universität wien

MASTERARBEIT / MASTER’S THESIS

Titel der Masterarbeit / Title of the Master‘s Thesis

„TreeShredder: A Program for Phylogenetic Analysis of Large Sets of Trees Based on Splits“

verfasst von / submitted by

Clement Bader, BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2023 / Vienna 2023

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 066 875
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Bioinformatik
Betreut von / Supervisor:	Univ.-Prof. Dr. Arndt von Haeseler
Mitbetreut von / Co-Supervisor:	Dr. Heiko A. Schmidt

Acknowledgements

I would like to thank Heiko A. Schmidt for supporting my journey through the trees, Arndt von Haeseler for his feedback, and Olga Chernomor for providing the Gentrius multi-gene trees. Furthermore, I thank Marc Zobel and Vincent Reiner for the initial Splits Extraction and Tree Creation Algorithms they have developed for a predecessor version of TreeShredder.

Vor allem danke ich meiner Familie. Sie ist das Wichtigste in meinem Leben.

Abstract

TreeShredder is a parallelized multi-tool software for the phylogenetic analysis of large sets of trees based on their splits. TreeShredder can deal with datasets of millions of trees with thousands of taxa. It offers well-established analysis approaches and extends them by additional, more recently introduced features. Many time-consuming procedures, such as parsing huge numbers of Newick tree strings, or calculating Transfer Bootstrap Expectation values, are parallelized. A space and time-saving file format for split and tree information storage and retrieval, the TreeShredder file, is introduced and its advantages are demonstrated. TreeShredder implements a matrix representation feature for supertree construction, which has seen discontinued maintenance and support of capable software in recent years. TreeShredder offers comprehensive reference tree and consensus tree features, including the newly introduced global relative majority consensus tree. Additionally, the user can map eight different split measures, including occurrence rates, Internode Certainty, or Transfer Bootstrap Expectation, but also newly developed measures such as a split's best incompatible split's support and the difference in their support, onto the reference and consensus trees. Unique among competitor software in the field, TreeShredder can find based on a set of splits or even incomplete spits congruent trees, determine the congruency status with the splits in the trees, and calculate congruency measures. Besides the well-established Robinson-Foulds distances, which show similarity between tree topologies, a new measure, called Split Co-Occurrence is introduced, which shows how often two splits co-occur in the same tree. Space-saving output compression comes without runtime increase. TreeShredder's performance compares favourably against similar features offered in software such as RAxML and BOOSTER, especially, but not exclusively, if the phylogenetic analysis is started from TreeShredder files. By means of diverse datasets of trees ranging in size from tens to thousands of taxa, I show that TreeShredder is a valuable and versatile addition to the phylogenetic analysis toolbox.

Zusammenfassung

TreeShredder ist ein paralleles Multifunktionsprogramm für die phylogenetische Analyse großer Gruppen von Bäumen basierend auf ihren Splits. Es bietet bewährte Analysewerkzeuge und erweitert diese um zusätzliche, neuere Funktionen. Viele zeitintensive Funktionen, wie das Parsen einer großen Anzahl von Newick-Tree-Strings und das Berechnen von Transfer Bootstrap Expectation-Werten, können parallel ausgeführt werden. Ein platz- und zeitsparendes Dateiformat zur Speicherung und Wiederabrufung von Split- und Baum-Information, die TreeShredder-Datei, wird vorgestellt und ihre Vorteile demonstriert. TreeShredder führt eine Matrix-Representation-Funktion für die Supertree-Konstruktion wieder ein, die in den letzten Jahren nicht mehr gewartet und unterstützt wurde. TreeShredder bietet Referenz- und umfangreiche Konsensus-Baum-Funktionen, einschließlich des neu eingeführten globalen relativen Mehrheits-Konsensus-Baumes. Zusätzlich können User acht verschiedene Split-Maße, einschließlich Häufigkeitsraten, Internode Certainty, oder Transfer Bootstrap Expectation, aber auch die neu entwickelten Maße der Unterstützung des besten inkompatiblen Splits und die Differenz zur Unterstützung des besten inkompatiblen Splits auf Referenz- und Konsensus-Bäume projizieren. Einzigartig unter vergleichbarer Software findet TreeShredder kongruente Bäume und berechnet Kongruenz-Maße für eine Gruppe von vollständigen oder sogar unvollständigen Splits, als auch Kongruenz-Status der Splits der Bäume. Zusammen mit der Robinson-Foulds-Distanzen-Funktion, die Ähnlichkeiten zwischen Baumtopologien anzeigt, wird ein neues Maß namens Split Co-Occurrence eingeführt, das zeigt, wie oft zwei Splits gleichzeitig im selben Baum auftreten. Platzsparende Ausgabekomprimierung bleibt ohne Laufzeiterhöhung. TreeShredder schneidet im Vergleich mit RAxML und BOOSTER positiv ab, insbesondere, aber nicht nur, wenn die phylogenetische Analyse von einer TreeShredder-Datei gestartet wird. Anhand von diversen Gruppen von Bäumen, die dutzende bis tausende Taxa groß sind, zeige ich, dass TreeShredder eine wertvolle und versatile Erweiterung des Repertoires der phylogenetischen Analyse ist.

Contents

List of Figures

List of Tables

Part I

Introduction and Materials

1 Introduction

In the field of phylogenetics, a typical aim of researchers is to find the optimal phylogenetic tree that depicts the evolutionary descent of species, organisms, or genes from a shared ancestor in a single graph. Fig. 1.1 shows such graphs representing example phylogenetic trees.

A phylogenetic tree consists of a set of nodes which are connected by a set of edges. Nodes that are connected to only one edge have degree $= 1$ and are called leaf nodes. They are depicted as squares in Fig. 1.1 and represent phylogenetic data such as gene sequences, species, or morphological properties. Labeled leaf nodes are also called taxa.

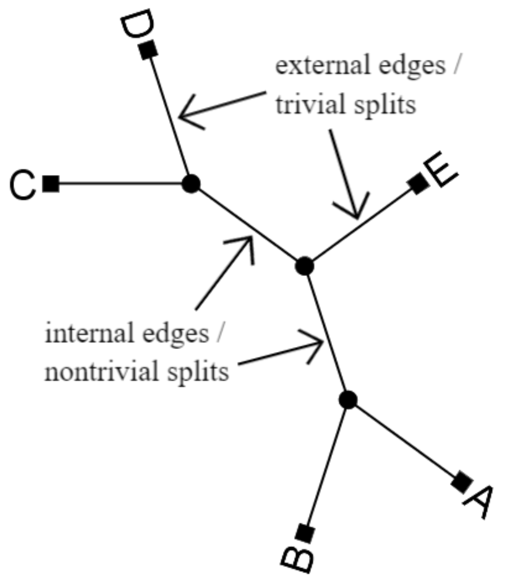
Nodes that are connected to two or more edges have degree ≥ 2 and are called internal nodes. They are depicted as circles in Fig. 1.1 and join subtrees of phylogenetic trees. Internal nodes are typically unlabeled.

Rooted trees imply a direction from the root node along the edges toward the leaf nodes, representing a (typically chronological) starting point at the root and an endpoint at the leaf nodes (see Fig. 1.1b for an example of a rooted tree). By convention, a root node is an internal node of degree $= 2$, joining two root edges. Conversely, unrooted trees depict the relationship between subtrees without implying a starting point (see Fig. 1.1a for an example of an unrooted tree).

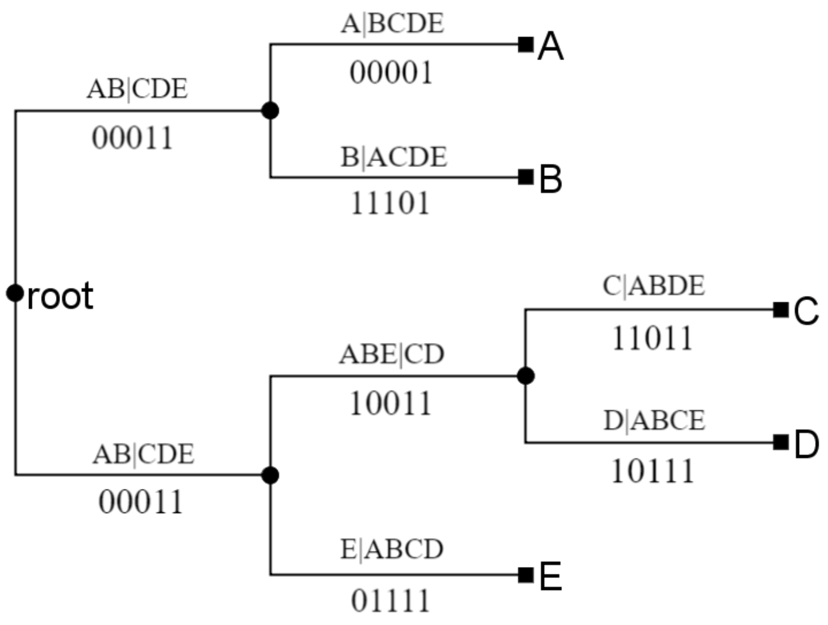
In phylogenetics, tree data are usually not represented as separate sets of nodes and edges but in Newick tree format (Felsenstein et al., 1986; Page, 2003). This text format is a string of characters which can transfer the relationship between the nodes of a tree and additional information such as the names of the nodes, the branch length of the edges, frequency data, or comments. An internal node is represented by a pair of parentheses which contain other nodes that are connected to them by edges. The nodes connected to an internal node are separated by commas. Unlike internal nodes, leaf nodes are not enclosed by parentheses. For example, the Newick tree string "`((A,B),(C,D),E);`" represents the unrooted tree in Fig. 1.1. Here, the outermost pair of parentheses contains three nodes separated by commas: the leaf node "`E`" and the two internal nodes "`(A,B)`" and "`(C,D)`". The two internal nodes contain the leaf nodes "`A`" and "`B`", and "`C`" and "`D`", respectively.

A tree connects each taxon to every other taxon via a unique path of contiguous nodes and edges (Semple and Steel, 2003, p. 7) – there is no possibility to travel along the edges of a tree and return to the starting point without passing the same edge at least twice. This means that if a single edge were removed, the set of taxa would be split into two disjoint subsets. Doing so causes a bipartition, better known as a split, on the set of taxa. Importantly, a tree is uniquely defined by its set of splits.

This split can be represented in "taxon notation" or "bit notation" format (see also section 7.1). For example, the split that splits the taxon set $\{A, B, C, D, E\}$ in



(a) An unrooted tree with Newick string "`((A,B),(C,D),E);`".



(b) A rooted tree with Newick string "`((A,B),((C,D),E));`".

Figure 1.1: Examples of an unrooted and a rooted tree: Round shapes indicate internal nodes (including the root node), square shapes indicate external nodes, i.e., taxa. Internal edges connect internal nodes, external edges connect external nodes with internal ones. Removing an edge induces a bipartition on the taxon set, a split. The bits in the "bit notation" format correspond to the taxa in reverse order.

Fig. 1.1b into the subsets $\{A, B, E\}$ and $\{C, D\}$ can be represented as `ABE|CD` or `10011`. The vertical line '`|`' shows the separation of the two taxon subsets in the "taxon notation" format. In the example in Fig. 1.1a, I imposed lexicographical order within the taxon subsets. In the "bit notation" format, the bits represent the taxa in reverse order because TreeShredder's data structure prints the least significant bit rightmost. This means that the bits, read from left to right, stand for E, D, C, B, and A. Taxa with bit value $= 0$ belong to one side of the split and those with bit value $= 1$ belong to the other.

If the smaller side of the split contains only one taxon, it is called a trivial split because it must necessarily occur in all trees with identical taxon sets, representing the external edges connecting a leaf node to the rest of the tree (see Fig. 1.1a). If the smaller side contains more than one taxon, it is a nontrivial split. Trees with identical taxa but different topologies (i.e., branching order of subtrees) have different sets of nontrivial splits. This is where the diversity of trees with identical taxon sets arises.

A tree with n taxa has n trivial splits, one for each external edge connecting a leaf node to the rest of the tree. A binary unrooted tree has $n-3$ nontrivial splits, one for each internal edge. This is evident from the fact that by inserting an additional leaf node (along with its external edge) into an unrooted binary tree, either an internal edge is split into two internal edges or an external edge is split into one internal and one external edge. In both cases, by adding a leaf node, the number of internal edges grows by 1 for trees with ≥ 3 taxa (Felsenstein, 1978). Taken together, the number of splits in an unrooted tree grows linearly by $2n - 3$ with the number of taxa.

Because of the many possibilities to connect the same set of taxa in a different way and build different trees, typically thousands to millions of trees are generated and evaluated in a phylogenetic reconstruction searching for the optimal tree. Due to the super-exponential growth of combinations to connect the taxa, the number of unrooted binary trees grows according to

$$(2n-5)!! = \frac{(2n-5)!}{2^{n-3}(n-3)!}$$

for $n \geq 3$, where n is the number of taxa (see Felsenstein, 1978; Penny, Hendy, and Holland, 2007, p. 502). For example, there are only 15 different unrooted trees for the taxon set $\{A, B, C, D, E\}$. These trees could easily be generated and evaluated individually. However, the number quickly grows prohibitively large for larger trees: there are more than $2 * 10^{20}$ different trees with 20 taxa.

Unfortunately, there are no efficient algorithms to overcome the unwieldy large number of possible trees and find the optimal maximum likelihood (Chor and Tuller, 2005) or maximum parsimony (Day, Johnson, and Sankoff, 1986) tree. Because of the potentially huge number of different trees to take into consideration, combined with the fact that there are no algorithms that guarantee to find the optimal tree in polynomial time (Penny, Hendy, and Holland, 2007, p. 502), researches cannot be sure to find the one optimal tree even after extensive search. To assess the quality of trees, they often resample the multiple sequence alignment to generate several trees, which can be used to calculate support values of subtrees or to create consensus trees. For example, bootstrapping or jackknifing are two such resampling strategies to determine general robustness of data estimates, which are prerequisites for a credible tree inference analysis and well-established in the scientific community (Felsenstein, 1985; Felsenstein, 1986). These trees can then be used for further analysis, strengthening robustness of the tree inference. However, presenting the data of several thousand different trees in a single, maximally informative tree is not trivial.

For this master's project, I designed and implemented the software TreeShredder that can deal with large sets of millions of trees with thousands of taxa and measure its performance in diverse applications. Additionally, I compare it in calculating and mapping Transfer Bootstrap Expectation values onto a reference tree, and in calculating Robinson-Foulds Distances against RAxML (Stamatakis, 2006) and BOOSTER (Lemoine et al., 2018), which are established programs for phylogenetic analysis.

1.1 Overview and Structure of the Thesis

This thesis is subdivided into four parts. Each part contains chapters that deal with related features of TreeShredder.

Part I, Introduction and Materials (the current part), introduces the reader to basic concepts in phylogenetics and the datasets and computational resources used to demonstrate TreeShredder's performance throughout this thesis.

Part II, Software and Optimization, is more technical and explains the program components and data structures of TreeShredder. It also describes how they work together in a typical TreeShredder call:

- Treeshredder's program structure is explained in chapter 3.
- The data structure used to represent splits and the challenges for parallelization are described in chapter 4.
- The first step in a typical TreeShredder workflow is obtaining splits. This is done by extracting them from Newick strings or reading them from a TreeShredder file, which is designed to speed up split information storage and retrieval (see chapter 5).

Part III, Methods in TreeShredder, introduces the methods implemented for TreeShredder's diverse applications in detail. Additionally, it presents the runtime and memory storage results of these applications:

- TreeShredder can map split information onto a reference tree (see chapter 6), or
- find trees in a set that are congruent with incomplete splits (see chapter 7), which are splits where some, ambiguous taxa may occur on either side of the split.

- Additionally, TreeShredder can create a variety of consensus trees (see chapter 8). This allows the researcher to depict the tree information from a set of (potentially conflicting) trees that have identical taxa in a single tree.
- However, if the taxon sets of the trees differ, a matrix representation for supertree construction is needed (see chapter 9).
- TreeShredder can also calculate the similarity of two trees using the Robinson-Foulds distance (see chapter 10), and
- give a measure of the overlapping occurrence of two splits in the trees, called Split Co-Occurrence (see chapter 11).

Part IV, Summary, is the overall conclusion of this thesis and provides an outlook for additional methods that could be implemented in TreeShredder in the future.

2 Materials and Resources

In the following I describe the datasets and computational resources I used to measure TreeShredder's performance in benchmarks as well as comparisons against other software.

2.1 Datasets

In this thesis, I gauge TreeShredder performance and compare it against RAxML and BOOSTER using eleven different tree datasets. For better readability I refer to the different datasets by a three-letter prefix followed by the number of taxa in that dataset. The eleven datasets are: Wnt (WNT-82), CRF01 (CRF-2696), pol_nonrecombinant (POL-9147), aco1 gene (ACO-225), PTPN13 gene (PTP-116), Pfam Domain PF01546 (PF1-367), Pfam Domain PF07690 (PF7-205), and Gentrius (GEN-404).

WNT-82 are 1,000 animal trees with 82 taxa computed with the MrBayes software (Lengfeld et al., 2009).

CRF-2696 are 1,000 HIV Ultrafast Bootstrap trees (UFBoot; Minh, M. A. T. Nguyen, and Haeseler, 2013) with 2,696 taxa and were provided by Dimitrios Paraskevis (Department of Hygiene Epidemiology and Medical Statistics, National and Kapodistrian University of Athens).

POL-9147 are 1,000 HIV-1 group M UFBoot trees with 9,147 taxa (Lemoine et al., 2018).

ACO-225 are 100,000 aco1 gene UFBoot trees with 225 taxa (Reddy et al., 2017).

PTP-116 are 100,000 PTPN13 gene UFBoot trees with 116 taxa from the OrthoMaM database (Ranwez et al., 2007).

PF1-367 are 100,000 Pfam Domain PF01546 UFBoot trees with 367 taxa from the PANDIT database (Whelan et al., 2006).

PF7-205 are 100,000 Pfam Domain PF07690 UFBoot trees with 205 taxa from the PANDIT database (Whelan et al., 2006).

GEN-404 are about 15 million multi-gene trees with 404 taxa forming a terrace (Sanderson, McMahon, and Steel, 2011) of equal likelihood where genes are missing. This dataset was created with the Gentrius software, which produces such terraces for multi-gene alignments. Olga Chernomor (CIBIV, University of Vienna), the developer of Gentrius, kindly provided the dataset.

RAN-200, **RAN-300**, and **RAN-400** contain 1 million random trees with 200, 300, and 400 taxa, respectively. They were created with the GenerateTrees tool (Schmidt, 2007).

2.2 Computational Resources and Benchmarks

The benchmarks on the Linux operating system were performed at the Center for Integrative Bioinformatics Vienna (CIBIV) on an AMD Ryzen 7 1700X Eight-Core Processor with 32 GB memory and 16 virtual processors allowing for hyper-threading. Parallel speedup benchmarks were done on an AMD EPYC 7501 32-Core Processor with 2.1 TB memory and 128 virtual processors. The benchmarks on the Windows operating system were performed on an Intel Core i5-8250U CPU with 4 cores, 8 threads, and 8 GB memory.

To avoid network traffic delays, input and output data were kept in local storage. Data points represent the median value of 5 repetitions.

Part II

Software and Optimization

3 Program Structure

The aim of this chapter is to give the user a better understanding of TreeShredder's components and how they work together during execution. TreeShredder is a C++ program and besides the obligatory `main()` function, it comprises four major classes, namely `Split`, `NewickParser`, `DataDepot`, and `CommandlineParser`. In addition, the libraries Nexus Class Library (NCL; Lewis, 2003), zlib (Gailly and Adler, 2022), `boost::dynamic_bitset` (Siek and Allison, 2017), and utility functions have a supportive role.

Fig. 3.1 shows the program structure of TreeShredder as a Unified Modeling Language (UML) diagram (Object Management Group, 2017). The `main()` function initializes a `CommandlineParser` object and then calls the `execute()` method which executes the tasks determined by the command-line flags in the correct order. Some of these flags are processed solely in the `CommandlineParser` object, but most require a `DataDepot` object. This object has C++ standard containers for trivial, nontrivial and root splits, as well as information about which split is contained in which tree. To parse Newick tree strings the `DataDepot` object uses a `NewickParser` object which returns extracted split information and creates `Split` objects which are stored in the split containers.

The Nexus Class Library (NCL) parses Nexus format files, zlib compresses and decompresses file output and input, and `boost::dynamic_bitset` represents bipartitions on taxon sets (i.e., splits; see chapter 4). These libraries are used by the respective classes they are attached to in Fig. 3.1. The util file contains utility functions that are used by all classes of TreeShredder and macros that allow the user to quickly change settings that affect the whole program, e.g., output decimal precision of number values and metacomment key names.

Because the creation of `dynamic_bitset`s is computationally expensive, copying `Split` objects is kept to a bare minimum. Additionally, to maintain conceptional simplicity, identical `Split`s only exist once in the program. To achieve this, pointers offered by the C++ memory library, chiefly `std::shared_ptr`, are used to keep track of created `Split`s, avoid unnecessary copying, and prevent memory leakage as the destruction of heap objects is automatically managed by the memory library.

3.1 Program Components

3.1.1 `main()` function

This is the entry point of the TreeShredder program. It has a try-catch block which tries to create an instance of `CommandlineParser`, and then tries to execute it using the `execute()` method. If an exception was not caught and dealt with further

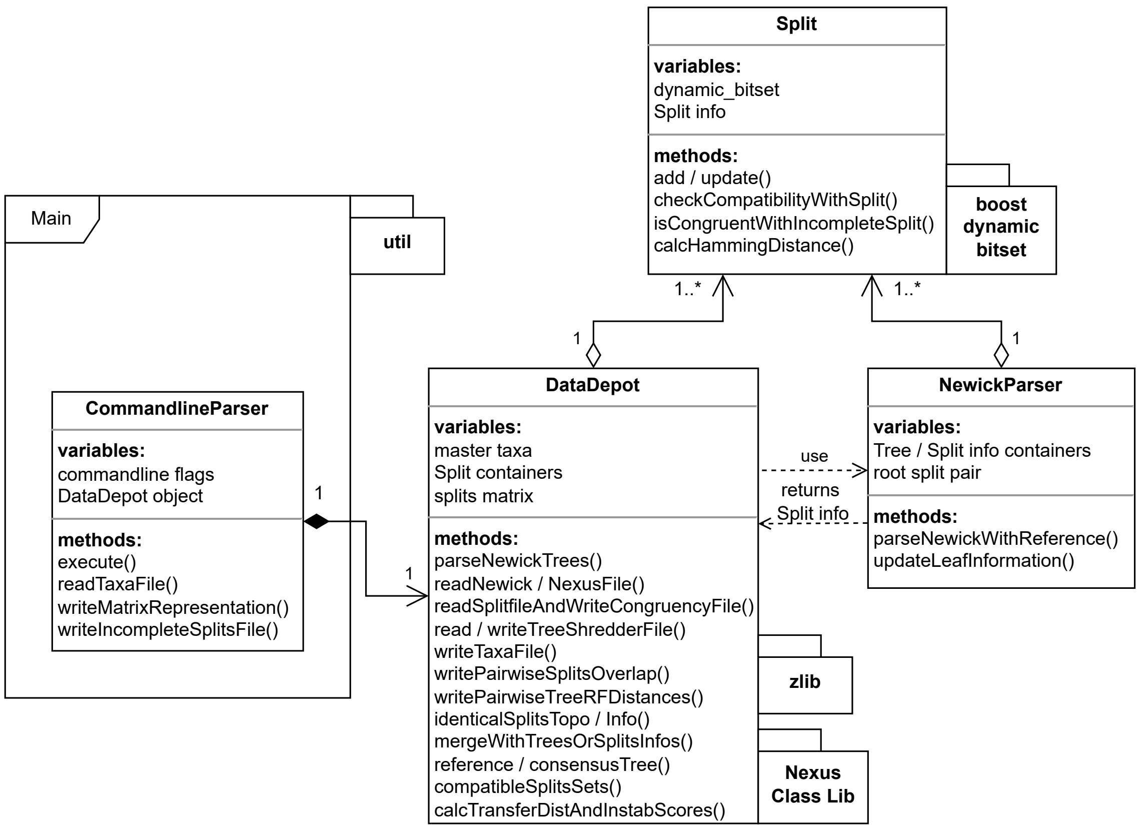


Figure 3.1: UML program structure of TreeShredder: For better clarity, only the main variables and methods of `CommandlineParser`, `DataDepot`, `NewickParser` and `Split` classes are listed. The zlib, Nexus Class Library and `boost::dynamic_bitset` libraries are attached to the classes they are used in. The utility functions are used in all classes, for visualization purposes the util file is attached to the `main()` function. The arrow with the full diamond shape connecting the `CommandlineParser` with the `DataDepot` class indicates a “composition” relationship. This is because the `CommandlineParser` object initializes one `DataDepot` object which is destroyed if the `CommandlineParser` is destroyed. In TreeShredder, a `DataDepot` object exists only within a `CommandlineParser` object. The dashed arrows connecting the `DataDepot` class with the `NewickParser` class indicate that the former uses the latter (when parsing Newick strings), and the latter returns split information. The arrows with the empty diamond shapes connecting the `DataDepot` and `NewickParser` with the `Split` class indicate an “aggregation” relationship. This is because the `Split` objects are not necessarily destroyed when either of the former two are.

downstream in the program logic, it will be caught here and the program will be terminated with an appropriate exit code and exception message.

