be considered as alternative formats to the Newick file. The TRZ file on the other hand is a viable format because set operations can be performed on it. Furthermore, since there is no loss of space savings, the TRZ file is a more efficient way of storing collections of trees.

2.4 Experimental Methodology

Our implementation of TreeZip used in the following experiments can be found at http://faculty.cse.tamu.edu/tlw/bmcbioinfo11. Experiments were conducted on a 2.5 Ghz Intel Core 2 quad-core machine with 4 GB of RAM running Ubuntu Linux 8.10. TreeZip is written in C++ and compiled with gcc 4.4.3 with the -O2 compiler option.

2.4.1 Biological trees

Below, we provide a description of the four biological tree collections used in this study. All of the weighted trees were obtained by running a Bayesian phylogenetic analysis using MrBayes [5]. The unweighted insects collection was derived from a Parsimony analysis using TNT [4]. For each dataset, the Newick tree file contains $t$ trees in the input file. All of the weighted collections we use for our experiments contain binary trees. The unweighted insects dataset, however, contains multifurcating trees. Thus, TreeZip can handle both binary and multifurcating tree collections.

1. freshwater: 20,000 trees obtained from an analysis of 150 taxa (23 desert taxa and 127 others from freshwater, marine, and oil habitats) [7]. The size of the Newick file for this tree collection is 67 MB. There are 1,318 unique bipartitions out of 5,960,000 total bipartitions.

2. angiosperms: 33,306 trees obtained from an analysis of 567 taxa (560 angiosperms, seven outgroups) [10]. The size of the Newick file for this tree collection is 429 MB. There are 3,011 unique bipartitions out of 37,702,392 total bipartitions.

3. fish: 90,002 trees obtained from an analysis 264 fish taxa (an unpublished collection from M. Glasner’s lab at Texas A&M University). Only binary trees are contained in this dataset. The size of the Newick file for this tree collection is 533 MB. There are 12,379 unique bipartitions out of 47,341,052 total bipartitions.
4. **insects**: 150,000 trees obtained from an analysis of 525 insect taxa [9]. The trees contained in this set are multifurcating. The size of the Newick file for this tree collection is 434 MB. There are 573 unique bipartitions out of 157,200,000 total bipartitions.

### 2.4.2 Measuring performance

We compare TreeZip to the 7zip compression algorithm. In our previous work [8], we found that 7zip is the most effective method for compressing phylogenetic trees amongst the standard compression methods (e.g. gzip, bz2). We measure the performance of our TreeZip algorithm in two primary ways: space savings and by using different, but equivalent Newick strings. Please note that 7zip here represents a Newick file compressed with the 7zip compression scheme.

**Space savings and running time.** We use the space savings measure to evaluate the performance of TreeZip in comparison to general-purpose compression algorithms. The space savings $S$ is calculated as $S = 1 - \frac{\text{compressed file size}}{\text{original file size}} \times 100$. A higher space savings percentage denotes better compression of the original file. The goal is to get the level of space savings as close to 100% as possible. A value of 0% indicates no difference from the uncompressed, original Newick file. We also use running time to calculate how long each algorithm requires to compress and decompress a file. Time is shown in seconds.

**Different, but equivalent Newick representations.** As mentioned previously, for any given tree of $n$ taxa, there are $O(2^{n-1})$ Newick string representations associated with it. Since general purpose compression methods such as 7zip compress tree files by looking for redundancy at the Newick string level, they are unable to efficiently compress trees when there is a lack of redundancy in the Newick string representations. To illustrate this, we created a different, but equivalent Newick file for each dataset. For a Newick file containing $t$ trees, each tree receives a different, but equivalent Newick representation. We note that using different, but equivalent Newick representations does not change the size of the resulting Newick file. For example, our fish dataset consisting of 90,002 trees over 264 taxa requires 533 MB of storage space. The Newick file containing different, but equivalent Newick strings still occupies 533 MB of disk space.