3.1.2 `CommandlineParser` class

The `CommandlineParser` class initializes a `DataDepot` object, stores the command-line flags that were passed in the TreeShredder program call, and then executes their corresponding methods in the correct order. Some flags, in particular those that do not need the help of a `NewickParser` or create `Splits`, will be executed by methods of the `CommandlineParser`. Most flags, however, are processed by the `DataDepot` class.

3.1.3 `DataDepot` class

When `DataDepot` reads Newick or Nexus files it uses `NewickParser` to parse their Newick strings. The `NewickParser` then returns the extracted split information, which is stored by the `DataDepot` containers as key:value pairs with unique keys. As a consequence, the information of identical splits is condensed into one unique `Split` object, which is then stored in one of three different containers depending on whether it is a trivial, nontrivial, or root split.

Additionally, *pointers* to the unique nontrivial `Split` objects are stored in a vector. This makes it possible to retrieve the order in which splits were created when writing a TreeShredder file (see section 5.5).

The information about which split is contained in which tree is stored in a vector of vectors of *pointers* to the `Split` objects. Here, the inner vectors represent the pointers to the nontrivial splits of the trees, which are held by the outer vector in the order these trees were parsed. This information is important for, e.g., Transfer Bootstrap Expectation (see subsection 6.1.5), which requires the original relationship of trees to their nontrivial splits.

The taxon names are stored in a vector of strings.

The original Newick string is stored to be retrieved by the reference tree routine (see section 6.2.2).

To keep the creations of `dynamic_bitsets` to a minimum, the splits matrix (a vector of `dynamic_bitsets`) belongs to the `DataDepot` but is passed by reference to the `NewickParser` when it parses a Newick string.

3.1.4 `NewickParser` class

The `NewickParser` class parses a Newick tree string and extracts node and edge information and maps it to the correct split using Marc Zobel's Splits Extraction Algorithm (see section 5.2).

To minimize the number of `dynamic_bitset` creations, two mitigation strategies are employed:

- The splits matrix (a vector of `dynamic_bitset`) for Marc Zobel's Splits Extraction Algorithm is reused such that it needs to be created only once (as opposed to as many times as there are trees). For this purpose, the splits matrix belongs to the `DataDepot` but is passed to the `NewickParser` by reference each time a Newick tree is parsed.

- Trivial and nontrivial `Split` objects are only created once they are known to be new and unique. This is done because in a set of similar trees, as is common in phylogenetic datasets, the number of *unique* splits is typically much smaller than the total number of splits, which is $t(2n-3)$ for t bifurcating trees with n taxa. Fig. 3.2a shows that the number of unique nontrivial splits saturates with increasing number of trees in biological datasets. Conversely, the number of unique nontrivial splits saturates much more slowly in the datasets with trees with randomized topology (see Fig. 3.2b).

3.1.5 `Split` class

TreeShredder is based on splits, which are bipartitions on the set of taxa of the trees. The basic operational unit of TreeShredder are objects of the `Split` class. A `Split` object has a `dynamic_bitset` to represent its bipartition on the set of taxa. Since there are two ways to represent the *same* split (two flipped equivalents: e.g., 00001 and 11110), by convention, the first taxon is defined to be equal to 1 to be able to distinguish between the two and, thus, guarantee uniqueness of the splits in the program. The fact that the same split can be represented in two ways can help speed up certain routines with bitwise operations (e.g., in Transfer Bootstrap Expectation calculation; see subsection 6.1.5). Therefore, at the cost of memory, a second `dynamic_bitset` is stored in every `Split` object, which is the flipped equivalent of the first.

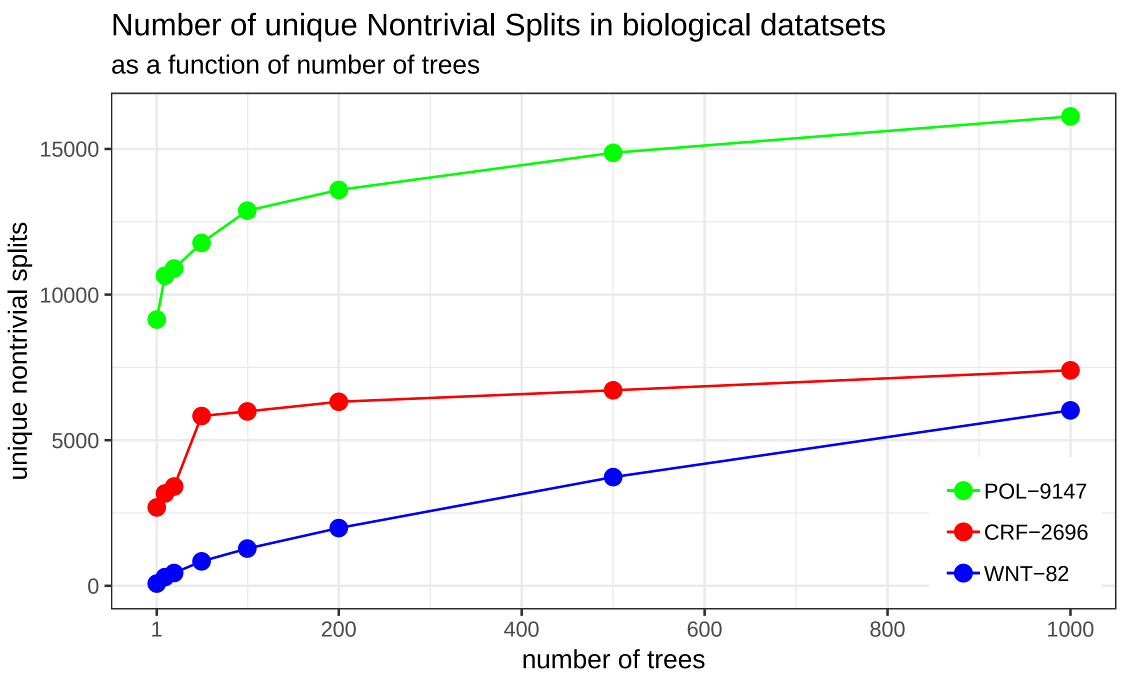
Additionally, the total number of taxa, the number of taxa on the smaller side of the split, the support of the split (i.e., how often the split occurs in all trees), and the sum of branch lengths (from which the average can be calculated) is stored.

Edge and node comments are stored as a vector of strings. Nexus edge and node metacomments (BEAST developers, 2017) are stored as key:string or key:double pairs. When parsing Newick trees, node (meta)comments are interpreted as belonging to the node that is farther from the root of the Newick string of the tree. Therefore, when gathering *node* information of identical splits, TreeShredder's reliance on bipartitions as the representation of splits prohibits knowing to which of the two adjacent nodes of an edge the node information belongs. This is because the directionality of parsing is determined by the placement of the root, which may differ in respect to the split in question. This may render the node (meta)comment information useless if that directionality is not preserved, which is probable when parsing multiple trees (see subsections 8.2.1 and 6.2.2). These restrictions do not apply to edge (meta)comments.

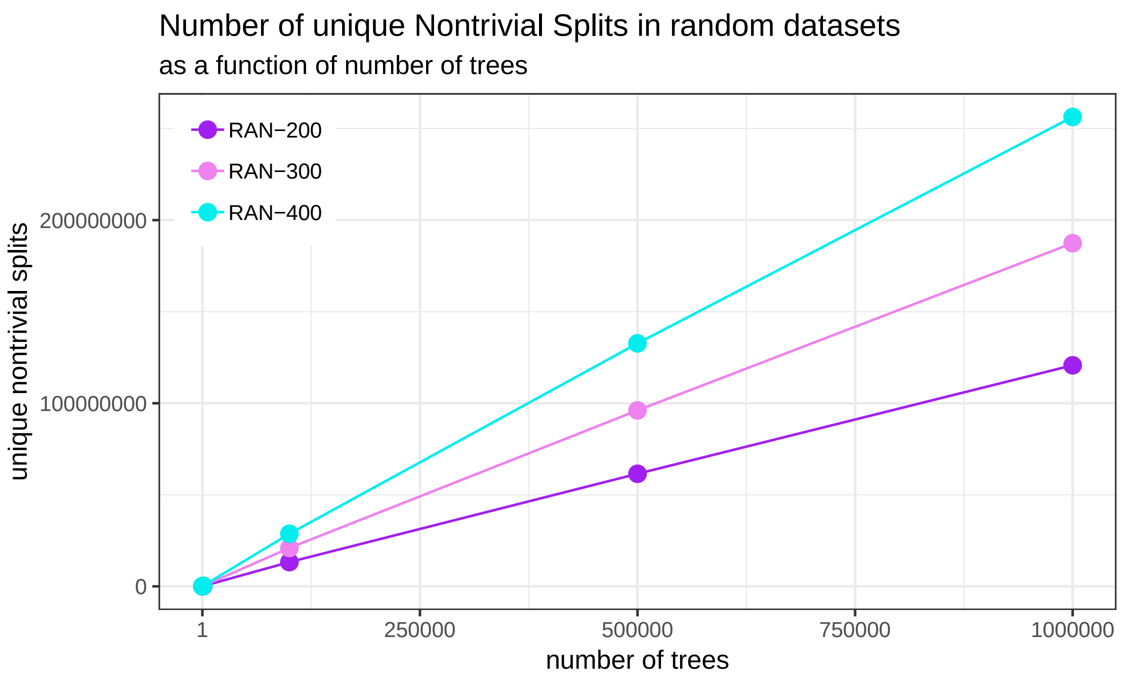
Newly created `Split` objects are given an ID when they are known to be unique.

3.1.6 util.cpp file and macros

The util.cpp file contains functions that are used by the classes but are not particular to either of them. Additionally, it defines macros that can be changed to customize



(a) Number of unique nontrivial splits in biological datasets: The number of unique nontrivial splits of CRF-2696 and POL-9147 increases logarithmically with the number of trees – it saturates quickly. WNT-82 saturates more slowly. Apparently, WNT-82 trees are more diverse.



(b) Number of unique nontrivial splits in random datasets: The number of unique nontrivial splits of RAN-200, RAN-300, and RAN-400 increases roughly linearly with the number of trees – it saturates slowly.

Figure 3.2: Number of unique nontrivial splits in biological and random datasets

TreeShredder to the user's preferences.

The most important macro is the index offset. Setting the index offset macro to 0 (which is the default) forces TreeShredder output indices to start at 0. Analogously, setting it to 1 forces indices to start at 1. Internally, TreeShredder works with 0 based indices. Thus, if it reads in indices, e.g., when reading the "Trees" block of TreeShredder files (see section 5.5), the index offset will be subtracted. This will cause problems when the output of one TreeShredder instance is read in by another instance with a different index offset. Therefore, I advise the TreeShredder community to decide on an index offset value and stick to it to facilitate sharing of output produced by TreeShredder. When TreeShredder is executed, it will output the index offset. A good practice is keeping log files for later reference.

The fraction notation macro can be used to customize the output of fractions. If it is set to 1.0 (which is the default), then fractions of certain split measures (see section 6.1) will be output as a fraction. However, choosing a different value, e.g., 100.0, will output these split measures as percentages. The same caveat applies as with the index offset macro when differently customized TreeShredder instances read each other's output. Note that the fraction notation macro must be a float, otherwise TreeShredder could cut off decimal places due to integer divisions in C++.

The branch length macro sets the default branch length of branches. TreeShredder assumes this branch length for branches that have no length assigned. Per default, this macro is 1.0.

The user can customize the number of decimal places displayed in TreeShredder float values output by changing the decimal places macro. TreeShredder will round the values, not merely cut off decimal places.

Last but not least, the user can customize the names of the keys that TreeShredder uses for its Nexus metacomment output (e.g., split measures) by customizing the metacomment macros. These names are reserved in TreeShredder and cannot be used for other keys in metacomments.

3.1.7 Libraries and other Accessories

There are four libraries used in TreeShredder (see Fig. 3.1):

The zlib library (Gailly and Adler, 2022): This is used to read compressed input from and write compressed output to a file.

The Nexus Class Library (Lewis, 2003): This offers tools to extract Newick tree strings from Nexus files. Once they are extracted, TreeShredder proceeds the same way as if they originated from a Newick file.

The `boost::dynamic_bitset` library (Siek and Allison, 2017): This is used for the basic representation of bipartitions on the set of taxa, i.e., splits (see subsection 4.1.1).

The util.cpp file: This file contains my functions that are globally useful and are not specific to one particular class, as well as macros to quickly customize TreeShredder.

3.2 Conclusion

TreeShredder comprises four classes, a `main()` function and four libraries. The `NewickParser` class parses Newick tree strings and returns the extracted splits and their information to the `DataDepot` class, which creates the `Split` objects. Features are processed in the `DataDepot` and `CommandlineParser` classes, which executes the tasks chosen with the command-line flags in the correct order. `Splits` are the basic objects in TreeShredder. They are managed by pointers to avoid time-consuming copying and guarantee their uniqueness in the program.

The start of indices, fraction notation, decimal places, default branch length, and reserved metacomment names can be customized by the user in the util.cpp file. The TreeShredder community should find a common set of customizations to reduce confusion and facilitate the sharing of TreeShredder resources.

4 Split Representation and Parallelization

4.1 Introduction

This chapter deals with determining a data structure to efficiently store the data. I evaluate this especially in the context of downstream parallelization.

4.1.1 Split Representation

Choosing the optimal representation and data structures to store splits is crucial for the performance of TreeShredder. This is because splits are the fundamental objects which are created and used in operations thousands to millions of times in a typical TreeShredder call.

The C++ standard library (`std::`) and third party libraries like BOOST (`boost::`) offer several classes suitable to store bipartitions on a set, such as

- The `std::vector<bool>` class: a vector of boolean values representing the taxa. Depending on the implementation, this class can be space-efficient by assigning each value 1 bit instead of 1 byte (which is typically the size of bool).

- The `std::vector<char>` class: a vector of char values where each char (8 bits) can represent 8 taxa. I implemented a wrapper class containing an object of this class. Individual bits of a byte of a char are assigned 1 taxon. Bit masks are used to retrieve the bits in a byte.

- The `std::bitset` class: a set of bits of a fixed size defined at compile time representing the taxa. This class offers standard logic operators that are extensively used for split operations. However, the number of taxa is not known at compile time of TreeShredder, which precludes defining the correct size.

- The `boost::dynamic_bitset` class (Siek and Allison, 2017) of the BOOST library (Boost developers, 2017): a set of bits similar to `std::bitset`, but with dynamic memory allocation. The size of the taxon set need not be known at compile time.

These classes differ in their implemented methods and operators, parallelizability, extent of optimization by the compiler, and dynamic memory allocation, with considerable runtime implications.

Dynamic memory allocation is crucial when the size of the sets is not known at compile time, as is the case with taxon sets in TreeShredder. This is why the `std::bitset` class cannot be used. The `boost::dynamic_bitset` class can be thought of as an extension of the static `std::bitset` class and was created with dynamic memory allocation in mind (Boost developers, 2017; Siek and Allison, 2017).

To find the best split representation class, I implement a testing routine that repeatedly executes – single-threaded and multi-threaded – certain tasks which are ubiquitous in TreeShredder : it creates splits using the different classes, sets specific taxa, flips all bits and inserts them into a container with an object of the class as key and a pointer to the inserted object as value.

4.1.2 Parallelization and Operating Systems

Besides efficient algorithms and data structures, parallelization often is key to runtime performance. But performance only improves if they work together seamlessly. Choosing the right parallelization API requires thorough testing of the program's basic operational units. In TreeShredder's case this is the `boost::dynamic_bitset` class, which is used to represent splits. TreeShredder is designed to deal with thousands or even millions of trees containing thousands of taxa. With these large numbers, the number of unique splits could grow by tn, where t is the number of trees and n is the number of taxa.

4.2 Methods

4.2.1 OpenMP

OpenMP is an easy-to-use collection of compiler directives, library routines and environment variables to enable shared-memory parallelism in C, C++ and Fortran (Dagum and Menon, 1998; OpenMP Architecture Review Board, 2018).

It implements a fork-join model of parallel execution defined by directives (see Fig. 4.1). This parallel execution is achieved by assigning execution work, i.e., tasks, to threads. Threads are execution entities with access to shared or thread-private variables. The shared access is for data in memory and other local storage, such as machine registers and cache, which speeds up access to shared data (e.g., in a for loop). Each thread also has access to its own thread-private memory for private variables that cannot be accessed by other threads. (OpenMP Architecture Review Board, 2018, p. 23)

If a thread enters a parallel region, denoted by a `parallel` construct, it becomes the master thread of a newly created team of threads that execute the tasks of the region in parallel. The number of threads in the team can be set with the `num_threads` clause in the directive. *Every* thread in the team (including the master thread) then executes *all* the code in the region, thus, they execute single instructions but on multiple data (SIMD, Rauber and Rünger, 2012, p. 19). (OpenMP Architecture Review Board, 2018, p. 74 ff.)

However, if the team of threads enters a `worksharing` construct, the work is divided among the threads instead of being executed multiple times (OpenMP Architecture Review Board, 2018, p. 21).

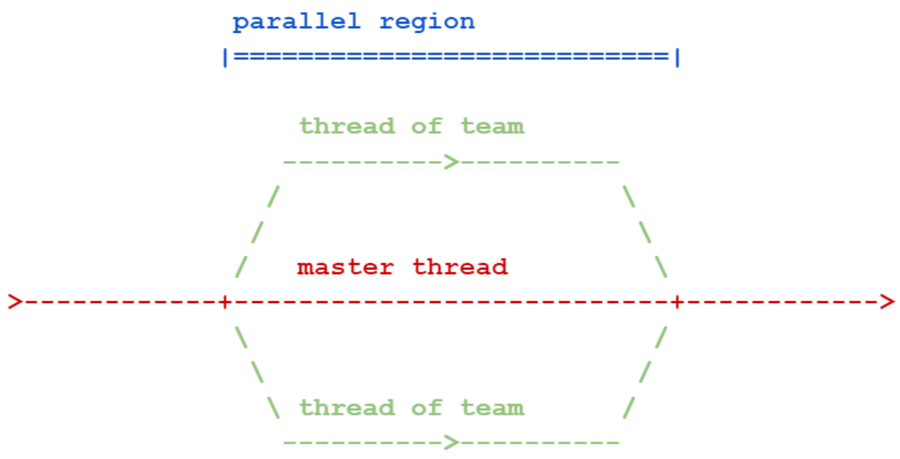


Figure 4.1: OpenMP fork-join model: When a thread encounters a `parallel` region (blue), it becomes the master thread (red) and creates a team of threads (green, includes the master thread): this is the forking step. At the end of the `parallel` region, there is an implicit barrier where the threads of the team wait for each other and cease to exist: this is the joining step. Only the master thread continues execution. Each thread can access shared variables in memory and private variables in thread-private memory. If the `parallel` region contains a worksharing-loop the iterations of the loop are distributed as chunks among the threads of the team.

There is an implicit barrier at the end of a `parallel` region where threads wait for each other to finish their tasks. After that, only the master thread continues execution of code outside the region. (OpenMP Architecture Review Board, 2018, p. 76)

4.2.1.1 Worksharing-Loop and Scheduling

The worksharing-loop construct of a `parallel` region causes the iterations of a loop to be divided into chunks. They are then distributed among the threads of a team and executed in parallel. An example is the `for` directive (OpenMP Architecture Review Board, 2018, p. 101 ff.). There are three kinds of scheduling strategies available:

- `static`: causes the iterations of the for-loop to be divided into chunks of equal size and distributed among the threads in a round-robin system. The chunk size can be customized. By assigning each thread exactly one chunk, the information of the tasks can be merged in the initial order of their iterations as if they had been executed sequentially.

- `dynamic`: like `static`, but the chunks are distributed on request by the threads once they are ready to execute a new chunk. This is useful when the amount of work needed to complete each chunk is uneven and, therefore, the distribution of an equal number of chunks to achieve roughly equal execution time for each thread (called load balancing) does not work. This scheduling has higher overhead than `static` scheduling because it dynamically distributes the chunks at runtime.

- `guided`: like `dynamic`, but the initial chunks are larger to decrease overhead, and chunks get smaller towards the end to improve load balancing.

4.3 Results and Discussion

Fig. 4.2 shows the parallel runtimes and speedup of the split representation classes mentioned in subsection 4.1.1 on Windows (Intel Core i5-8250U CPU with 4 cores, 8 threads, and 8 GB memory). This is done by simulating for each class common operations that are expected to be executed many times in a typical TreeShredder call using different numbers of threads. These operations are: the creation of splits for each class, setting specific bits, flipping all bits and inserting them into a container with an object of the class as key and a pointer to the inserted object as value.

It can be seen that the runtime on Windows generally increases the more threads are used, the opposite of what would be expected. The split representation classes `vector<char>` and `dynamic_bitset` are the fastest when using only 1 thread and get slower with more threads.

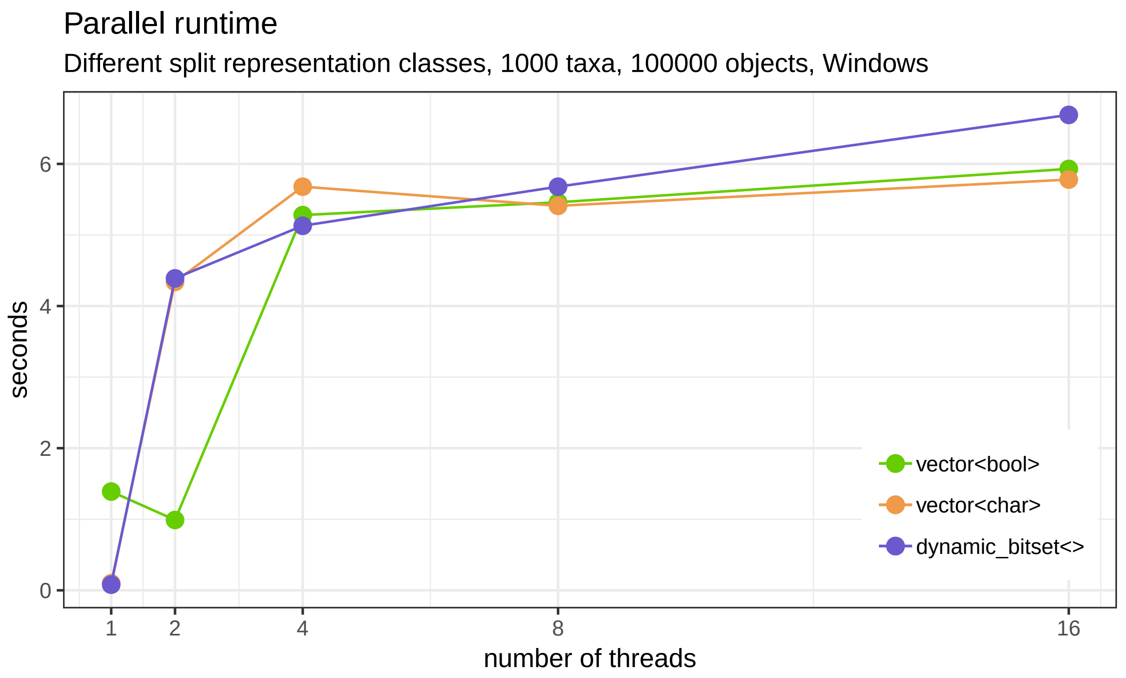
Fig. 4.3 shows the parallel runtimes and speedup of the split representation classes on Linux. Here, the increase in speedup generally slows down beyond four threads. Even though `vector<bool>` shows the best speedup, `vector<char>` and `dynamic_bitset` have the fastest runtimes overall.