### 3 Results and Discussion

In this section, we explore the compression and decompression performance of 7zip, TreeZip, and TreeZip+7zip. Our previous results [8] show that 7zip is the best general-purpose compressor in comparison to gzip and bzip. The TreeZip+7zip compressed format is the TreeZip (TRZ) format which is then fed to 7zip for further compression. Moreover, our previous study showed that TreeZip outperforms TASPI. Since no implementation of TASPI is available and since none of the trees we had available that were used in the TASPI experiments had branch lengths, we could not compare TreeZip to TASPI in the context of this study.

Finally, each point in the plots represents the average performance over three runs.

#### 3.1 Compression performance

Figure 4 shows the performance of TreeZip’s compression algorithm. Figure 4(a) shows run-time information, and Figure 4(b) shows space savings results. On the freshwater and angiosperms datasets, we note that TreeZip is actually faster than 7zip. However, as the number of trees under consideration increases in size, so does the amount of time needed for compression. In terms of size, the TRZ file by itself is larger than the 7zip file. However, we obtain an average of 75% space savings on our weighted collections, and about 99% space savings on our unweighted collection. The discrepancy in space savings between the weighted and unweighted cases underlines the complexity of compressing branch lengths. However, we note that when the TRZ file is combined with 7zip, the TRZ+7zip file has space savings on average of about 96%. 7zip by itself, on the other hand, averages about 93% space savings.
Figure 4: Compression performance for our biological datasets. In this figure, (a) shows running time of compression approaches, while (b) shows space savings.

Effects of Branch Rotations on Compression performance. How robust are the different compression techniques under study to branch rotations? To measure the effects of branch rotations on our datasets, we took each set of trees and gave them a random, but equivalent Newick string representation. We refer to this process as commuting the Newick representation. Figure 5 shows the performance of the various compression schemes on different, but equivalent Newick string representations. The TRZ and TRZ+7zip files did not increase in file size. 7zip took up to 4.4 times longer on this new file. Figure 5(a) shows the change in space savings of the different compression schemes between the equivalent Newick files and the original files. Here, 100% of the Newick strings in the file have been commuted. A value of 1 signifies no change in file size. The space savings achieved by TreeZip and TreeZip+7zip does not change, despite the use of different, but equivalent Newick strings. This highlights TreeZip’s robustness to branch rotations. This is not the case for 7zip. On our weighted sets, the size of the 7zip compressed file became almost 4 times larger. On the unweighted (insects) set, the 7zip compressed file becomes 61 times bigger. This is equivalent of an increase of the size of the 7zip compressed Newick file from 696 KB to 38 MB. Figure 5(b) highlights the effect of these different, but equivalent Newick string representations on the efficacy of 7zip to compress the file. The x-axis indicates the percent of the original file that received commuted Newick string representations. For each percentile, p percent of the trees in the file contain a different, but equivalent Newick string representation. The 0% mark is the original Newick file. All the datasets have a universal value of 1 at this point, since there is no change in the compression quality. The 100% mark is equivalent to the files that were used in Figure 5(a). As one can see, as an increasing amount of the file is “randomized“ at the surface with different, but equivalent Newick string representations, the performance of 7zip becomes increasingly worse. Thus, while TreeZip’s compression approach is a little slower than 7zip, its advantages in terms of robustness to branch rotations and great levels of compression gives it serious advantages. This is especially true when considering the fact that operations can be performed directly on the TRZ file, without any loss of space savings.