Contrary to what was expected, using OpenMP together with each of the classes did not lead to a speedup on Windows, but instead to a slowdown. Apparently, the shared-memory architecture of Windows is not conducive to creating the split representation objects in parallel. Even extensive tests were not able to elucidate why no speedup could be achieved by using more CPU cores with OpenMP. This surprising result, where the speedup depends on the operating system, necessitates that the user should first test the suitability of their machine and operating system for parsing Newick trees in parallel (see subsection 5.1.2).

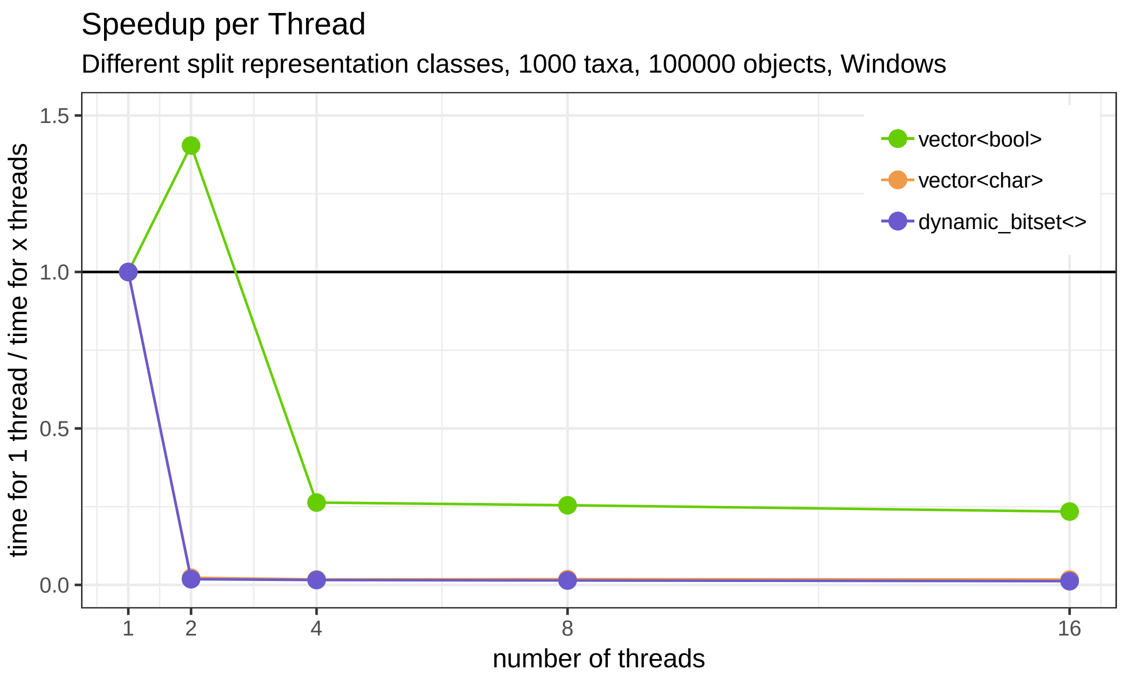
4.3.1 Testing Parallelizability on Operating System

To test whether the operating system TreeShredder is run on is suitable for the parallel parsing of Newick trees, I implemented the `-testParallel` flag option. This routine creates `dynamic_bitsets`, sets bits, and flips all bits. This is done equally often using one thread (serial) and two threads (parallel), subsequently the serial and parallel runtimes are compared by calculating their ratio. If the ratio of the runtimes is very low, the parallelizability of `dynamic_bitsets` is lacking. In that case, using the `-parseParallel` flag to parse Newick files in parallel is not advised. However, if the runtimes ratio is high (> 1), parsing Newick files in parallel is justified.

Note this testing routine does not parse any Newick trees, but there is a grey area where the parallel part of the test takes longer than the serial part (i.e., if

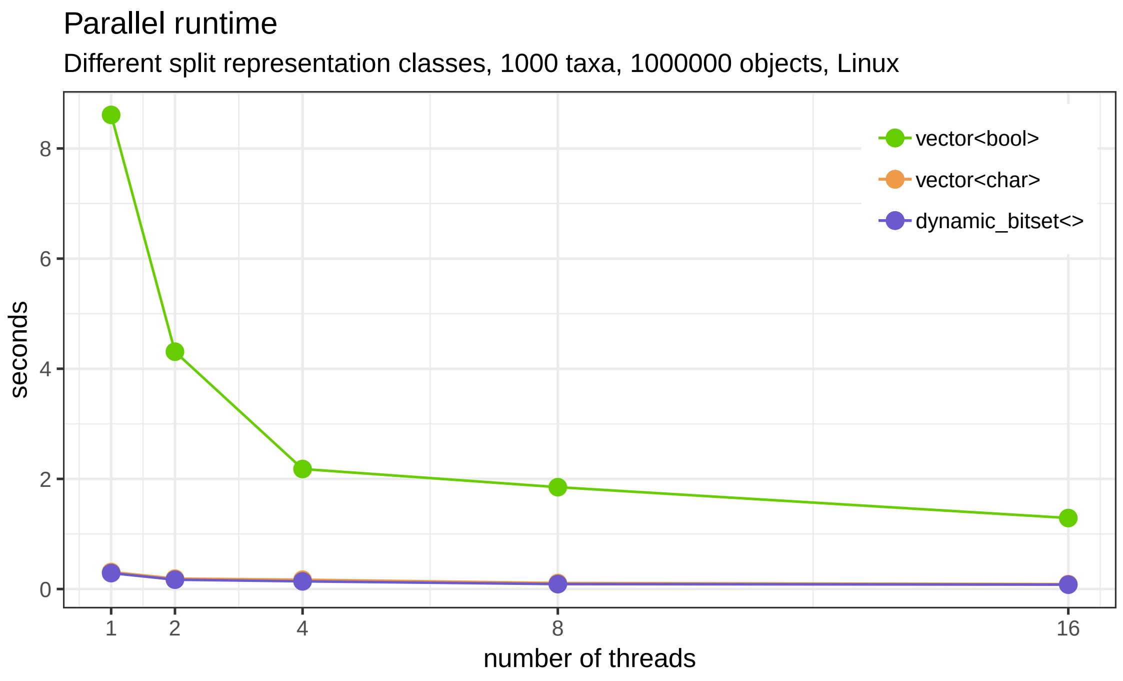


(a) Parallel runtimes of different split classes on Windows: The testing routine of the split representation class tends to take *longer* the more threads are used. Single-threaded `vector<char>` and `dynamic_bitset` are fastest.

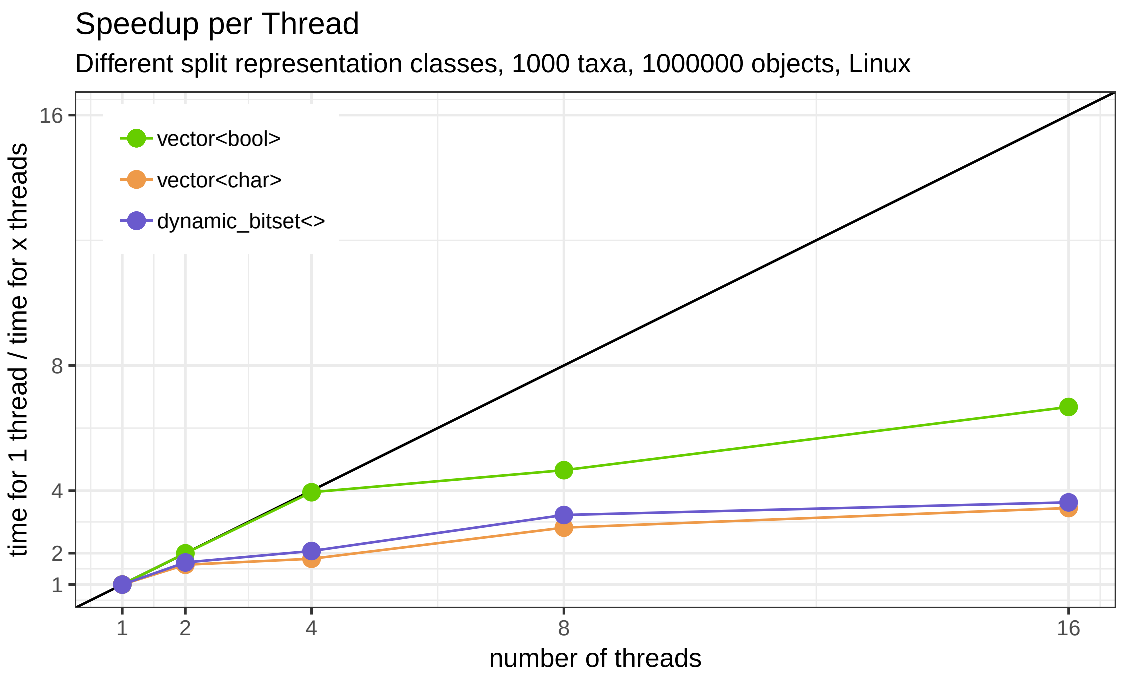


(b) Speedup of different split classes on Windows: Results below the black line indicate slowdown, rather than speedup.

Figure 4.2: Parallel runtimes and speedup of different split representation classes on Windows: Test runs simulate the parallel creation of splits of the different representation classes (`vector<bool>`, `vector<char>`, and `dynamic_bitset`), setting specific bits, flipping all bits and inserting them into a container with an object of the class as key and a pointer to the inserted object as value.



(a) Parallel runtimes of different split classes on Linux: `vector<char>` and `dynamic_bitset` are the fastest.



(b) Speedup of different split classes on Linux: The black line indicates perfect speedup. `vector<bool>` has the best best speedup.

Figure 4.3: Parallel runtimes and speedup of different split representation classes on Linux: Test runs simulate the parallel creation of splits of the different representation classes (`vector<bool>`, `vector<char>`, and `dynamic_bitset`), setting specific bits, flipping all bits and inserting them into a container with an object of the class as key and a pointer to the inserted object as value.

the runtimes ratio is < 1). In this grey area, speedup using multiple threads when parsing a Newick file in parallel could still be possible. This is because the speedup could be achieved in other aspects of the tree parsing routine (see section 5.2), even though the creation of `dynamic_bitset`s and setting of bits is slower.

Therefore, before parsing a large Newick file in parallel, I recommend double-checking the parallel parsing of Newick strings with a smaller Newick file example on the machine and operating system planned to use.

4.4 Conclusion

Based on the results, I decided to use `boost::dynamic_bitset` to represent splits in TreeShredder for the following reasons: It allows for dynamic memory allocation, which is crucial when the number of taxa is not known at compile time. It achieves the fastest runtimes when run multi-threaded on Linux (with acceptable speedup), and the fastest when run single-threaded on Windows. Finally, the BOOST library offers extensive, well-documented standard logical operations such as bitwise AND, OR, and XOR.

5 Reading Files and Parsing Trees

There are diverse input file formats accepted by TreeShredder, which contain various information of trees, splits, or taxa and will be described in the following.

Files containing tree information such as Newick, Nexus and TreeShredder files are read with the `-f` flag and can be compressed or uncompressed. They are the main source of information for TreeShredder. Two other types of files are Taxa files and Split files, which contain taxon names and incomplete splits, respectively.

5.1 Reading Newick and Nexus files

The Newick (Felsenstein et al., 1986) and Nexus (D. R. Maddison, Swofford, and W. P. Maddison, 1997) file formats are the common formats to store tree information input in phylogenetic programs.

The Newick tree format is a text format that joins nodes (which represent whole subtrees when they are internal nodes) of a tree by enclosing them in parentheses forming new nodes (or subtrees). Thus, the relationship of nodes (or subtrees) of a tree can be written as a single string of characters. The subtrees are separated by commas and the Newick string is terminated by a semicolon (Felsenstein et al., 1986; Page, 2003). By convention, a Newick tree is assumed to be rooted if the outermost parenthesis encloses exactly 2 subtrees, and unrooted in all other cases. See Figs. 1.1 and 5.1 for a depiction of Newick trees and their corresponding Newick strings. The Newick file format consists of one or more Newick trees, which may be newline-separated. The Nexus file format is modular and consists of blocks such that related information can be stored in addition to Newick trees (D. R. Maddison, Swofford, and W. P. Maddison, 1997). This related information may include morphological, molecular, distance, assumptions, or sets data, and more. TreeShredder uses the Nexus Class Library (Lewis, 2003) to extract information of specific blocks (mainly the TREES block) from Nexus files (see chapter 3).

The routine for reading Newick and Nexus files is a method of the `DataDepot` class (see subsection 3.1.3). It is a two-step process: first, the Newick tree strings are extracted from the file, then they are parsed.

5.1.1 Newick trees extraction step

The Newick trees are counted and then extracted from the file. With the `-range` flag, one can specify the first and last trees of a range tha is to be extracted. The Nexus Class Library is used to extract Newick tree strings from Nexus files.

The set and order of the taxa in the first extracted Newick string becomes the master taxon order (except if a Taxa order file is given by the user with the `-t` flag, see section 5.3), which is used to check that all trees in a file contain the same taxon set even if the order differs (which is to be expected).

5.1.2 Newick trees parsing step – in serial and in parallel

Once the Newick tree strings are extracted, they are parsed using the `NewickParser` class (see subsection 3.1.4). This part of the routine can be parallelized using the `-parseParallel` flag (only after checking that the parallel parsing of Newick trees is possible on the operating system using the `-testParallel` flag, see subsection 4.2.1 f.).

The serial mode creates a `NewickParser` object for each Newick tree in a serial manner. The `NewickParser` object stores the split and tree information in the correct containers corresponding to trivial, nontrivial, and root splits (see subsections 3.1.3 and 3.1.4). Additionally, the relationship of splits and their trees is also stored (i.e., which split was contained in which tree). These containers belong to the `DataDepot` object but are passed to the `NewickParser` by reference.

The parallel mode creates `NewickParser` objects in parallel using OpenMP with the `static` schedule. This schedule will distribute *contiguous* Newick tree blocks to the threads. The `ordered` directive then ensures that the threads merge their split and tree information in the order they were previously assigned the blocks. This is important for the correct merging of the split and tree information after parsing. This way, the internal state of the `DataDepot` object is identical independent of whether the trees were parsed in serial or in parallel.

5.1.3 Merging files

Using the `-merge` flag, one can merge the tree information of any number of Newick, Nexus, *and/or* TreeShredder files provided their taxon sets are identical.

In addition to being able to merge the information of different file formats this way, the user can also achieve a kind of pseudo-parallelization when reading the trees of very large Nexus or Newick files (i.e, several million trees). The first step to do this is “splitting” the file’s trees, e.g., by reading consecutive ranges (containing millions) of trees and outputting TreeShredder files for the trees in each of those ranges separately. In the next step, these TreeShredder files can then be merged using the `-merge` flag. This pseudo-parallelization method is independent of the “proper” parallelization method using the `-parseParallel` flag (see subsection 5.1.2). Splitting tree information and then merging their TreeShredder files can also be used when the tree information of interest resides in non-consecutive ranges of (large) Newick or Nexus files.

5.2 Splits Extraction Algorithm

The Splits Extraction Algorithm extracts all bipartitions on the taxon set from a Newick tree string that are induced by parenthesis notation (see section 5.1). The approach used in TreeShredder was developed by Marc Zobel for the software development course “PR Praktikum aus Bioinformatik” (“PR Laboratory Bioinformatics") at University Vienna in 2015. Unlike other approaches, the splits are directly extracted from the Newick string, without the detour of first constructing the tree graph.

Originally, the algorithm determined all nontrivial splits of a tree while parsing the Newick string and keeping track of taxon names, and opening and closing parentheses. I extended the algorithm in four ways:

1. I extended the algorithm to simultaneously extract the node and edge information of trivial and nontrivial splits, such as comments, metacomments, or branch lengths following a colon '`:`'. (Note that it is the `DataDepot` class which connects the node and edge information of the trivial splits to their respective trivial splits, which are not extracted by the Splits Extraction Algorithm.)

2. I extended the algorithm to determine the rootedness of the Newick tree (see subsection 5.2.1).

3. I extended the algorithm to re-use the splits matrix and its `dynamic_bitsets` to parse different Newick tree strings. This is necessary because creating the splits matrix again for each Newick string would be computationally very expensive and detrimental to parallelization.

4. I extended the algorithm to count the opening parentheses in the Newick string to determine how many `dynamic_bitsets` are needed. Combined with the knowledge how many taxa there are in the trees, the sizes of both, the splits matrix and the `dynamic_bitsets`, can be predefined. Doing this is much faster than the alternative of expanding the dimensions of the matrix whenever an opening parenthesis or taxon name was parsed.

The Splits Extraction Algorithm is implemented in the `NewickParser` class (see subsection 3.1.4). Fig. 5.2 (page 35) explains the algorithm with an example of a rooted Newick tree (see Fig. 5.1). An opening parenthesis '`(`' indicates that a new split is begun. An index variable that increases with each encountered opening parenthesis is pushed onto a stack. Those index variables correspond to a `dynamic_bitset` in the $n \times m$ splits matrix, where n is the number of splits (= number of opening parentheses) and m is the number of taxa. A closing parenthesis '`)`' indicates that a split is completed – the previously added index variable is popped from the stack. Whenever a taxon name is encountered, the bits at the position of that taxon in all splits whose indices are on the stack will be set to 1. Thus, a vector of `dynamic_bitsets` will be filled with the split information induced by each pair of parentheses of the Newick string.

To keep the computationally expensive creation of `dynamic_bitsets` (see subsection 4.1.1) to a minimum (which also facilitates parallelization of the parsing routine), the splits matrix is re-used after setting all bits to 0 after parsing a Newick string. Thus, the runtime complexity of parsing t Newick strings of size x is given by $\mathcal{O}(t * x)$.

The Splits Extraction Algorithm does not extract trivial splits, i.e., those of edges that connect a leaf node with an internal node. These splits must be created separately (and only once, because the set of trivial splits is identical among trees).

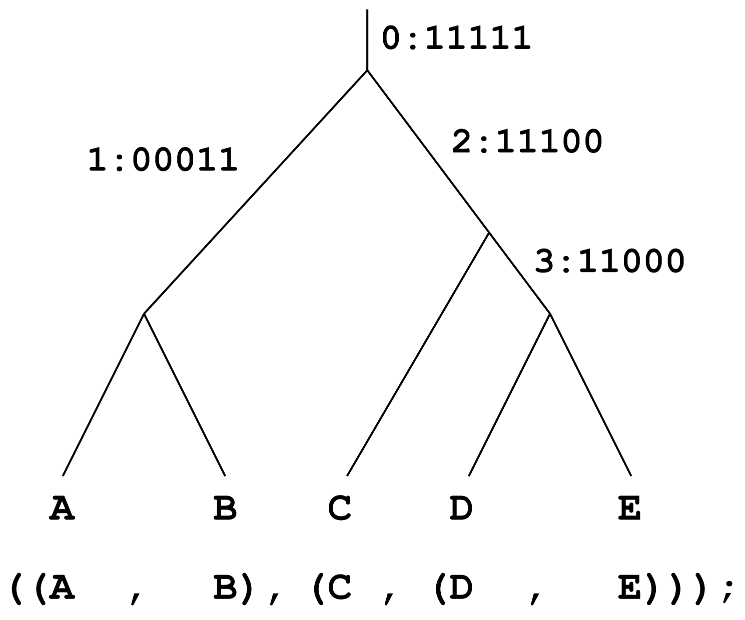


Figure 5.1: Example of a rooted tree and its corresponding Newick string: This tree's Newick string is used to demonstrate the Splits Extraction Algorithm in Fig. 5.2. Only nontrivial splits are extracted by the algorithm. The number before the colon indicates the index of the split in the splits extraction matrix. The `0:11111` split corresponds to the outermost pair of parentheses, which does not induce a bipartition on the taxon set. The bits in the "bit notation" format correspond to the taxa in reverse order.

5.2.1 Extending the Algorithm to determine Rootedness

Rootedness of trees introduces a special case where the two root split arms induce identical bipartitions on the set of taxa. A Newick tree is assumed to be rooted if the outermost pair of parentheses contains exactly 2 nodes. There are three possibilities: these 2 nodes are 2 internal nodes (e.g., "`((A,B),(C,D));`"), 1 internal node and 1 leaf node (e.g., "`(A,(B,C));`"), or 2 leaf nodes (e.g., "`(A,B);`").

By counting how often the stack contains exactly 1 index variable (the one corresponding to the outermost split) after a taxon name or a closing parenthesis was parsed, I extended the Splits Extraction Algorithm to determine if the Newick tree is rooted while extracting splits. The tree is rooted if the root node degree count is exactly 2, and unrooted in all other cases. In Fig. 5.2, the stack contains exactly 1 index variable in steps 5 and 12, thus, the example Newick tree is correctly classified as rooted.

5.3 Reading Taxa files

The Taxa file contains newline-delimited taxon names. It is read with the `-t` flag. The taxon names will then comprise the master taxa. This is used to set a custom taxon order, or provide taxon names for the `-incompleteSplits` flag, which generates random incomplete splits of a custom taxon set.

5.4 Reading Split files

The Split file contains newline-delimited incomplete splits (see section 7.1). It is read with the `-cong` flag. This is used to find trees that are congruent with a given set of incomplete splits, obtain congruency measures of the incomplete splits, and determine the congruency status of the nontrivial splits with the incomplete splits (see section 7.1).

5.5 Reading TreeShredder files

I developed the TreeShredder file format to store all necessary information for a TreeShredder analysis. Its objective is to save time when reanalysing or merging datasets. The file format contains blocks of specific data relating to split information and tree information that resembles the internal data structure of TreeShredder after parsing Newick strings. It contains information that is more condensed than the information of an equivalent Newick file. Thus, it is smaller, faster to read (see subsection 5.6.1), and easier to manipulate. Converting Newick file information to TreeShredder file information is irreversible – some information is lost. This is because node and edge information (e.g., branch length) is summed up or otherwise condensed, which makes it impossible to discern to which tree these data belong. The relationship of root splits to their trees is also lost, while the relationship of nontrivial splits to their trees is not. However, all information that is useful to TreeShredder is fully preserved.