3.2 Decompression performance

Figure 6 shows the decompression performance of 7zip and TreeZip-based decompressors. 7zip is a faster decompressor than the TreeZip-based approaches. For our two largest datasets (angiosperms and fish), 7zip is two orders of magnitude faster than TreeZip and TreeZip+7zip at decompression. This is mainly due to the fact that TreeZip decompresses trees in a serial manner. That said, the TreeZip is capable of decompressing subsets of trees of semantic interest to the user. The less the number of trees to be returned,
the faster the procedure. For example, we can return the set of unique trees in a particular file, the strict consensus tree, or the majority consensus tree. Of particular interest is the fact that the strict and majority consensus trees on our datasets can be produced in less than second on average. This is something that no other algorithm has been able to accomplish so far on these large datasets. TreeZip’s consensus operation is a 150 times faster than HashCS [11], the fastest algorithm for computing consensus trees. We note that HashCS has been found to be a 100 times faster than the consensus procedure in MrBayes [11]. We would also like to note that none of these operations can easily be performed on a 7zip file. Since the 7zip file is binary, all the trees would necessarily have to be decompressed before trees of semantic interest can be extracted, which can result in significant overhead. We explore this in greater detail in our set operations experiments, shown in Section 3.3.

3.3 Set operations performance

Figure 7 shows our performance results of set operations performed on Newick files, Newick files compressed with 7zip, TRZ files, and TRZ+7zip files. Figure 7(a) shows run-time results. On weighted trees, it is up to 3 times faster to perform set operations on TRZ files over Newick files. On the unweighted case, it is about 55 times faster. While there is little overhead in combining the TRZ file with 7zip, there is more significant overhead when combining Newick files with 7zip. While it is only at most 10 seconds slower to combine 7zip with a TRZ file for set operations, the overhead of combining 7zip with a Newick file is as much as a minute. As a result, the speedup results are more significant when comparing set operations on a TRZ+7zip file versus a Newick+7zip file. On weighted trees, the speed up is at most 5.25. On unweighted trees, the speedup is up to 62.4. We note that the vast differences in speedup between the unweighted and weighted cases has to do with the fact that TreeZip has to do a lot of extra processing in order to handle branch lengths. Since 7zip does not view branch lengths as being any different from raw text, there is no additional processing needed.

Figure 7(b) shows the average space savings of storing the results of set operation in a TRZ file versus a Newick file, TRZ+7zip file and a 7zip file. In terms of size, the results of set operations are more efficiently stored in TRZ files than Newick files. On weighted trees, the TRZ file storing the results of the set operations is
Figure 6: Decompression performance for our biological datasets.

Figure 7: Performance of set operations on our biological datasets. (a) The running time of a random collection of set operations run on different file formats. (b) The amount of disk space required by the result of the set operations.

74.1% smaller than the Newick file. On unweighted case, it is up to 99.7% smaller. This is very consistent with the general space savings of using a TRZ file over a Newick file on weighted and unweighted trees respectively. TRZ+7zip files have at most a 21% improvement in space savings over Newick+7zip files in the weighted case. On the unweighted case, the TRZ+7zip takes up 78.75% less space than the Newick+7zip file. The TRZ+7zip file produced up to 5 times faster than the Newick file compressed with 7zip, takes up less space. There are no space savings using a Newick file, and we reiterate that the files compressed with 7zip have to be decompressed entirely before set operations can be performed on them. This is another reason why set operations on files compressed with 7zip are slower than those compressed with TreeZip+7zip. Together, these results underline the benefit of using the TRZ file for set operations versus using a Newick file or a file compressed with 7zip.
4 Conclusions

There is a critical need for phylogenetic compression techniques that reduce the space requirements of large tree collections. Phylogenetic searches typically produce tens to hundreds of thousands of candidate evolutionary trees. Furthermore, when these techniques are used to reconstruct larger trees, even larger number of candidate trees are produced. In this paper, we present a modified version of our TreeZip algorithm [8], which incorporates several important extensions. First, the TreeZip algorithm is augmented to allow for the compression of branch lengths. Second, we offer an extensible decompresser which allows for filtering and extraction of sets of trees of interest. Lastly, we offer functionality that allows for extremely fast set operations directly on the TRZ file without a loss of space savings.