The TreeShredder file reading routine is a method of the `DataDepot` class. The blocks of the TreeShredder file correspond to instance variables of the `DataDepot` class. Split information strings are parsed by and stored in `Split` objects which are then stored in the corresponding containers of the `DataDepot` object. TreeShredder files must contain a "Taxa" and a "Trivial Splits" block, but may miss, e.g., "Nontrivial Splits", "Root Splits" and "Trees" blocks (see Fig. 5.3b).

```
   ((A,B),(C,(D,E)));
0    0 0   0  0 0
     0 0   0  0 0
     0 0   0  0 0
     0 0   0  0 0    RDG=0
```

Opening parenthesis is parsed.
Index variable 0 is
pushed onto the stack.

```
   ((A,B),(C,(D,E)));
0    0 0   0  0 0
1    0 0   0  0 0
     0 0   0  0 0
     0 0   0  0 0    RDG=0
```

Opening parenthesis is parsed.
Index variable 1 is
pushed onto the stack.

```
   ((A,B),(C,(D,E)));
0    1 0   0  0 0
1    1 0   0  0 0
     0 0   0  0 0
     0 0   0  0 0    RDG=0
```

Taxon name is parsed.
Corresponding bits at the index
variables are set to 1.

```
   ((A,B),(C,(D,E)));
0    1 1   0  0 0
1    1 1   0  0 0
     0 0   0  0 0
     0 0   0  0 0    RDG=0
```

Taxon name is parsed.
Corresponding bits at the index
variables are set to 1.

```
   ((A,B),(C,(D,E)));
0    1 1   0  0 0
1    1 1   0  0 0
     0 0   0  0 0
     0 0   0  0 0    RDG=1
```

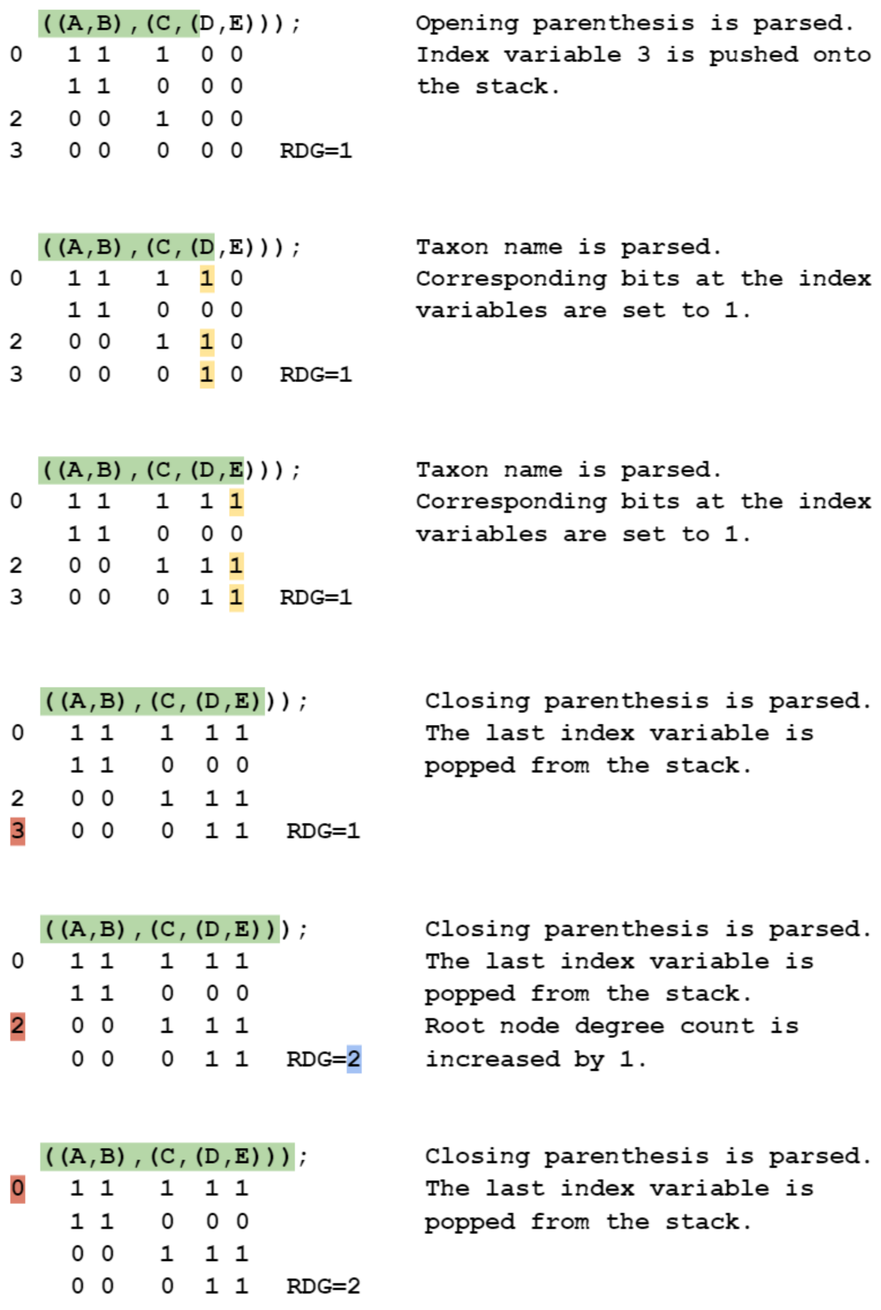
Closing parenthesis is parsed.
The last index variable is
popped from the stack.
Root node degree count is
increased by 1.

```
   ((A,B),(C,(D,E)));
0    1 1   0  0 0
     1 1   0  0 0
2    0 0   0  0 0
     0 0   0  0 0    RDG=1
```

Opening parenthesis is parsed.
Index variable 2 is pushed onto
the stack.

```
   ((A,B),(C,(D,E)));
0    1 1   1  0 0
     1 1   0  0 0
2    0 0   1  0 0
     0 0   0  0 0    RDG=1
```

Taxon name is parsed.
Corresponding bits at the index
variables are set to 1.



```
 ((A,B),(C,(D,E)));        Opening parenthesis is parsed.
0   1 1   1  0 0           Index variable 3 is pushed onto
    1 1   0  0 0           the stack.
2   0 0   1  0 0
3   0 0   0  0 0   RDG=1
```

```
 ((A,B),(C,(D,E)));        Taxon name is parsed.
0   1 1   1  1 0           Corresponding bits at the index
    1 1   0  0 0           variables are set to 1.
2   0 0   1  1 0
3   0 0   0  1 0   RDG=1
```

```
 ((A,B),(C,(D,E)));        Taxon name is parsed.
0   1 1   1  1 1           Corresponding bits at the index
    1 1   0  0 0           variables are set to 1.
2   0 0   1  1 1
3   0 0   0  1 1   RDG=1
```

```
 ((A,B),(C,(D,E)));        Closing parenthesis is parsed.
0   1 1   1  1 1           The last index variable is
    1 1   0  0 0           popped from the stack.
2   0 0   1  1 1
3   0 0   0  1 1   RDG=1
```

```
 ((A,B),(C,(D,E)));        Closing parenthesis is parsed.
0   1 1   1  1 1           The last index variable is
    1 1   0  0 0           popped from the stack.
2   0 0   1  1 1           Root node degree count is
    0 0   0  1 1   RDG=2   increased by 1.
```

```
 ((A,B),(C,(D,E)));        Closing parenthesis is parsed.
0   1 1   1  1 1           The last index variable is
    1 1   0  0 0           popped from the stack.
    0 0   1  1 1
    0 0   0  1 1   RDG=2
```

Figure 5.2: Example of the Splits Extraction Algorithm, rooted tree: This figure begins on the previous page. Parsed characters of the Newick string are highlighted in green. The leftmost column is the stack of index variables. Popped index variables are highlighted in red. There is one row of bits for each opening parenthesis (i.e., split) in the splits matrix. Bits of the splits matrix which are newly set to 1 are highlighted in yellow. Root node degree count (RDG) increments are highlighted in blue. Note that the second and third row of bits of the splits matrix are flipped equivalents. This is because they represent the two edges connected to the root node (see Fig. 5.1).

5.5.1 Structure of TreeShredder files

Fig. 5.3 shows an example Newick string and the equivalent TreeShredder file. Corresponding data snippets are highlighted in color.

A TreeShredder file starts with a "`#@TreeShredderFile`" string followed by blocks of data in a specific order:

1. The "Taxa" block contains the number of taxa followed by the newline-separated taxon names.
2. The "Trivial Splits" block contains the number of unique trivial splits followed by the newline-separated information of the trivial splits.
3. The "Nontrivial Splits" block contains the number of unique nontrivial splits followed by the newline-separated information of the nontrivial splits. The order of the nontrivial splits corresponds to their index (starting at 0 per default; see subsection 3.1.6), which I implemented to be the exact same order as they are created when parsing the trees in a file containing Newick trees using the Splits Extraction Algorithm (see section 5.2).
4. The "Root Splits" block contains the number of unique root split arms (must be a multiple of 2) followed by the information of the newline-separated root split arms.
5. The "Trees" block contains the number of trees followed by newline-separated arrays of space-separated indices of the nontrivial splits (see "Nontrivial Splits" block) of each tree.

The "Trees" block is optional, TreeShredder can be prevented from reading or writing it with the `-noTreesBlock` flag. The "Nontrivial Splits" and "Root Splits" blocks are omitted if there are no nontrivial splits (as in star trees) or root splits (as in unrooted trees).

5.5.2 Split information string in a TreeShredder file

The information of a split in a TreeShredder file consists of entries in six tab-delimited columns (see Fig. 5.3b):

1. A bit string of 0s and 1s where the bits correspond to taxa in *reverse* order, which means that the first taxon is represented by the rightmost (least significant) bit, and the last taxon by the leftmost (most significant) bit. To guarantee uniqueness of splits, the rightmost bit (i.e., the first taxon) is defined to be always 1.
2. The number of taxa on the smaller side of the split. This is $= 1$ for trivial, and ≥ 2 for nontrivial splits.
3. The occurrence count of the split. This is ≥ 1.

(a)
```
Newick string:
(A:[&metaNum=0.25],((B,C:0.66)cherry:0.33,(D[&metaStr=bla]:[&metaNum=2.75],
(E[blaNode]:[blaEdge],F))99):[&metaNum=0.35]);
```

(b)
```
#@TreeShredderFile

Taxa:
6
A
B
C
D
E
F

Trivial Splits:
6
000001  1       1       2.000000        [&]:[&metaNum=0.350000] []:[]
111101  1       1       1.000000        [&]:[&] []:[]
111011  1       1       0.660000        [&]:[&] []:[]
110111  1       1       1.000000        [&metaStr=bla]:[&metaNum=2.750000]      []:[]
101111  1       1       1.000000        [&]:[&] [blaNode]:[blaEdge]
011111  1       1       1.000000        [&]:[&] []:[]

Nontrivial Splits:
3
111001  2       1       0.330000        [&nodeInfoString=cherry]:[&]    []:[]
000111  3       1       1.000000        [&]:[&nodeAsEdgeInfo=99.000000] []:[]
001111  2       1       1.000000        [&]:[&] []:[]

Root Splits:
2
000001  1       1       1.000000        [&]:[&metaNum=0.350000] []:[]
000001  1       1       1.000000        [&]:[&metaNum=0.250000] []:[]

Trees:
1
0 1 2
```

Figure 5.3: Example of a Newick string and its equivalent TreeShredder file: Corresponding data snippets are highlighted in the same color in (a) the Newick string and (b) the TreeShredder file. Branch lengths that were given in the Newick string are highlighted in green. Comments are highlighted in purple. Metacomments are highlighted in orange. The node strings “`cherry`” and “`99`” highlighted in red are interpreted as a string that belongs to the node and a number that belongs to the edge, respectively. They are stored as metacomments with automatically assigned keys. (The keys are reserved macro string literals assigned by TreeShredder. They cannot be used for other metacomments.) The 2 splits of the “Root Splits” block are the 2 arms of the root split. The information of these 2 arms is gathered into 1 trivial split (the first split in the “Trivial Splits” block) in a special way described in subsection 5.5.2. The sum of the 2 default branch lengths of the 2 root split arms is highlighted in blue. Note that the metacomments of the root split arms were *not* added up, rather the maximum was taken.

4. The branch length of the split. This is > 0. (When parsing Newick strings, TreeShredder assumes a default branch length of 1 if none is given.)
5. The node and edge metacomments (BEAST developers, 2017) of a split. The ampersand '`&`' distinguishes it from a normal comment. Metacomments contain key:value pairs separated by commas '`,`'. The value can be a string, which is replaced if an identical split's new metacomments with the same key is parsed, or a number, which is summed up if an identical split's new metacomments with the same key is parsed. Metacomments that belong to the node that is on the distant end of an edge from the root is enclosed in the first square bracket "`[&]`". Metacomments that belong to the edge of the split are enclosed in the second square bracket "`[&]`". The square brackets are separated by a colon '`:`'. A placeholder string "`[&]:[&]`" indicates that there are no metacomments for this split.
6. The node and edge comments of a split. This is analogous to the metacomments, but without the ampersand '`&`'. Comments are treated as strings, and will be concatenated (separated by a comma '`,`') for identical splits.

5.5.2.1 Root split information

By convention, Newick strings are in rooted format if the outermost pair of parentheses encloses exactly 2 nodes (see section 5.2). This induces 2 identical splits – the 2 arms of the root split – joined by the root node of degree 2.

The information of these arms is stored, redundantly, in two ways: once in the "Root Splits" block, and once either in the "Trivial Splits" or "Nontrivial Splits" block, depending on the classification of the root split. The information in the "Root Splits" block is to correctly retrieve the information of the root arms when mapping tree information onto a reference tree (see section 6.2.2). The "Trivial Splits" and "Nontrivial Splits" blocks are for all other cases where the root split is treated as a single split in an unrooted context, not as 2 arms of a split.

The information of the root split arms is combined thus (see first split in the "Trivial Splits" block of Fig. 5.3b): The branch lengths are added; if the value of metacomments with identical keys is a number then the *maximum* is taken, if it is a string the value is replaced; comments are concatenated.

5.6 File-IO Results and Discussion

Here I compare runtimes of different input and output steps of TreeShredder. For information about the datasets used, see chapter 2.

5.6.1 Comparing Newick and TreeShredder file runtimes

Fig. 5.4a shows that the runtime of reading Newick files (including parsing the trees) grows linearly with the number of trees.

Fig. 5.4b shows that reading TreeShredder files is up to 30 times faster than reading Newick files containing the same tree information, making it the preferred

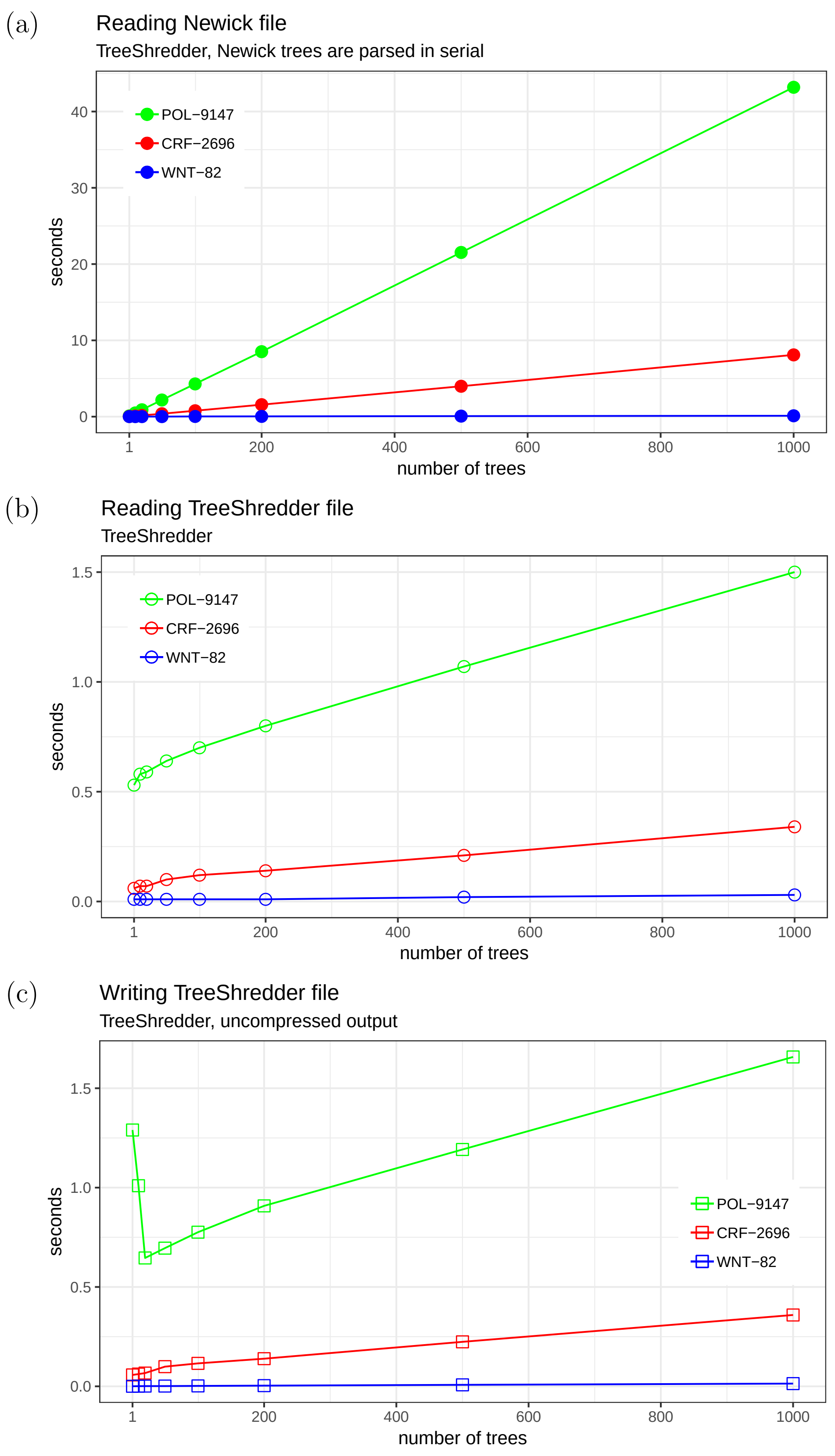


Figure 5.4: TreeShredder runtimes of reading Newick files, and reading and writing TreeShredder files for the WNT-82, CRF-2696, and POL-9147 datasets: The plots show the runtime of (a) reading the Newick file and parsing the trees in serial, (b) reading the TreeShredder file, and (c) writing the uncompressed TreeShredder file.

starting point for further analysis with TreeShredder for large datasets such as POL-9147.

Once a Newick file is read and its trees parsed, transforming its information into a TreeShredder file is quick and well worth the effort (see Fig. 5.4c). The runtime increases roughly linearly with the number of trees, likely because the "Trees" block in the TreeShredder file grows linearly with the number of trees. This overrides the effect of the "Nontrivial Splits" block which grows sub-linearly with the number of trees (see Fig. 3.2a). Since reading Newick files of small datasets (e.g., WNT-82) is fast anyway, transforming the data to a TreeShredder file may not be necessary. The inherent TreeShredder file modifiability advantage over Newick files remains, however.

Fig. 5.5 shows the runtime of the serial reading routine of Newick files and the serial reading and writing routine of TreeShredder files for the ACO-225, PTP-116, PF1-367, and PF7-205 datasets with up to 100,000 trees. Reading TreeShredder files is about 20 times faster than reading the Newick files (e.g., 2 seconds vs 40 seconds for ACO-225). Writing TreeShredder files is also fast.

Fig. 5.6a shows the runtime of the parallelized reading routine of Newick files, and the serial reading and writing routine of TreeShredder files for the GEN-404 dataset comprising almost 15 million of trees. Reading TreeShredder files is up to four times faster than reading the Newick files in parallel using 16 threads. Converting large Newick files into TreeShredder files is worth the effort and extra time spent on writing TreeShredder files.

Fig. 5.6b shows the runtime speedup per thread of reading 10 million GEN-404 Newick trees. Overall, speedup is weak.

Fig. 5.7 shows the runtime of the serial reading routine of Newick files and the serial reading and writing routine of TreeShredder files for the RAN-200, RAN-300, and RAN-400 datasets with up to 1,000,000 trees. Reading TreeShredder files takes about a quarter to a third less time than reading the Newick files. The time needed to convert large Newick files into compressed TreeShredder files increases super-linearly with increasing number of taxa and trees. However, converting can still save time in subsequent analyses.

5.6.2 Output compression

Fig. 5.8a shows the composition of time of reading millions of GEN-404 Newick trees and writing their information to a compressed or uncompressed TreeShredder file. The benchmarks were done on Linux. Surprisingly, the writing time of compressed TreeShredder file output is *lower* than for uncompressed output. The user time, which measures the CPU time spent executing user code (i.e., program code) in user mode (Kerrisk, 2010), increases slightly for compressed output because of the compression work needed to be done. The system time, which measures the CPU time spent executing system code (i.e., I/O calls) in kernel mode, is *greater* for the uncompressed file output. Apparently, compressing output and, thus, writing less output to the disk is faster than writing uncompressed output for large files. This is because disk access is known to be utterly slow, while memory access is fast. Thus, compression in memory easily pays off by saving slow access while writing. In this

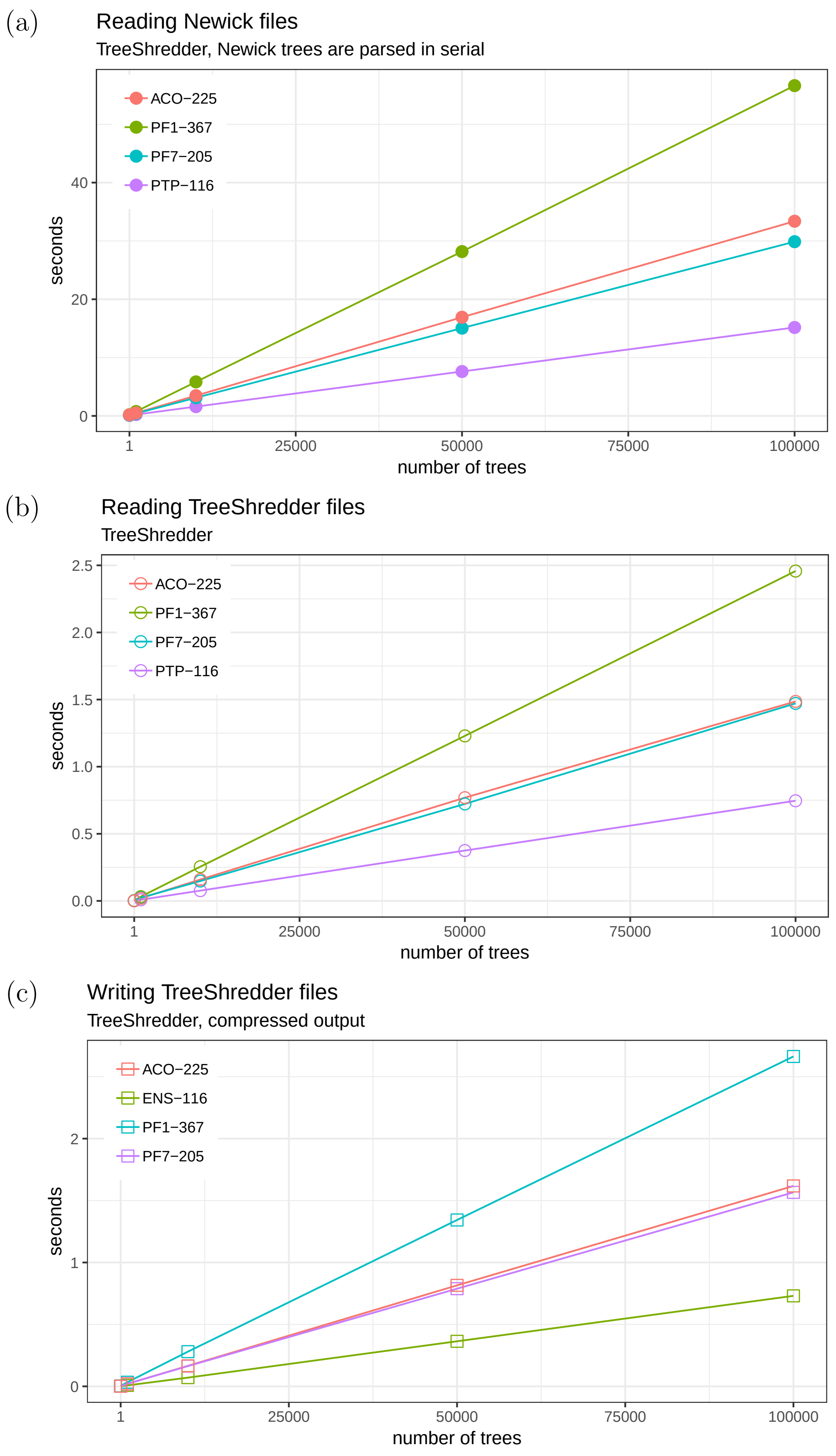
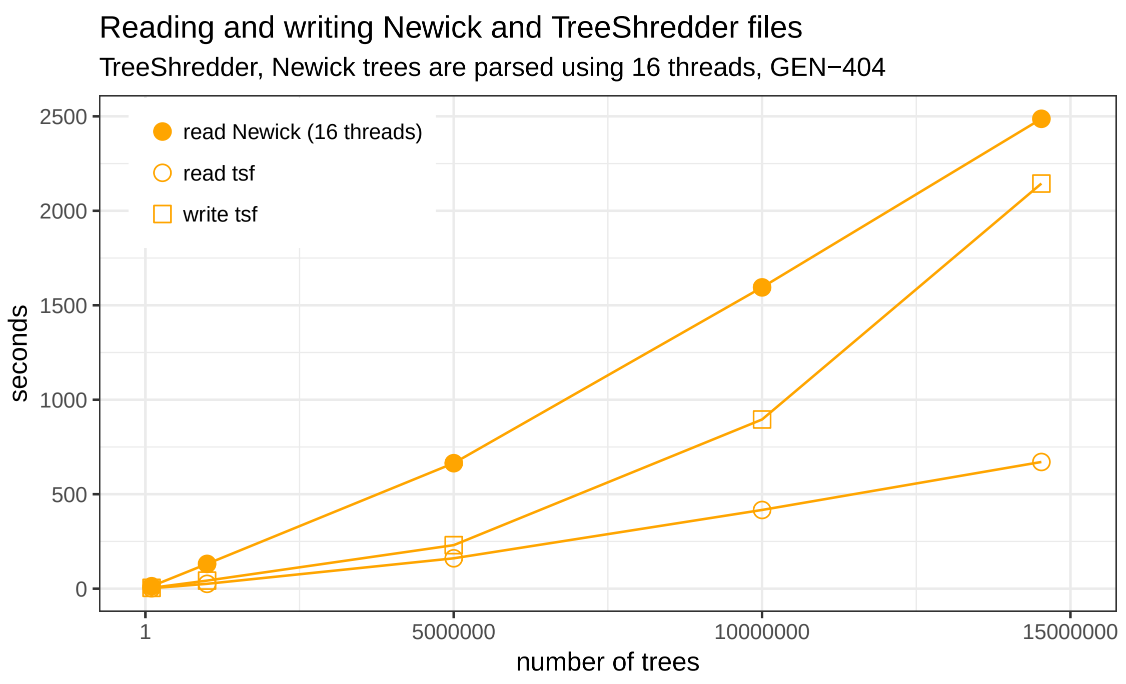
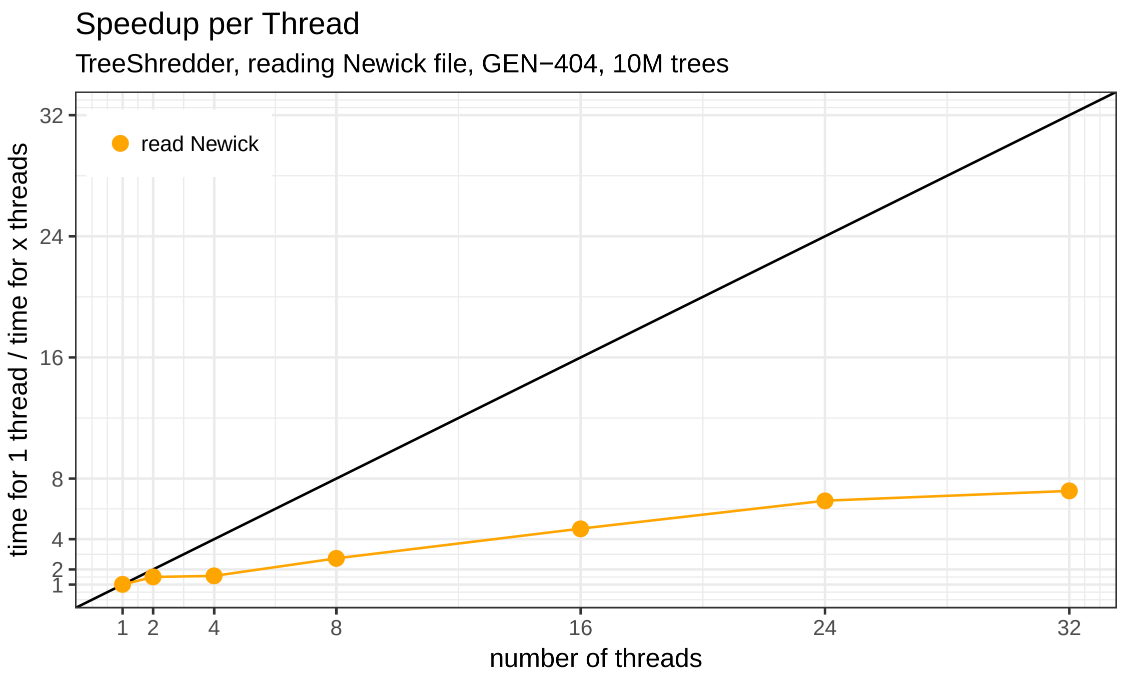


Figure 5.5: TreeShredder runtimes of reading Newick files, and reading and writing TreeShredder files for the ACO-225, PTP-116, PF1-367, and PF7-205 datasets: The plots show the runtime of (a) reading the Newick file and parsing the trees in serial, (b) reading the TreeShredder file, and (c) writing the compressed TreeShredder file.



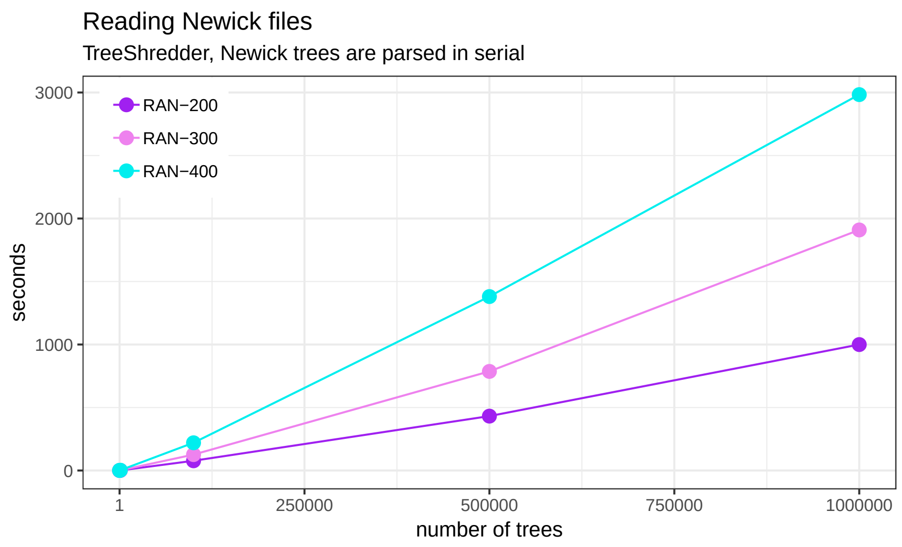
(a) TreeShredder runtimes of reading Newick files, and reading and writing TreeShredder files for the GEN-404 dataset: Newick trees were parsed in parallel using 16 threads. Still, reading a TreeShredder file is up to four times faster.



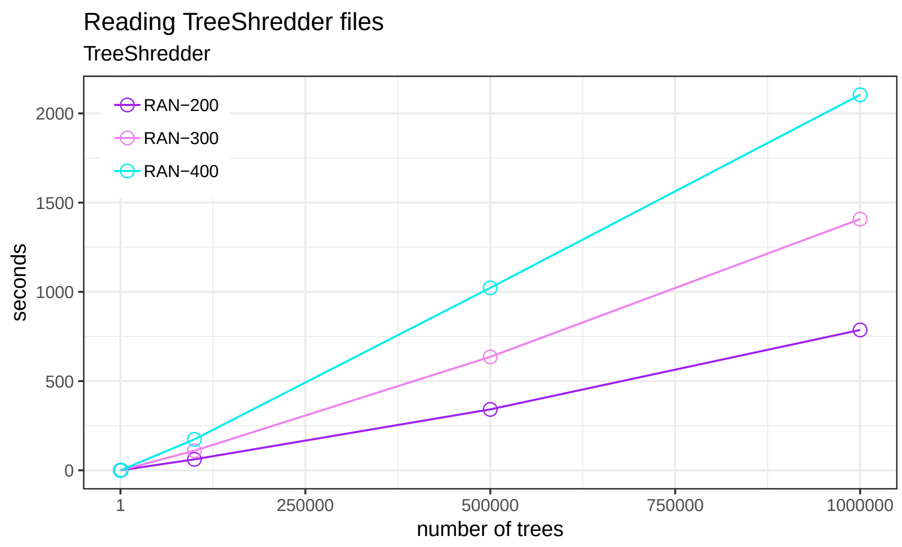
(b) TreeShredder speedup per thread of reading Newick file: The black line indicates perfect speedup. These benchmarks were run on 32 physical cores.

Figure 5.6: Comparing runtime and speedup for the GEN-404 dataset, reading and writing Newick and TreeShredder files.

(a)



(b)



(c)

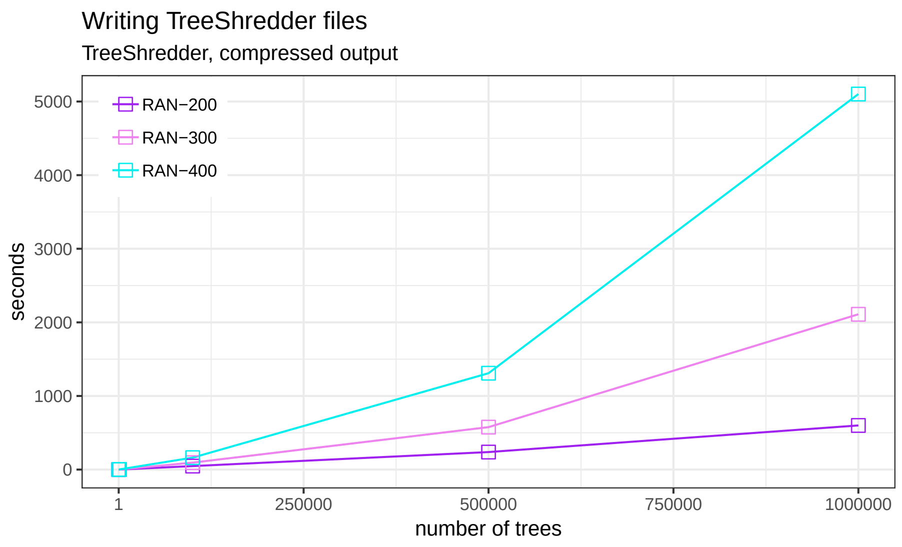


Figure 5.7: TreeShredder runtimes of reading Newick files, and reading and writing TreeShredder files for the RAN-200, RAN-300, and RAN-400 datasets: The plots show the runtime of (a) reading the Newick file and parsing the trees in serial, (b) reading the TreeShredder file, and (c) writing the compressed TreeShredder file.

case, the compressed files are about 3% the size of the uncompressed ones, which amounts to 530 MB vs 21 GB for 14.5 million GEN-404 trees (see Fig. 5.8b).

The real time is the wall clock time of reading the Newick trees and writing their information to a TreeShredder file. It can be seen that the difference in compressed and uncompressed real runtime is entirely due to the difference in writing the compressed or uncompressed file output. This is supported by the system time, which shows the same discrepancy.

Therefore, it is advised to compress file output (using the `-gz` flag) because it is faster than writing uncompressed output and comes with significant disk space savings (see also subsection 11.3.1).

5.7 Conclusion

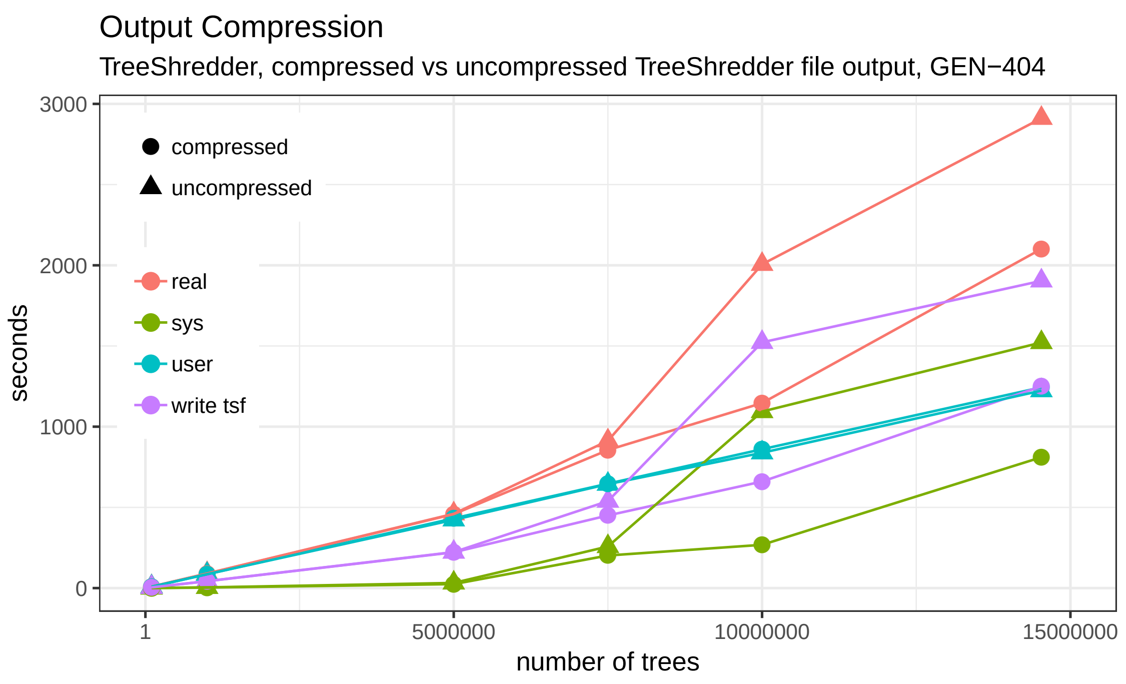
TreeShredder can read Newick and Nexus files, and parse the Newick tree strings from both formats in parallel. I extended the Splits Extraction Algorithm to also extract the node and edge information of nontrivial splits and to simultaneously determine the rootedness of the tree.

TreeShredder can read 15 million trees with about 400 taxa in 40 minutes using 16 threads. The TreeShredder file is a condensed version of a tree file containing tree information and speeds up reading that information 30-fold for 1,000 trees of more than 9000 taxa, which makes it the ideal starting point for different phylogenetic analyses with TreeShredder. Thus, if a dataset is used for many analyses, by once investing time in transforming a Newick file into a TreeShredder file, one can save time in subsequent analyses. This only pays off for datasets of sufficient size, e.g., those whose product of taxa and trees exceeds approximately one million. However, the inherent modifiability advantage of a TreeShredder file over a Newick file remains, independent of size.

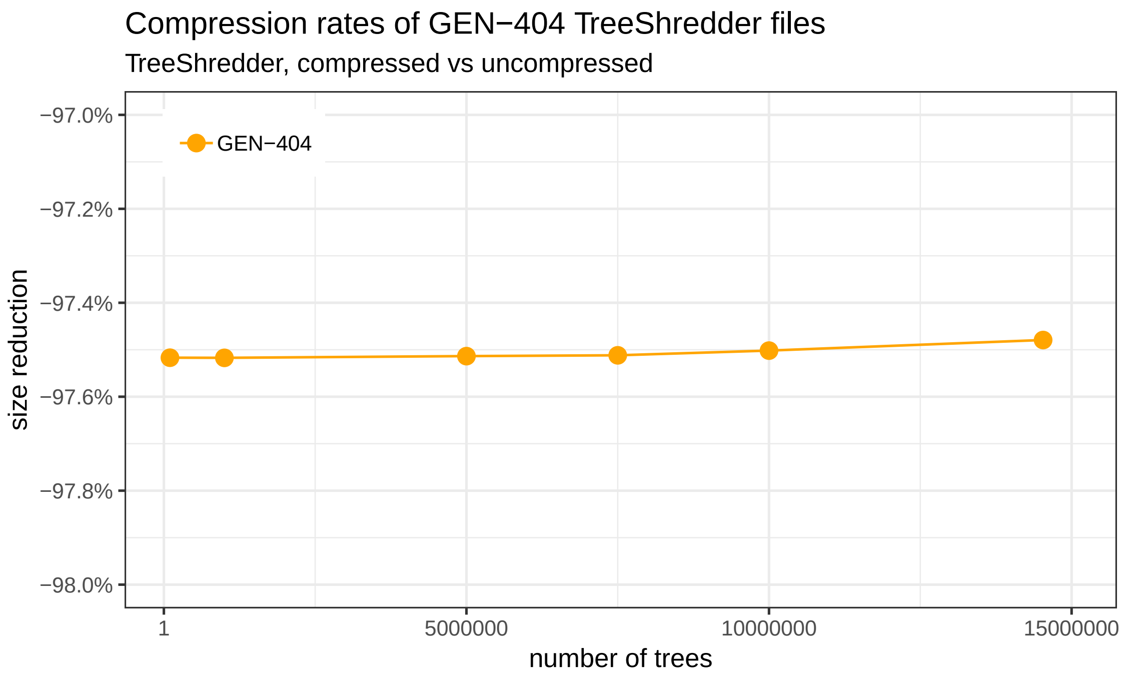
The payoff of converting Newick files to TreeShredder files for the datasets with random trees is much smaller than for datasets with typical trees. This is because the former have many more unique nontrivial splits than the latter (see Fig. 3.2), which greatly diminishes the potential of condensing the information of splits in TreeShredder files because each unique nontrivial split needs to be incorporated in the “Nontrivial Splits” block (see subsection 5.5.1).

The `-range` and `-merge` flags are useful when the researcher is only interested in a subset of the trees of a file or wants to merge the tree information of different files (of potentially different formats).

It was unexpected to find that output compression leads to faster writing times than uncompressed output. Combined with the benefit of significant disk space savings, there is no reason not to compress TreeShredder output on Linux.



(a) TreeShredder writing time of compressed and uncompressed TreeShredder file: "sys" is the CPU time spent executing system code (i.e., I/O calls) in kernel mode; "user" is the CPU time spent executing user code (i.e., program code) in user mode; "real" is the wall clock time (in this case the sum of sys and user time plus idle time) (Kerrisk, 2010). "write tsf" is the wall clock time of the TreeShredder file writing routine.



(b) TreeShredder compression rates of TreeShredder files: Compression rates are roughly constant.

Figure 5.8: TreeShredder runtime and efficacy of output compression

Part III

Methods in TreeShredder

6 Split Measures and Reference Trees

In many phylogenetic analyses, large sets of trees with the same set of taxa are used. For a robust analysis, the information of these trees then needs to be projected onto a single reference tree. To do this, there is a wide range of split measures to choose from. This chapter explains how to calculate and map these measures onto a reference tree.

6.1 Introduction

TreeShredder offers eight split measures, some of which are well-established, others I have newly developed for or implemented the first time in TreeShredder. The well-established measures are: absolute/relative occurrence of the split, Internode Certainty (Kobert et al., 2016), and Transfer Bootstrap Expectation (Lemoine et al., 2018).

For TreeShredder, I developed the absolute/relative occurrence of the split's best incompatible split and absolute/relative occurrence difference to the split's best incompatible split measures. Additionally, I generalized the Internode Certainty to the case where some splits from a reference tree may be missing from the set of trees that is to be mapped onto it.

6.1.1 Split occurrence

The absolute occurrence of a split, more commonly known as Felsenstein's Bootstrap supports (Felsenstein, 1985), measures how often a specific split is contained in a set S of trees. The relative occurrence of a split is the absolute occurrence normalized to the number of trees $|S|$ in the set. The absolute occurrence can be any natural number in $[0, |S|]$, the relative occurrence can be any rational number in $[0, 1]$.

This measure is widely used and gives an easy to interpret number or ratio of occurrence of a split in the trees.

6.1.2 Best incompatible split occurrence

In some cases, researchers may be more interested in the support of the most supported split that is incompatible with a given split (see section 7.1), rather than the support of the split itself. For this, I developed and implemented the best incompatible split occurrence measure. The absolute occurrence of the best incompatible split is the number of trees in a set S of trees that a given split's best incompatible split occurs in. The relative occurrence of the best incompatible split of a split is the absolute occurrence normalized to the number of trees $|S|$ in the set. The absolute occurrence can be any natural number in $[0, |S|]$, the relative occurrence can be any rational number in $[0, 1]$.

6.1.3 Occurrence difference to best incompatible split

Combining the information of split occurrence and best incompatible split occurrence gives a measure of difference between a split and its best incompatible split in the trees. This makes it possible to further differentiate between splits. A well-supported split A may have a highly-supported incompatible split in the trees, while another, less-supported split B may have no incompatible split at all. This difference could cause researchers to give more credence to split B than split A, despite it being less supported. Therefore, I developed and implemented the occurrence difference to the best incompatible split measure.

The absolute occurrence difference of a given split to its best incompatible split measures the difference in occurrence of those two splits in a set S of trees. The relative occurrence difference of a given split to its best incompatible split is the absolute difference normalized to the number of trees $|S|$ in the set.

Unlike the split occurrence and best incompatible split occurrence measures discussed above, the occurrence difference can be negative if the split's best incompatible split occurs more often in S than the split itself. The absolute occurrence difference can be any integer in $[-|S|, |S|]$. Accordingly, the relative occurrence difference can be any rational number in $[-1, 1]$.

6.1.4 Internode Certainty

An extension of the occurrence difference to best incompatible split (see subsection 6.1.3) using Shannon's measure of entropy (Shannon, 1948) is the Internode Certainty (IC, Salichos, Stamatakis, and Rokas, 2014). Slightly confusingly, "internode" means internal edges (or branches) in this case. The IC measures the extent of contradiction of a split B and the best supported incompatible split B^* in the *same* set of splits. In the following, I use the definitions of Salichos, Stamatakis, and Rokas (2014) and Kobert et al. (2016) and extend them in subsection 6.1.4.1 to apply them to reference trees.

Let x_B and x_{B^*} denote the relative frequencies of B and B^*, respectively, and the function f denote their support (i.e., occurrence):

$$x_B = \frac{f(B)}{f(B) + f(B^*)} \tag{6.1}$$

and

$$x_{B^*} = \frac{f(B^*)}{f(B) + f(B^*)} \tag{6.2}$$

with

$$x_B + x_{B^*} = 1.$$

Then, the Internode Certainty of split B (Salichos, Stamatakis, and Rokas, 2014) is defined as

$$IC(B) = 1 + x_B * log_2(x_B) + x_{B^*} * log_2(x_{B^*}) \tag{6.3}$$

if $x_B \geq x_{B^*}$ (i.e., $f(B) \geq f(B^*)$) and

$$IC(B) = -1 - x_B * log_2(x_B) - x_{B^*} * log_2(x_{B^*}) \tag{6.4}$$

if $x_B < x_{B^*}$ (i.e., $f(B) < f(B^*)$). For unexplained reasons, the latter case (equation 6.4) has been omitted in Kobert et al. (2016).

It can be seen that x_B and x_{B^*} fall in the ranges $(0, 1]$ and $[0, 1)$, respectively. This is because the absolute occurrence of B, $f(B)$, is at least equal to 1 because split B must necessarily occur at least once in the set of splits. Conversely, $f(B^*)$ is at least equal to 0 because there may not be a split that is incompatible with B in the set. In this case the above equations 6.3 and 6.4 are undefined because the term $x_{B^*} * log_2(x_{B^*})$ with $x_{B^*} = 0$ is undefined. However, this problem can be solved satisfactorily by defining $IC(B) := 1$ when no incompatible split is found. This is justified because the split B in question is not contradicted by any other split in the set and, thus, should be assigned the highest possible certainty score.