By itself, TreeZip compresses a Newick file into a plain text compressed representation that is at least 73% smaller than the original file on the weighted case, and over 98% smaller on the unweighted case. When combined with 7zip, the TreeZip+7zip file achieves an average space savings of 92% on the weighted case, and space savings of over 99% on the unweighted case. This makes it the best method to compress large collections of phylogenetic trees. Furthermore, in cases of high degrees of bipartition sharing, our TreeZip algorithm is able to compress a collection of trees faster than 7zip. Lastly, TreeZip is robust to different representations of a phylogenetic tree allowing the TreeZip and TreeZip+7zip compressed formats to achieve consistently good space savings. The space savings achieved by 7zip, on the other hand, decreases as the number of different Newick representations for the same phylogenetic tree increases.

TreeZip’s most powerful advantage arises from the TRZ file’s textual format. This allows the TRZ file to be used as input to phylogenetic applications without a need to rebuild all the trees into Newick format. Given the flexibility of the TRZ format, we can easily design extensible decompressers that extract the relevant phylogenetic tree information of interest. Our results illustrate two decompresser applications—removing duplicate trees and creating consensus trees—that can be created to extract information from the TRZ file. We show that due to the textual form of the TRZ file, set operations can be performed up to five times faster than on a Newick file and store results in as much as 99% less space. This is not functionality that standard compression methods such as 7zip can achieve since they cannot leverage the phylogenetic information contained within the Newick representation of a phylogenetic tree.

Overall, our results provide evidence that TreeZip has the potential for playing a vital role in how biologists manage their phylogenetic tree collections. In the future, we plan to augment the decompresser with additional functionalities. Finally, we plan on adding additional optimizations to improve running time of our compression and decompression routines.

Acknowledgements

Funding for this project was supported by the National Science Foundation under grants DEB-0629849, IIS-0713168, and IIS-1018785. Moreover, this publication is based in part on work supported by Award No. KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST).
References

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

[2] R. S. Boyer, W. A. Hunt Jr., and S. Nelesen. A compressed format for collections of phylogenetic trees and improved consensus performance. In Proc. 5th Int’l Workshop Algorithms in Bioinformatics (WABI’05), volume 3692 of Lecture Notes in Computer Science, pages 353–364. Springer-Verlag, 2005.

[3] J. Felsenstein. The Newick tree format. Internet Website, last accessed, September 2009. Newick URL: http://evolution.genetics.washington.edu/phylip/newicktree.html.

[4] P. A. Goloboff, J. S. Farris, and K. C. Nixon. TNT, a free program for phylogenetic analysis. Cladistics, 24(5):774–786, 2008.

[5] J. P. Huelsenbeck and F. Ronquist. MRBAYES: Bayesian inference of phylogenetic trees. Bioinformatics, 17(8):754–755, 2001.

[6] W. A. Hunt Jr. and S. M. Nelesen. Phylogenetic trees in ACL2. In Proc. 6th Int’l Conf. on ACL2 Theorem Prover and its Applications (ACL2’06), pages 99–102, New York, NY, USA, 2006. ACM.

[7] L. A. Lewis and P. O. Lewis. Unearthing the molecular phylodiversity of desert soil green algae (chlorophyta). Syst. Bio., 54(6):936–947, 2005.

[8] S. J. Matthews, S.-J. Sul, and T. L. Williams. A novel approach for compressing phylogenetic trees. In Bioinformatics Research and Applications, volume 6053 of Lecture Notes in Computer Science, pages 113–124. Springer-Verlag, 2010.

[9] A. D. Molin, S. Matthews, S.-J. Sul, J. Munro, J. B. Woolley, J. M. Heraty, and T. L. Williams. 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.

[10] D. E. Soltis, M. A. Gitzendanner, and P. S. Soltis. A 567-taxon data set for angiosperms: The challenges posed by bayesian analyses of large data sets. Int. J. Plant Sci., 168(2):137–157, 2007.

[11] S.-J. Sul and T. L. Williams. An experimental analysis of consensus tree algorithms for large-scale tree collections. In Proceedings of the 5th International Symposium on Bioinformatics Research and Applications (ISBRA’09), pages 100–111, Berlin, Heidelberg, 2009. Springer-Verlag.

[12] H. E. Williams and J. Zobel. Compressing integers for fast file access. The Computer Journal, 42:193–201, 1999.