The sum of IC scores of the nontrivial splits of a tree is called Tree Certainty (TC; Salichos, Stamatakis, and Rokas, 2014). When comparing two trees, the TC can be used as a global measure for their respective incongruence (Salichos, Stamatakis, and Rokas, 2014).

6.1.4.1 Internode Certainty and Reference Trees

The above subsection 6.1.4 concerns IC scores when the splits and their incompatible splits belong to the same set of trees, which was a precondition in Salichos, Stamatakis, and Rokas (2014) and Kobert et al. (2016). However, when calculating IC scores for a reference tree whose splits may *not* exist in the set of trees, it is no longer guaranteed that the occurrence of split B is at least equal to 1. Split B, incompatible split B^*, or even both might be absent in the trees. Thus, in the following, I introduce a way to extend computing IC scores to the case of reference trees.

The relative frequencies x_B and x_{B^*} in equations 6.2 and 6.1 are undefined if splits B and B^* do not exist in the trees because the divisor $f(B) + f(B^*) = 0$. In the case of reference trees, therefore, relative frequencies are given by

$$^{ref}x_B = \begin{cases} x_B & \text{if } B \text{ exists} \\ 0 & \text{if } B^* \text{ exists, } B \text{ does not} \\ \dfrac{1}{2} & \text{if } B \text{ and } B^* \text{ do not exist} \end{cases}$$

and

$$^{ref}x_{B^*} = \begin{cases} x_{B^*} & \text{if } B \text{ exists} \\ 1 & \text{if } B^* \text{ exists, } B \text{ does not} \\ \dfrac{1}{2} & \text{if } B \text{ and } B^* \text{ do not exist} \end{cases}$$

with

$$^{ref}x_B +^{ref} x_{B^*} = 1$$

instead.

Analogously to the case where B exists but not B^*, equations 6.3 and 6.4 are also undefined if B^* exists but not B. Conversely, defining $IC(B) := -1$ solves this problem. This is justified because the split B in question is contradicted by an incompatible split B^* but does not itself exist in the set and, thus, should be assigned the lowest possible certainty score.

Finally, when neither split B nor B^* exist in the set, $IC(B)$ should be $= 0$. This is in accordance with all other cases where their absolute frequencies are equal: When $f(B) = f(B^*)$, then x_B and x_{B^*} are both equal to $\frac{1}{2}$ and equation 6.3 becomes

$$IC(B) = 1 + \frac{1}{2} * log_2(\frac{1}{2}) + \frac{1}{2} * log_2(\frac{1}{2}) = 0.$$

TreeShredder also computes the Tree Certainty score of a reference tree, which I define, analogously to Salichos, Stamatakis, and Rokas (2014), as the sum of IC scores of its nontrivial splits.

6.1.5 Transfer Bootstrap Expectation

Another measure is the Transfer Bootstrap Expectation (TBE), as suggested and studied by Lemoine et al. (2018). Their definitions and discussion are detailed in the following. Phylogenies combining thousands of taxa are becoming more common because of the increasing number of available sequences. This poses a problem for Felsenstein's Bootstrap Proportions (FBP) measure (Felsenstein, 1985), which is widely used to assess robustness or repeatability of inferred branches. In large phylogenies, FBP proportions tend to produce low values even if the dataset contains a lot of phylogenetic information.

FBP is based on resampling with replacement the sites of the original multiple sequence alignment to produce pseudo-alignments for replications. From these, trees are inferred using the same inference method that was used to infer the reference tree from the original alignment. The support of each branch of the reference tree is the proportion of identical branches in the resampled trees.

However, if the branches in the trees do not match the branch of the reference tree perfectly, they are counted as absent in FBP's strict binary presence/absence index. A difference of one single taxon in a set of thousands is enough for that index to be 0. Taxa of uncertain placement are also called "rogue" taxa. These taxa obtain diverse positions in the resampled trees and, unfortunately, there are good biological and computational reasons for their existence. They can be caused, among others, by evolutionary convergence, recombination, lack of conservation, and sequencing errors. When rogue taxa are found it is common to remove them and repeat the phylogenetic analysis, in the hope to obtain (higher) FBP values that better reflect the true underlying phylogenetic signal. This can be computationally expensive and is statistically questionable. Moreover, this does not help when the phylogenetic signal is weak, e.g., in alignments with a high number of sequences but a low number of sites. (Lemoine et al., 2018)

It is clear that FBP's all-or-nothing approach to measuring the exact presence of a branch in a tree should be replaced by a more gradual approach when the phylogeny is large. A new measure, called Transfer Bootstrap Expectation (TBE; Lemoine et al., 2018), applies a gradual function in the $[0, 1]$ range instead. Inferred branches are allowed to contain errors, they are not simply deemed correct or incorrect as in FBP. This helps reveal phylogenetic signal, especially in deep branches, where FBP would not due to its restrictive $0, 1$ binary presence/absence indicator function (Lemoine et al., 2018).

For this, the transfer distance $\delta(b, b^*)$ is used. It measures the distance between a branch b of the reference tree T and a branch b^* of a bootstrap tree T^* and counts the number of taxa that need to be transferred from one side to the other to convert one split into the other. The transfer index $\phi(b, T^*) = \min_{b^* \in T^*}\{\delta(b, b^*)\}$ measures the minimal transfer distance of branch b to any branch b^* in tree T^*. There are three important properties (Lemoine et al., 2018):

- $\phi(b, T^*) = 0$ iff b belongs to T^*.
- $\phi(b, T^*) \leq p - 1$ where p is the number of taxa on the smaller side of the split. This is because trivial splits with $p = 1$ are necessarily present in every tree.
- $\phi(b, T^*)/(p - 1)$ is close to 1 if T^* is large and random.

On the basis of these properties, Lemoine et al. (2018) define the Transfer Bootstrap Expectation as follows:

$$\text{TBE}(b) = 1 - \frac{\overline{\phi(b, T^*)}}{p - 1} \tag{6.5}$$

where $\overline{\phi(b, T^*)}$ is the average minimal transfer distance among all bootstrap trees. TBE is close to 0 if the bootstrap trees are large and random, and therefore do not contain any signal regarding b. Conversely, TBE is 1 if b occurs in all trees. TBE is guaranteed to be larger than FBP or equal if b is a cherry (Lemoine et al., 2018).

TBE suffers from some of the same limitations as FBP, e.g., the assumption of site independence does not hold. Neither measure should be interpreted as a probability that the inferred branch is correct, but rather be thought of in terms of repeatability: Does the alignment, even after random perturbations, really contain the phylogenetic signal implied by a given branch?

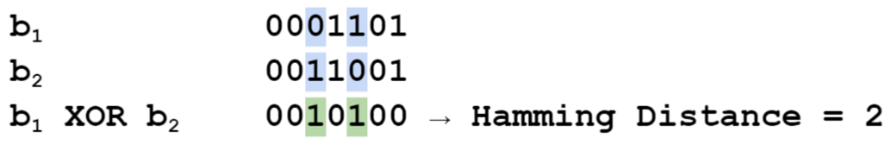
6.1.5.1 Instability Score

Additionally, calculating the transfer index $\phi(b, T^*)$ gives an instability score for each taxon by counting how often that taxon would need to be transferred to convert split b into split b^* with minimal transfer distance $\delta(b, b^*)$ (Lemoine et al., 2018). This score helps identify rogue taxa, which could then be removed from the phylogeny, or studied to find the reason for their unstable phylogenetic position. TreeShredder writes these instability scores to a separate file.

6.2 Methods

6.2.1 Implementation of the TBE Algorithm in TreeShredder

To calculate the Transfer Bootstrap Expectation, I developed and implemented the following approach in TreeShredder. Let n be the taxon set size of a split, p the



```
b1              0001101
b2              0011001
b1 XOR b2       0010100 → Hamming Distance = 2
```

Figure 6.1: Example of Hamming Distance calculation: Pairs of bits with opposite value are highlighted in blue. The Hamming Distance is equal to the number of bits = 1 (highlighted in green) resulting from a bitwise XOR operation.

number of taxa on the smaller side of the split (= 1 in trivial splits, ≥ 2 in nontrivial splits), t the number of bootstrap trees and s the number of bootstrap trees containing that split. Then the transfer distance (TD) of two splits can quickly be computed using the Hamming Distance (HD) between their bitsets b_1 and b_2. The XOR set operator of `dynamic_bitset` applies a logical XOR to each corresponding pair of bits. Counting the resulting bits that are equal to 1 obtains the Hamming Distance. Fig. 6.1 shows an example of two splits with Hamming Distance = 2, which corresponds to a transfer distance $\delta(b_1, b_2) = 2$, because 2 taxa have to be transferred to the other side of the split, i.e., the bits have to be flipped ($0 \to 1$ or $1 \to 0$).

By calculating the Hamming Distance between a given split and every split in a bootstrap tree, the split with minimal transfer distance in each tree can be found. Additionally, I developed shortcuts to achieve speedup. A lower bound for the Hamming Distance between two splits is given by

$$\mathrm{HD}_{lower\ bound} = min(|p_1 - p_2|, n - p_1 - p_2) \leq \mathrm{HD}_{actual}$$

where p_1 and p_2 are the number of taxa on the smaller side of the splits b_1 and b_2, respectively. Furthermore, each split has a maximum transfer distance of $p - 1$, which is imposed by the guaranteed existence of a trivial split with $p = 1$ in every tree. Thus, the actual transfer distance of a split (either to a trivial or to a nontrivial split) is

$$\mathrm{TD}_{actual} \leq p - 1 = \mathrm{TD}_{maximum}.$$

Thus, if $\mathrm{HD}_{lower\ bound}$ is *not* smaller than $\mathrm{TD}_{maximal}$ no set operation need be done. This is because HD_{actual} between those two splits would then necessarily be equal to or greater than $\mathrm{TD}_{maximal}$, a limit imposed by trivial splits.

For splits with $p = 2$ (i.e., cherries) the transfer distance is at most 1 because only 1 taxon need be transferred to obtain a trivial split, which is in the tree by definition. The instability score of both taxa in the cherry is then increased by a weighted score of $a(p-1)/p = a/2$, where a is the number of bootstrap trees not containing that split, since either taxon could be transferred to obtain a trivial split. Analogously, when $\mathrm{TD}_{actual} = \mathrm{TD}_{maximal}$ the instability score of each taxon on the smaller side of the split is increased by a weighted average of $(p-1)/p$.

Only for splits where $\mathrm{TD}_{actual} < \mathrm{TD}_{maximal}$ the taxa which need to be transferred must be determined through an actual set operation, and their instability score increased. This is because only then the split with minimal transfer distance must necessarily be a nontrivial split and the guaranteed existence of trivial splits in the trees cannot be used as a shortcut.

6.2.2 Mapping Split Measures onto Reference Trees

It seems straightforward to map computed split measures onto user-given reference trees, however, a number of pitfalls exist. In particular, mapping the support information of a set of splits M onto a reference tree is not trivial. Some difficulties have to be dealt with:

- One or more splits of the reference tree may not exist in M. This may cause problems in the calculation of the Internode Certainty (see subsection 6.1.4).
- The node information of the splits of M may not be usable because it is not clear to which of the two nodes connected by the edge it belongs. The node information of the reference tree splits can be used, however, because the directionality induced by the reference tree root is preserved.
- If the reference tree is rooted the two root split arms need to be preserved. Furthermore, the branch length of the equivalent split in M (if it exists) that is to be mapped onto the two root split arms is halved. Other information of that split is mapped onto both root split arms unprocessed.

While parsing the original reference tree Newick string (see section 5.2) the reference tree routine simultaneously builds a new Newick string that contains, both, the original and the mapped information. This new tree is not only topologically identical to the reference tree, in addition, its Newick string is arranged identically (same taxon order) as the Newick string of the reference tree.

6.2.2.1 IQ-TREE format

If several information values are stated for an edge, the software IQ-TREE (L.-T. Nguyen et al., 2015) outputs these values separated by slashes (e.g., “0.99/12/0.5”) directly following a closing parenthesis. This is a problem for some tree visualization programs (e.g., FigTree, Rambaut, 2010) which cannot interpret that information as separate numbers.

The `-iqtreeNames` flag option causes TreeShredder to read the numbers that are in IQ-TREE format separately and store them with custom-specified keys as edge metacomments (see subsections 3.1.5 and 5.5.2) when parsing Newick strings. Using this flag and mapping a Newick tree that has IQ-TREE format onto itself converts it to a Newick tree with identical information but a more widely readable format.

6.3 Results and Discussion

6.3.1 TreeShredder split measures

Figs. 6.2, 6.3, 6.4, and 6.5 show the relative support of unique splits of 1,000 POL-9147 trees plotted against the relative best incompatible support, the relative

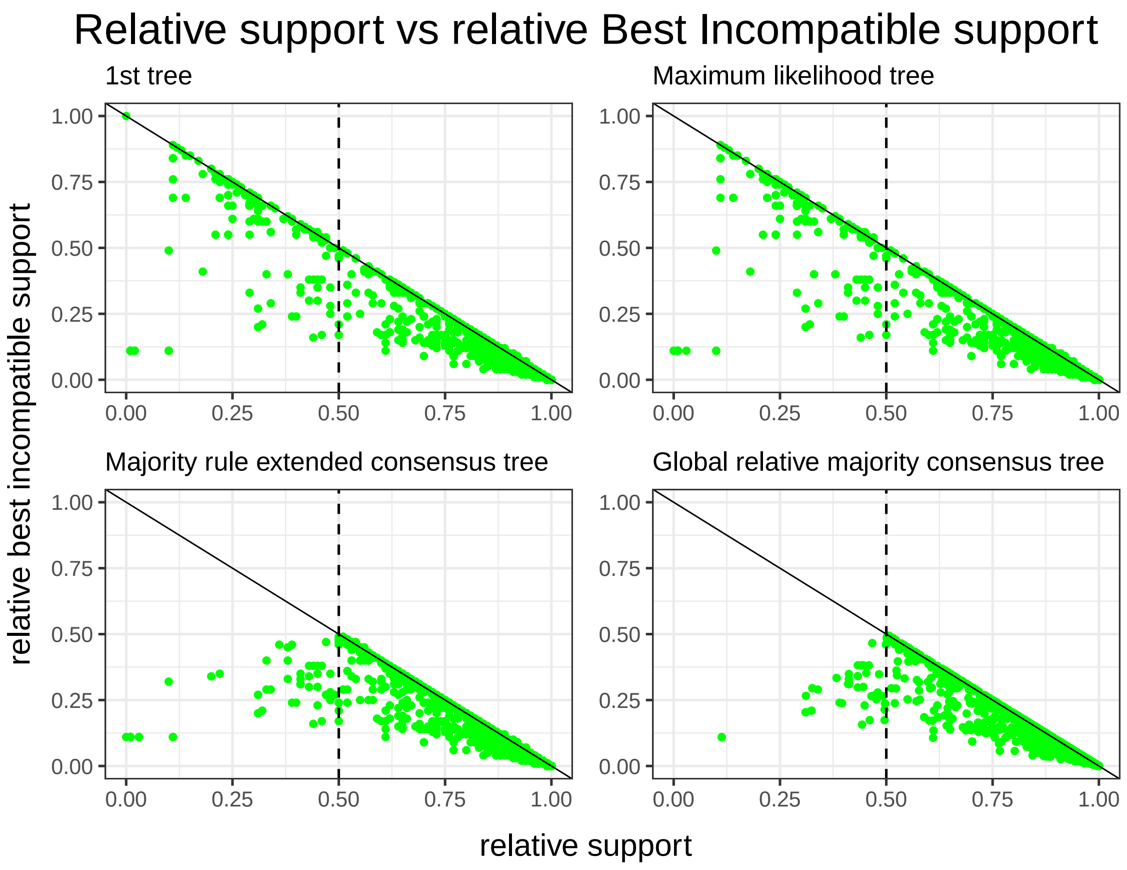


Figure 6.2: Comparison of relative support and relative best incompatible support measures of four POL-9147 trees: Data points to the right of the vertical line in the consensus tree plots indicate the 50% majority consensus tree.

difference to the best incompatible support, the IC, and the TBE measures, respectively, for the 1^{st} tree in the file, the maximum likelihood tree, the majority rule consensus tree, and the global relative majority consensus tree. For an explanation of consensus trees, see chapter 8. The 1^{st} tree in the file was chosen to represent any arbitrary tree of the file. Throughout, the scatter plots for the 1^{st} and the maximum likelihood trees are similar, differing only slightly in splits with low support.

Fig. 6.2 shows the relative support plotted against the relative best incompatible support for the four POL-9147 trees mentioned above. Because two incompatible splits cannot occur on the same tree, their sum of support is $< 100\%$. This explains why data points occur only below the diagonal line. By sliding the vertical line to the right in the consensus tree plots, one can manually create the plots for the corresponding majority consensus tree with higher threshold. One can see that the majority rule extended and the global relative majority consensus trees both contain splits with $< 50\%$ support. Interestingly, these splits' best incompatible splits tend to have low support too.

Fig. 6.3 shows the relative support plotted against the difference to the relative best incompatible support for the four POL-9147 trees mentioned above. Data points below the horizontal line indicate that the split has lower support than its best incompatible split. Because the global relative majority consensus tree contains

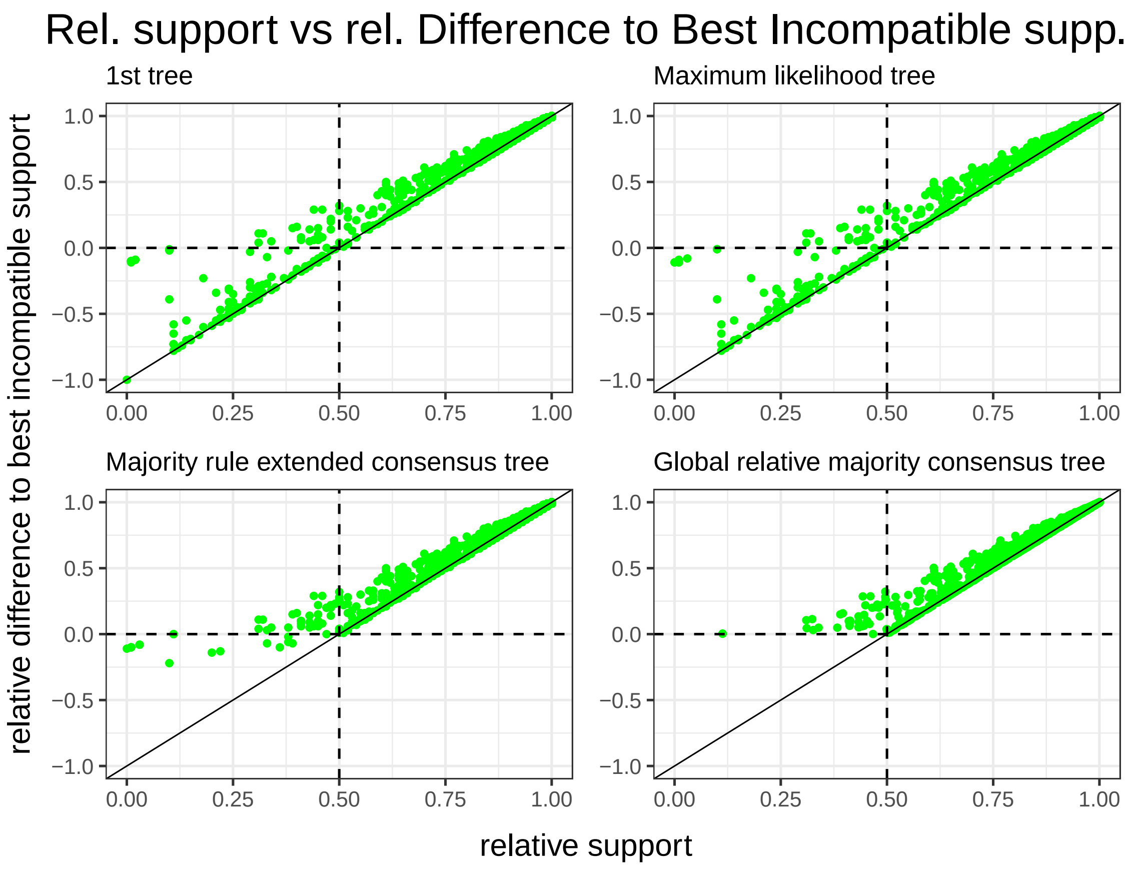


Figure 6.3: Comparison of relative support and relative difference to best incompatible support measures of four POL-9147 trees: Data points to the right of the vertical line in the consensus tree plots indicate the 50% majority consensus tree.

only splits that are compatible with *all* splits with higher support in the trees, there are no splits below the horizontal line. This is not true of the majority rule extended consensus tree, where there are some data points (slightly) below the horizontal line.

Fig. 6.4 shows the relative support plotted against the Internode Certainty for the four POL-9147 trees mentioned above. Here too, because the global relative majority consensus tree contains only splits that are compatible with *all* splits with higher support in the trees, there are no splits below the horizontal line. The Internode Certainty is only negative if the split occurs less often than its best incompatible split. The majority rule extended consensus tree, however, does contain splits with negative Internode Certainty.

Fig. 6.5 shows the relative support plotted against the Transfer Bootstrap Expectation for the four POL-9147 trees mentioned above. Because a split's TBE value is equal to or greater than its support, data points are only at or above the diagonal line. One can see that splits with low relative support can have high TBE support. This is demonstrated in the first plot, where the two data points colored in red are splits with, both, identical relative support (= 0.02) and identical average minimal transfer distance (= 1.76) but very different TBE. This is because the smaller side of the split with the lower TBE is smaller than the smaller side of split with higher

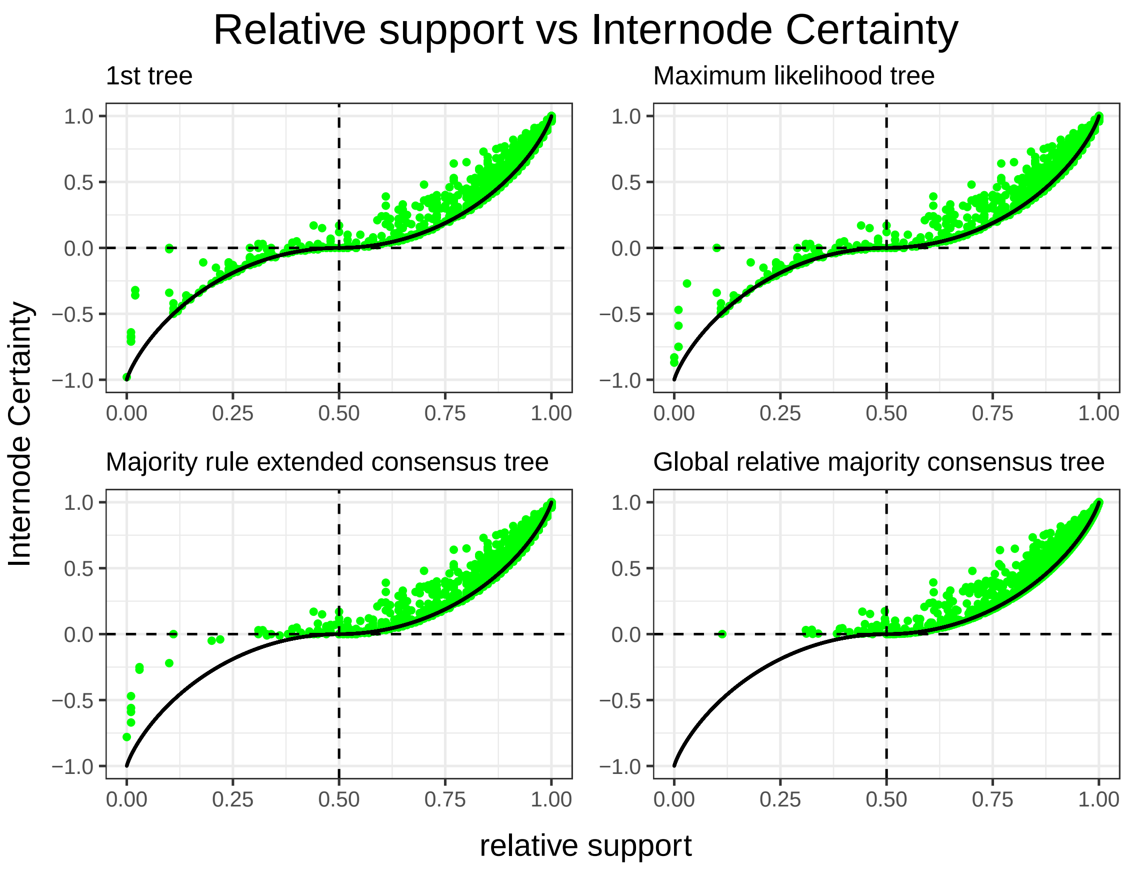


Figure 6.4: Comparison of relative support and Internode Certainty measures of four POL-9147 trees: Data points to the right of the vertical line in the consensus tree plots indicate the 50% majority consensus tree.

TBE ($p = 3$ and 9, respectively; see equation 6.5).

Figs. 6.7 and 6.8 show the effect that increasing subtree sizes (up to 256 taxa) have on TBE. The subtrees have the prefixes `LT` and `RT` for left and right subtree, respectively. The different taxa in the subtrees are distinguished by a binary postfix. Each tree represents 1 of 3 similar trees that differ only in the position of the rogue taxa `x1` and `x2` (see Fig. 6.6): The topology seen here, one where `x1` and `x2` are swapped, and one where `x1` and `x2` form a cherry. These three trees' information was then mapped onto the reference trees.

In Fig. 6.7 it can be seen that TBE increases from ≈ 0.33, ≈ 0.67, ≈ 0.83, to ≈ 1 when *both* subtrees increase from 1, 2, 4, to 256 taxa. Conversely, the relative support is ≈ 0.33 for all four splits, irrespective of subtree sizes.

In Fig. 6.8 it can be seen that TBE remains at ≈ 0.33 when only *one* subtree increases from 1, 2, 4, to 256 taxa and the other consists of 1 taxon. Similarly, the relative support is ≈ 0.33 for all four splits, irrespective of subtree sizes.

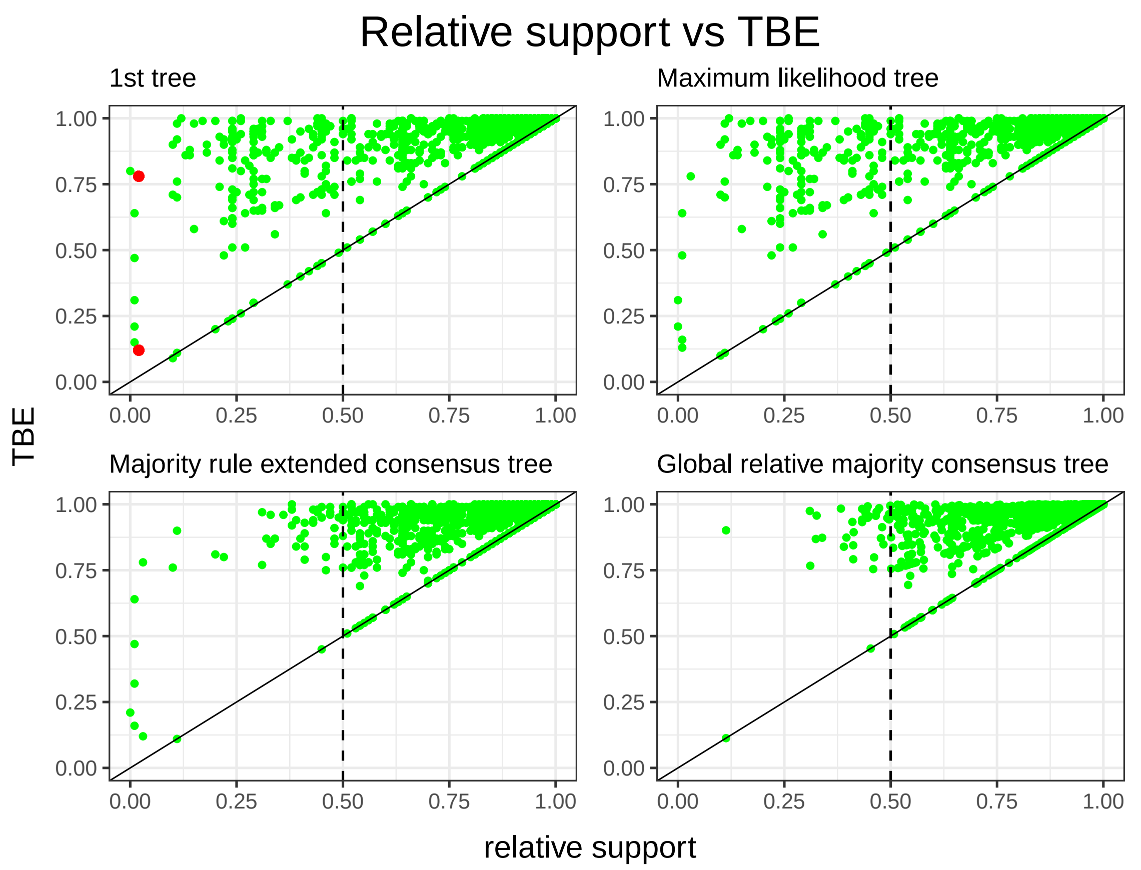


Figure 6.5: Comparison of relative support and Transfer Bootstrap Expectation measures of four POL-9147 trees: Data points to the right of the vertical line in the consensus tree plots indicate the 50% majority consensus tree. The two red dots in the 1st tree's plot are highlighted as an example where two splits can have identical relative support and identical average minimal transfer distance (not shown in the plot) but very different TBE.

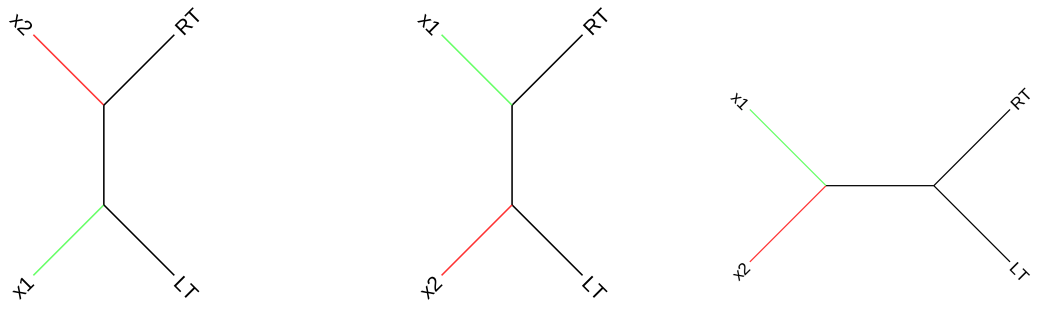
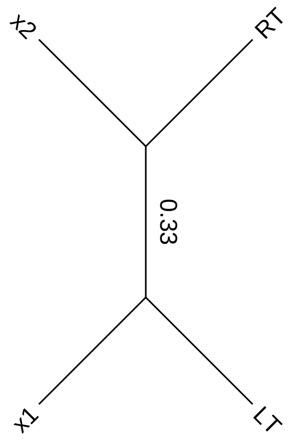
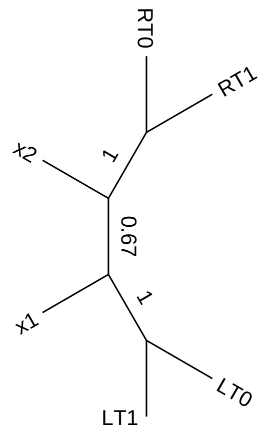


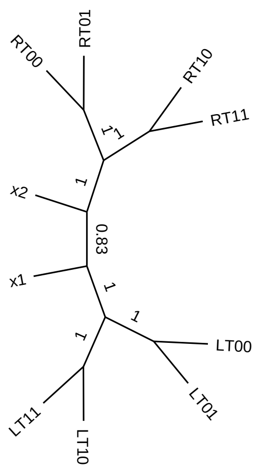
Figure 6.6: Three trees where rogue taxa change position: The branch with `x1` is colored in green, the one with `x2` in red. Mapping these three trees onto the first one produces the tree and TBE value of the first tree in Figs. 6.7 and 6.8. By only increasing the subtree sizes of `RT` or both, `RT` and `LT`, the other trees and their TBE values depicted in Figs. 6.7 and 6.8 can be produced in the same way.



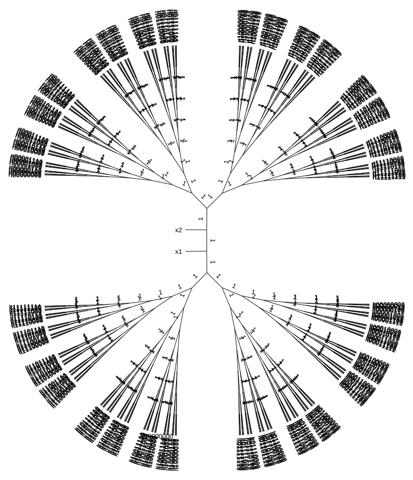
(a) Subtrees consist of 1 taxon: TBE ≈ 0.33



(b) Subtrees consist of 2 taxa: TBE ≈ 0.67

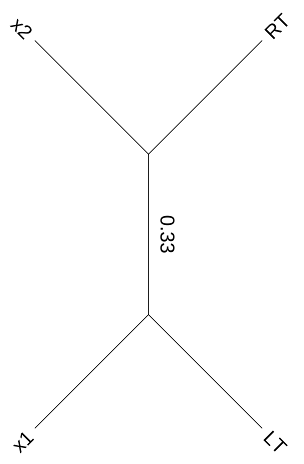


(c) Subtrees consist of 4 taxa: TBE ≈ 0.83

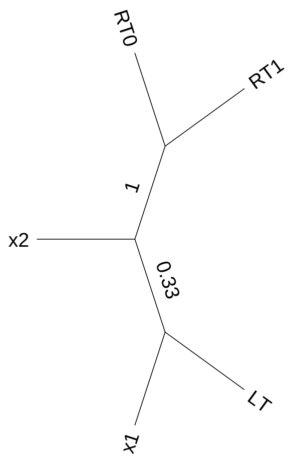


(d) Subtrees consist of 256 taxa: TBE ≈ 1

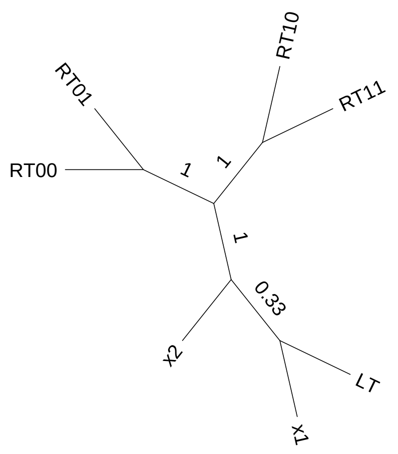
Figure 6.7: Effect on TBE of two subtrees increasing in size: The left and right subtrees comprise taxa with the prefixes `LT` and `RT`, respectively. Both subtrees increase in size. For each of these four trees, there are three trees in total (see Fig. 6.6): one where `x1` and `x2` have the positions seen here, one where they are swapped, and one where they form a cherry. The three trees are mapped onto the (reference) trees seen here.



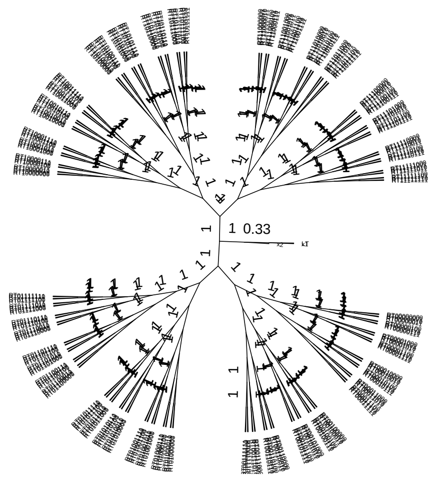
(a) Right subtree consists of 1 taxon: TBE ≈ 0.33



(b) Right subtree consists of 2 taxa: TBE ≈ 0.33



(c) Right subtree consists of 4 taxa: TBE ≈ 0.33



(d) Right subtree consists of 256 taxa: TBE ≈ 0.33

Figure 6.8: Effect on TBE of one subtree increasing in size: The left and right subtrees comprise taxa with the prefixes `LT` and `RT`, respectively. Only the right subtree increases in size, the left subtree consists of 1 taxon. For each of these four trees, there are three trees in total (see Fig. 6.6): one where `x1` and `x2` have the positions seen here, one where they are swapped, and one where they form a cherry. The three trees are mapped onto the (reference) trees seen here.

6.3.2 TBE calculation runtime

Fig. 6.9a shows TreeShredder's TBE calculation runtime for the 50% majority rule consensus tree (M_{50}, see chapter 8 for definitions of consensus trees). The runtime increases linearly with the number of trees.

Fig. 6.9b shows TreeShredder's TBE calculation speedup per thread for datasets with 1,000 trees. POL-9147 and CRF-2696 speedup increases more slowly beyond 32 threads. WNT-82 speedup is considerably worse and decreases beyond 24 threads. This is because the computational problem is not big enough to offset the parallelization overhead.

Figs. 6.10a and 6.10b compare TBE runtimes of BOOSTER, RAxML, and TreeShredder for CRF-2696, and POL-9147 using 8 threads, respectively. (The WNT-82 dataset was intentionally left out because the runtime was too low to be usefully compared.) Runtimes increase linearly with the number of trees (note the log-scales). RAxML started from a Newick file is fastest for the big datasets CRF-2696 and POL-9147. However, TreeShredder started from a TreeShredder file comes a close second, taking only 2 minutes compared to RAxML's 1 minute to finish POL-9147 with 1,000 trees. As expected, TreeShredder started from a TreeShredder file is faster than started from a Newick file because no costly parsing of Newick trees occurs. BOOSTER started from a Newick file takes 10 times longer than RAxML and TreeShredder for CRF-2696, and up to a 370 and 160 times longer, respectively, for POL-9147, taking 5 hours to finish 1,000 trees. Runtimes for WNT-82 are short overall.

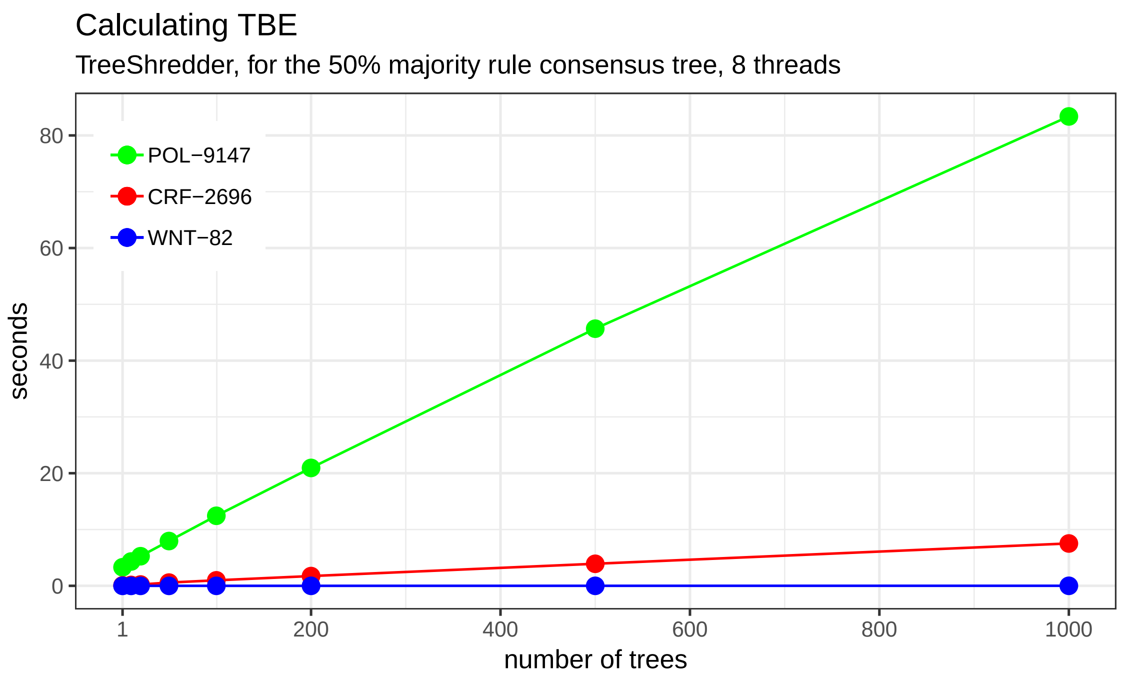
6.4 Conclusion

TreeShredder provides eight split measures, including the occurrence of the split's best incompatible split, difference in occurrence between a split and its best incompatible split, Internode Certainty, and Transfer Bootstrap Expectation.

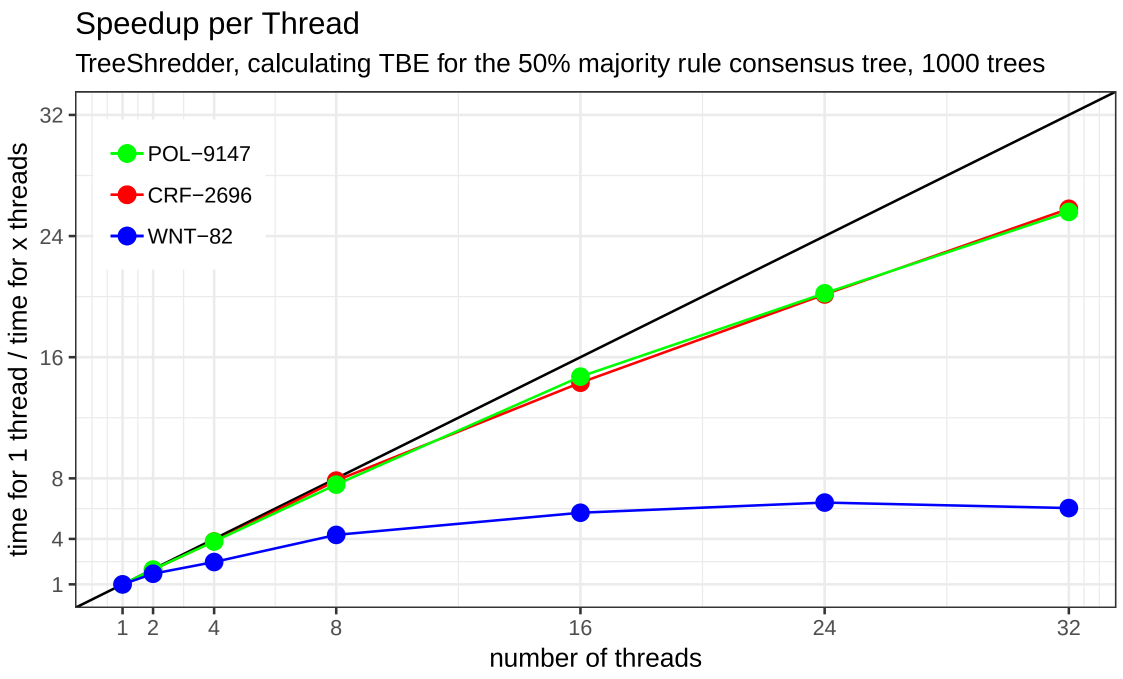
I extended IC to be applicable for reference trees and produce reasonable results when the reference tree contains splits that do not occur in the trees that are to be mapped onto it. While IC is useful to measure the degree of conflict between a split and its best incompatible split (Salichos, Stamatakis, and Rokas, 2014), the absolute quantity of how often each split occurred, and as a consequence, the absolute difference between the splits' occurrences is lost. This is because IC uses the *relative* occurrences of the splits (Kobert et al., 2016). Therefore, two splits with similar IC can have very different absolute occurrences.

The relative difference to best incompatible split measure, while simpler than IC, has this problem too. Two splits with different support can have equal relative difference to best incompatible split. However, the interpretation is more straight forward than IC: the relative difference to best incompatible split tells how much more or less often a split's best incompatible split occurs in the trees.

The relative best incompatible support, difference to relative best relative incompatible support, and Internode Certainty measures offer additional tools to assess the validity of splits. E.g., two splits with similar relative support can differ substantially in these three measures. Also, a split with low support that has a positive

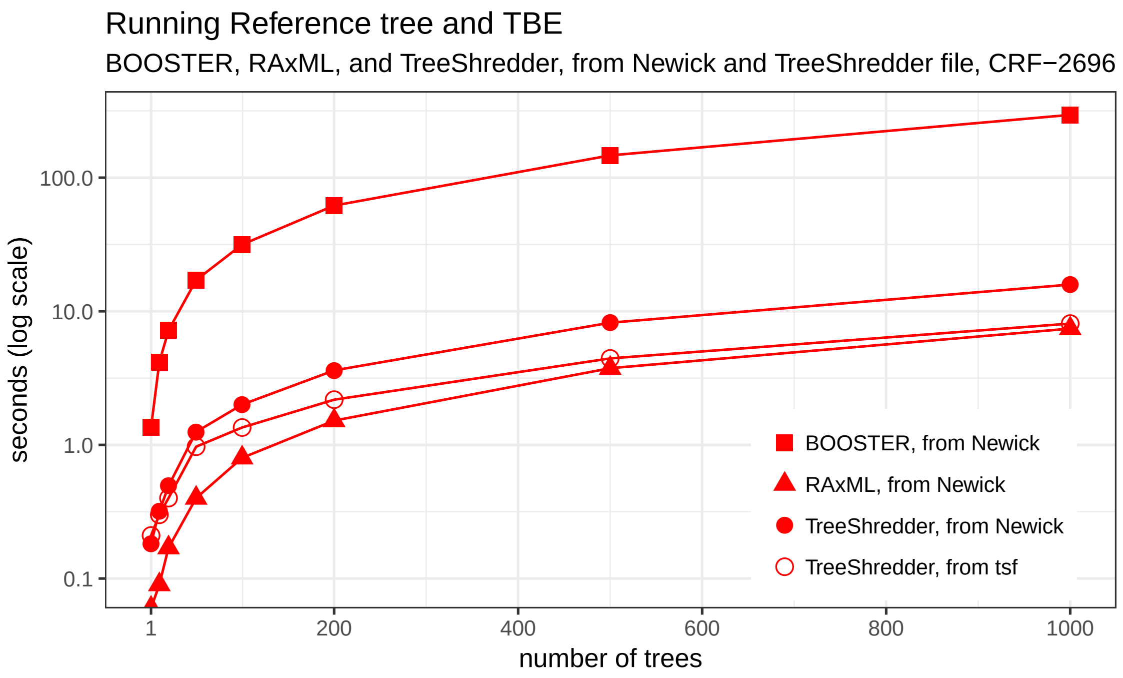


(a) TreeShredder TBE calculation runtime: TBE calculation time increases linearly with the number of trees. (This plot shows the calculation time of TBE only, it excludes the time it takes to generate the consensus tree, which is a few seconds at most; see subsection 8.3.)

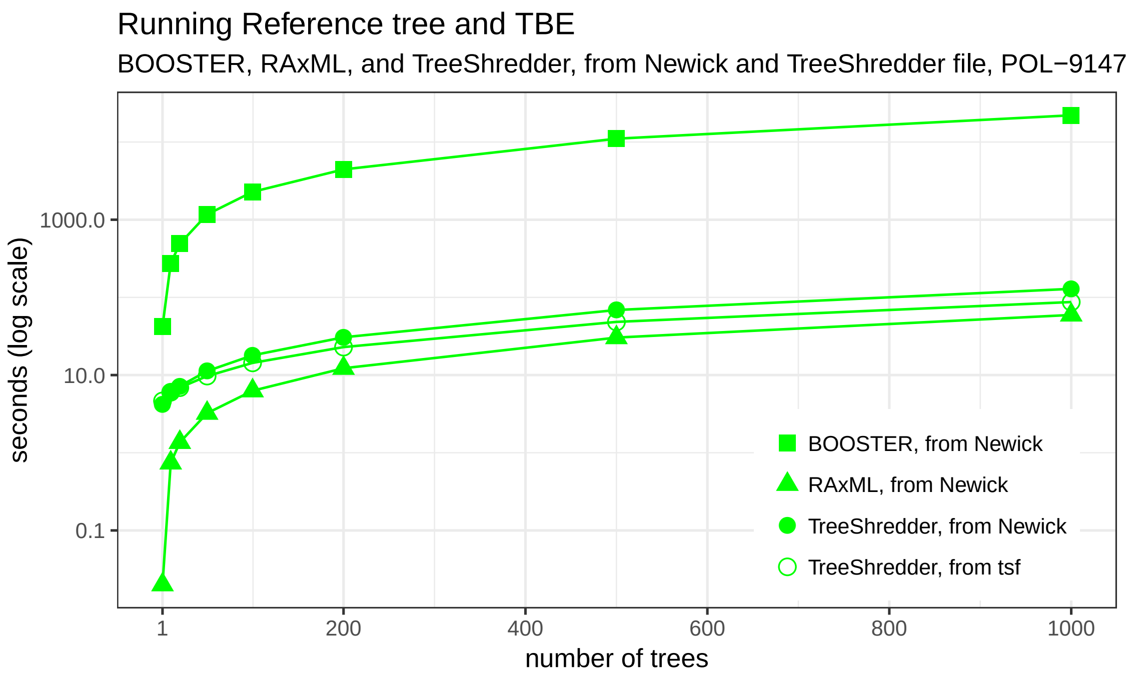


(b) TreeShredder TBE calculation speedup per thread: The black line indicates perfect speedup. POL-9147 and CRF-2696 speedup is good. WNT-82 speedup is weak. These benchmarks were run on 32 physical cores.

Figure 6.9: TreeShredder TBE calculation runtime and speedup



(a) Runtimes for reference tree and TBE calculation of BOOSTER, RAxML, and TreeShredder from Newick and TreeShredder file, CRF-2696: The y-axis is log-scale. Runtimes increase linearly with the number of trees. BOOSTER is the slowest by an order of magnitude.



(b) Runtimes for reference tree and TBE calculation of BOOSTER, RAxML, and TreeShredder from Newick and TreeShredder file, POL-9147: The y-axis is log-scale. Runtimes increase linearly with the number of trees. BOOSTER is the slowest by 2 orders of magnitude, taking up to several hours to finish for large datasets.

Figure 6.10: TBE calculation runtime comparison of BOOSTER, RAxML, and TreeShredder

difference to best incompatible support or IC can appear better than a split with higher support that has negative difference to best incompatible support or IC.

TBE uses a gradual function to measure the presence of a branch in a tree. As can be seen in Fig. 6.5, this generally leads to higher values than the simple relative support measure, even for splits with low support. This is because there are many similar splits in the trees, that differ only slightly from the split in question and, thus, have a low transfer distance, which leads to high TBE values. This effect is stronger the bigger the splits are, as shown in Fig. 6.7. The reason is that, even though the average transfer distance between splits and their respective nearest splits is equal for all four trees, that average transfer distance is normalized by $p-1$ in the calculation of TBE, where p is the number of taxa on the smaller side of the split (see subsection 6.1.5). If p is held constant as shown in Fig. 6.8, where only one of the two subtrees increases in size, then TBE is constant as well. TreeShredder's TBE algorithm offers parallelism and takes about 100 seconds to calculate the TBE values for 1,000 input trees with more than 9000 taxa when using 8 threads. Calculating TBE is fastest in RAxML and TreeShredder, with BOOSTER taking 100 times longer.

When mapping split information onto reference trees, the root split must be treated in a special way and the node information of the input trees cannot be used. TreeShredder can transform IQ-TREE format to the widely-readable Nexus metacomment format.

7 Incomplete Splits and Congruency

7.1 Introduction

An incomplete split represents a group of splits which only differ in some ambiguous taxa that can occur on either side of the split. Thus, an incomplete split is a convenient way to make queries on a set of trees that, e.g., only concerns a subset of taxa. In some analyses, there is interest in the support of the separation of certain taxa irrespective of some other (maybe rogue) taxa. To this end, one needs to check incomplete splits that contain all taxa of interest, while excluding all those one does not care about.

Here I define incomplete splits, i.e., splits where some taxa may occur on either side of the split. The general split notations to denote complete splits can be extended to denote the ambiguous taxa. In the "taxon notation" format, a vertical line '`|`' separates one side from the other and a question mark '`?`' separates a side from the ambiguous taxa. An incomplete split string may have one or both partition characters (but the '`|`' must occur first) because the missing subset can be deduced from the other two. Taxon names and partition characters may be separated by a space '` `' or a comma '`,`'. The "bit notation" format is similar to the splits denoted by bits, however, the ambiguous taxa are replaced by question marks '`?`'. Note that in this format, the order of the bits represent the taxa in reverse order. Fig. 7.1 shows the three "taxon notation" formats and the "bit notation" format accepted for an example incomplete split and its implied complete splits. TreeShredder can process queries with incomplete splits provided in a Split file (see section 5.4).

To determine if two complete splits are congruent, the intersection method is used (see Bryant, 2003, and references therein). Two complete splits A and B, with the sides $a_1|a_2$ and $b_1|b_2$, respectively, are incongruent iff none of the four possible intersections of their sides are empty:

$$a_1 \cap b_1 \neq \emptyset \ \wedge \ a_1 \cap b_2 \neq \emptyset \ \wedge \ a_2 \cap b_1 \neq \emptyset \ \wedge \ a_2 \cap b_2 \neq \emptyset.$$

Conversely, A and B are congruent if one or two intersections are empty – the latter being the case if the splits are identical (see Tab. 7.1).

An incomplete split implies 2^m complete splits, where m is the number of ambiguous taxa, because they can occur on either of two sides. (It will not have escaped my reader's notice that an incomplete split with $m = 0$, where there are no ambiguous taxa, is, in fact, a complete one.) This shows that the number of implied complete splits can quickly grow prohibitive. Simply repeating, e.g., compatibility checks with all complete splits implied by an incomplete one is not feasible. Therefore, the above method of determining congruency of two complete splits must be extended to the general case of a complete and an incomplete split. For this, the notation of an induced subsplit C restricted by I, which is the split that remains when restricting

```
The three "taxon notation" formats:

          A B | C D E ? F G
          A B | C D E
          A B ? F G

The "bit notation" format:

          ??00011

The implied complete splits:

          A B F G | C D E
          A B F | C D E G
          A B G | C D E F
          A B | C D E F G
```

Figure 7.1: Examples of the formats used to represent an incomplete split and its implied complete splits: In the incomplete split, taxa A and B occur on one side of the split and taxa C, D, and E on the other side. Taxa F and G can occur on either side and are highlighted in yellow. The incomplete split is congruent with each of its four implied complete splits.

Table 7.1: Example of the intersection method for complete and incomplete splits: "C+S" colored in green means the splits are congruent and supportive. "C-S" colored in green means the splits are congruent but not supportive. "I" colored in red means the splits are incongruent.

Complete splits:

A $(a_1\|a_2)$:	B $(b_1\|b_2)$:			
1234|567	123|4567	1|234567	1234|567	1235|467
$a_1 \cap b_1$	123	1	1234	123
$a_1 \cap b_2$	4	234	∅	4
$a_2 \cap b_1$	∅	∅	∅	5
$a_2 \cap b_2$	567	567	567	67
	C+S	C-S	C-S	I

Complete and incomplete splits:

C $(c_1\|c_2)$:	I $(i_1\|i_2)$:			
1234|567	12|56	12|35	12|34	15|26
$c_1 \cap i_1$	12	12	12	1
$c_1 \cap i_2$	∅	3	34	2
$c_2 \cap i_1$	∅	∅	∅	5
$c_2 \cap i_2$	56	5	∅	6
	C+S	C-S	C-S	I

a complete split to the set of unambiguous taxa of an incomplete split, is required. Let C be a complete split with the sides $c_1|c_2$ and I be an incomplete split with the sides $i_1|i_2$, then the four intersections are:

$$\begin{aligned} c_1 \text{ restricted by } I \cap i_1 &= c_1 \cap i_1 \\ c_2 \text{ restricted by } I \cap i_1 &= c_2 \cap i_1 \\ c_1 \text{ restricted by } I \cap i_2 &= c_1 \cap i_2 \\ c_2 \text{ restricted by } I \cap i_2 &= c_2 \cap i_2 \end{aligned}$$

It can be seen that four intersections are analogous to the case with two complete splits, i.e., the compatibility check is valid even if the taxon sets are different.

In TreeShredder, the two sides of an incomplete split are represented as two `dynamic_bitset`s where each bit corresponds to a taxon. If the taxon occurs on that side the bit is set to 1. If an incomplete split I can occur on the same tree as a complete split C they are said to be congruent, otherwise they are incongruent. A congruent incomplete split is additionally supportive iff both of its sides are subsets of different sides of split C. An incomplete split is incongruent iff none of its sides are subsets of either side of split C.

More formally, an incomplete split I with the sides $i_1|i_2$ and a complete split C with the sides $c_1|c_2$ are incongruent iff

$$i_1 \nsubseteq c_1 \;\wedge\; i_2 \nsubseteq c_2 \;\wedge\; i_1 \nsubseteq c_2 \;\wedge\; i_2 \nsubseteq c_1,$$

congruent *and* supportive iff

$$(i_1 \subseteq c_1 \;\wedge\; i_2 \subseteq c_2) \;\vee\; (i_1 \subseteq c_2 \;\wedge\; i_2 \subseteq c_1),$$

and congruent but *not* supportive in all other cases.

7.2 Methods: Finding congruent trees and maximum/sum of occurrence

To find trees that are congruent with a set of incomplete splits M^i one needs to check whether for each of those incomplete splits a given tree contains at least one split that is congruent *and* supportive.

It is straightforward to find the maximum occurrence among splits in the trees that are supportive of (or incongruent with) each incomplete split in M^i. For this, simply checking compatibility against each split in `DataDepot`'s vector of unique nontrivial splits (see subsection 3.1.3) will do. This can be combined with calculating the *sum of occurrence* of supportive (or of incongruent) splits that are incongruent with the splits whose occurrence has already been summed up. To do this, one needs to sort all supportive (or incongruent) splits in descending order. To determine quickly whether a split is incongruent with another, one can use the property that the sum of occurrence of two incongruent splits is not greater than the number of trees. Otherwise they would need to occur together in at least one tree and, thus, could not be incongruent. Only if the sum of occurrence is $\leq 100\%$, one needs to explicitly check congruency using the intersection method.

I implemented two algorithms of different greediness:

Version #1, greedier, no block handling: The first split has maximum occurrence among all supportive (or incongruent) splits of the trees and is used to seed the set of supportive (or incongruent) splits from which to calculate the sum. A supportive (or incongruent) split is added in descending order to the set if it is incongruent with every split in the set.

Version #2, less greedy, with block handling (`-blockHandling` flag): By block handling, splits of equal support are considered together. This guarantees that splits of equal occurrence are treated equally, enabling the algorithm to choose the local combination of splits that has (local) maximum sum of occurrence. If there are multiple local combinations of equal sum the first such combination is then added to the set. Unfortunately, this arbitrariness in choosing can prevent the consideration of combinations in blocks further downstream that would lead to a higher global sum of occurrence.

In both versions, the sum of occurrence depends on the order in which the splits are added to the set and can vary with a different order. Version #2 suffers from the same constraints as version #1 but to a smaller extent because locally maximal combinations are found, at the cost of increased complexity and runtime. The globally maximal sum of occurrence may not be found. A different parsing order of trees when splits are extracted can lead to a different order of splits in the sorted supportive (or incongruent) splits and, thus, to a different summation result. Conversely, the maximum support is independent of order.

7.2.1 Determining congruency status of splits

Given a set of incomplete splits (`-cong`), TreeShredder can determine the congruency status of all splits of an input file with those incomplete splits. The output file contains three blocks of data for each incomplete split: the first block contains all congruent and supportive splits of the incomplete split, the second block contains all congruent but not supportive splits, and the third block contains all incongruent splits. To obtain all congruent splits, one has to combine the congruent and supportive splits and the congruent but not supportive splits. To obtain all splits that are not supportive, one has to combine all congruent but not supportive splits and all incongruent splits.

7.2.2 Output files

The `-cong` flag produces three output files. One contains the congruency status of splits with the incomplete splits. Another contains the trees that are congruent with the set of incomplete splits. And a third contains the incomplete splits and their respective maximum or sum of occurrence among supportive or incongruent splits. The first column contains the incomplete splits, the second contains the maximum occurrence among supportive splits, the third the sum of occurrence among supportive splits, the fourth the maximum occurrence among incongruent splits, and the fifth the sum of occurrence among incongruent splits. The columns are separated by tabs.

7.3 Results: Finding congruent trees and maximum/sum of occurrence

TreeShredder can find all trees that are congruent with a set of incomplete splits. There are 945 different trees with 7 taxa. Of all 945 possible trees, only the 6 trees in Tab. 7.3 are congruent with the example set of incomplete splits in Tab. 7.2. Tab. 7.2 contains TreeShredder's congruency measures of the splits in the trees. For each incomplete split, the best supported split that is congruent and supportive with it occurs in about 11% (i.e., in 105 or exactly a ninth) of the trees. Because the values in the second and third column are identical, it can be concluded that there are no other splits in the trees that are also congruent and supportive with the incomplete splits *and* incongruent with the best such split: The set of congruent and supportive splits that are additionally incongruent with each other only contains one split, which is the one with maximum occurrence. The same is not true of the splits in the trees that are incongruent with the incomplete splits. Here, the sum of occurrence of the set of incongruent splits that are incongruent with each other is bigger than the best supported incongruent split (about 22% vs 11%).

7.4 Conclusion

TreeShredder can find congruent trees, congruency measures for incomplete splits, and determine the congruency status of the splits of an input file with the incomplete splits. Given a set of incomplete splits and a set of trees, TreeShredder can find the trees that are congruent with all of them. This can be useful, e.g., when one is interested only in a subset of the taxa of the trees.

Similarly, one can find all splits in an input file that are congruent, congruent and supportive, congruent but not supportive, incongruent, and not supportive with a set of incomplete splits.

The congruency measures for a set of incomplete splits are the maximum occurrence among congruent and supportive splits, the sum of occurrences among congruent and supportive splits, the maximum occurrence among incongruent splits, and the sum of occurrences among incongruent trees. To calculate the respective sums of occurrences, the occurrences of incongruent trees are summed up.

Table 7.2: Set of four example incomplete splits and their congruency measures: The trees compatible with these incomplete splits (first column) are shown in Tab. 7.3. The other four columns contain the congruency measures of the complete splits of the trees: the maximum occurrence among congruent and supportive splits, the sum of occurrence among congruent and supportive splits, the maximum occurrence among incongruent splits, and the sum of occurrence among incongruent splits.

incompleteSplits	maxSupportive	sumSupportive	maxIncongruent	sumIncongruent
B D | E G ? A C F	0.1111	0.1111	0.1111	0.2222
A E | F G ? B C D	0.1111	0.1111	0.1111	0.2222
A B | C E ? D F G	0.1111	0.1111	0.1111	0.2222
A C | D G ? B E F	0.1111	0.1111	0.1111	0.2222

Table 7.3: Trees that are congruent with the set of four example incomplete splits: These are the 6 trees (out of 945 trees) that are congruent with the four incomplete splits in Tab. 7.2.

```
(A,B,(C,(D,(E,(F,G)))));
(A,(((B,D),F),G),(C,E));
(A,((B,D),(F,G)),(C,E));
(A,(((B,D),G),F),(C,E));
(A,((B,(D,F)),G),(C,E));
(A,(((B,F),D),G),(C,E));
```

8 Consensus Trees

8.1 Introduction

Having large sets of trees, there is often the demand of summarizing them in one tree. Such trees are typically called consensus trees. There are consensus tree construction methods for rooted and unrooted trees. An example of a method for rooted trees was introduced by Adams III (1972), which creates a consensus tree from all three-taxon statements (i.e., incomplete splits where two taxa are more closely related than a third; see section 7.1), that are not contradicted by any other in the input trees. For a method for unrooted trees, rooted trees are considered to be identical if they differ only in their roots (Felsenstein, 2004). The construction method I implemented in TreeShredder – finding the splits from which the consensus tree will be created – does not differentiate between rooted and unrooted trees, as it solely considers nontrivial, non-root splits of the input trees. Information about roots is ignored.

Depicting the topologies of multiple trees in a single consensus tree requires a set of compatible splits of those trees, from which such a consensus tree can be constructed. A set of splits is said to be compatible if every split is compatible with every other split in the set. For t binary trees with n taxa each, there are $t(n-3)$ or $\mathcal{O}(tn)$ nontrivial splits. However, the number of *unique* nontrivial splits increases typically only by $\mathcal{O}(n \ln (tn))$ (see Fig. 3.2a). Thus, finding a set of compatible splits for t trees would imply $\mathcal{O}((n \ln (tn))^2)$ pairwise comparisons. Here, one can take advantage of the fact that the sum of occurrences of two incompatible splits cannot be greater than the total number of trees because this would imply that they occurred together on at least one tree. Thus, if the sum is greater, they must be compatible and no compatibility check need be done.

There are several kinds of consensus trees, which differ in the rules for finding the set of compatible splits from which they are created. They will be introduced in the following. (Note that the UK English term ”relative majority” is equivalent to the US English term ”plurality”, meaning a split occurring in more trees than any other option, but no more than half of all trees.)

8.1.1 Majority Rule Consensus Tree (M_l)

The most common consensus tree is the majority rule consensus tree (M_l) which comprises the set of splits that have an occurrence ratio equal to or higher than a custom specified integer threshold l in the range of $[50, 100]$ percent (Margush and McMorris, 1981). Splits with an occurrence ratio of 50% (i.e., those that occur in exactly half of trees) need to be discarded to prevent draws where two incompatible splits have equal occurrence ratios. No compatibility checks need be done because the sum of occurrence of any two splits in the set is guaranteed to be greater than the number of trees and, therefore, the splits are guaranteed to be compatible. The M_l only contains splits that occur in more than half of trees.

8.1.2 Majority Rule Extended Consensus Tree (M_{ext})

Finding the set of compatible splits for the majority rule extended consensus tree (M_{ext}) requires sorting the splits by occurrence in descending order. Then, starting with the split with maximal occurrence, splits are added to the set if they are compatible with every split already in the set (Felsenstein, 1993). Again, splits with occurrence ratios $> 50\%$ are added without the requirement of checking compatibility. However, splits at or below 50% can result in draws, where splits of equal occurrence are incompatible. Thus, the set of compatible splits (and, as a result, the M_{ext}) can vary depending on the order of the splits in the sorted list. Two splits with *equal* occurrence need not have the same compatibility status with the splits in the compatible set, or adding one before the other could lead to the exclusion of different splits further down the list. This means there may exist multiple M_{ext} consensus trees for the same set of trees, depending on the order of splits in the sorted list (the order of splits with equal occurrence is not defined). Furthermore, the M_{ext} may contain splits that are incompatible with splits that have higher occurrences but were outvoted already by other splits and are therefore not included in the M_{ext} consensus tree.

8.1.3 Relative Majority Consensus Tree (M_{rel})

Finding the set of compatible splits for the relative majority consensus tree (M_{rel}, Schmidt, 2003) is similar to finding the set of compatible splits for the M_{ext}. Here, however, encountering the first incompatible split terminates the addition of splits. Furthermore, splits with equal occurrence as that first incompatible split are removed from the set of compatible splits (Schmidt, 2003). This guarantees that splits with equal occurrence are either all in the set of compatible splits (and the consensus tree), or none are. The M_{rel} contains only splits for which there exist no incompatible split in the trees that has higher or equal occurrence.

8.1.4 Global Relative Majority Consensus Tree (M_{glob})

Finding the set of compatible splits for the global relative majority consensus tree (M_{glob}) is similar to finding the set of compatible splits for the M_{rel}, but splits are only added if they are compatible with *all* splits (of all trees) that have an equal or higher occurrence, not only those in the set of compatible splits. The M_{glob} contains only splits for which there exist no incompatible split in the trees that has higher or equal occurrence. It is a potentially better resolved extension of the M_{rel}. In TreeShredder, the compatibility checks are parallelized.

8.1.5 Strict Consensus Tree (M_{strict})

The strict consensus tree (M_{strict}) contains all splits that occur in all trees and thus are added to the set of compatible splits (Bryant, 1997). As they occur in all trees, they are naturally guaranteed to be compatible.

8.1.6 Semi Strict Consensus Tree (M_{semi})

The semi strict consensus tree (M_{semi}) contains only splits that are compatible with all splits of all trees (Bremer, 1990). For this, only splits with no incompatible split in any tree are added to the compatible set. This is the most computationally expensive kind of consensus tree because it requires comparing each split with every other split. In TreeShredder, these comparisons are parallelized.

8.2 Methods

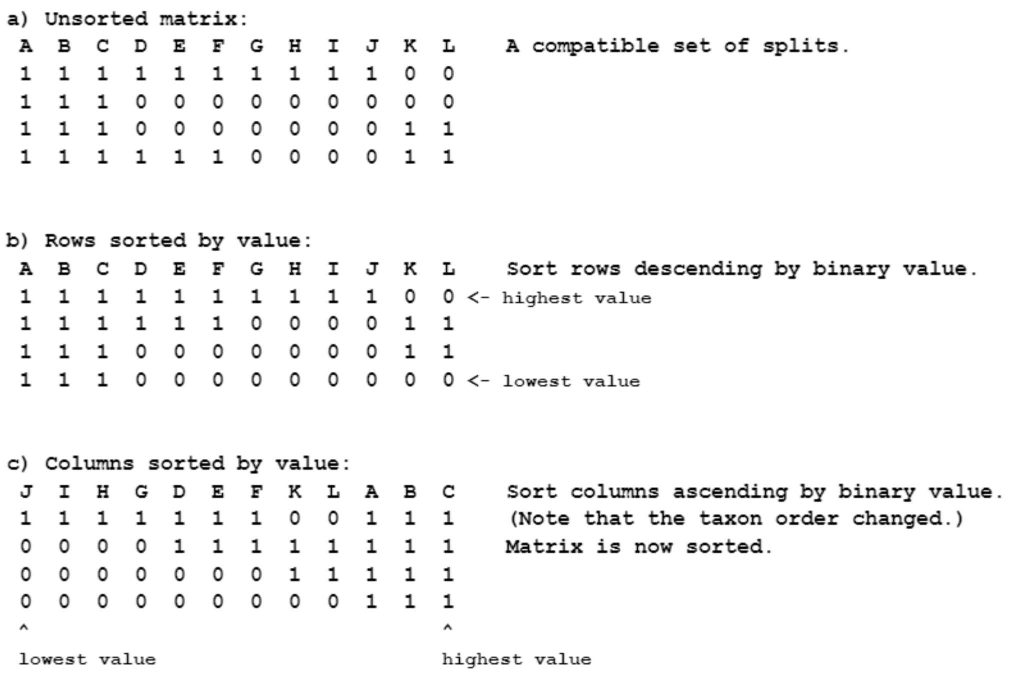
8.2.1 Tree Creation Algorithm

From a set of compatible splits, a tree can be created. The algorithm was described and implemented by Marc Zobel and Vincent Reiner for the software development course "PR Praktikum aus Bioinformatik" ("PR Laboratory Bioinformatics") at University Vienna in 2015. This approach might have also been used by others earlier in the past in several phylogenetic software.

Let the set of compatible splits form a matrix, where the rows correspond to splits and the columns to taxa, as shown in Fig. 8.1. The matrix is sorted, first by descending value of the splits when interpreted as binary value (see b)), then by ascending value of the columns when interpreted as binary value (see c)).

From the sorted matrix, the Newick string that represents the tree that contains the set of compatible splits is built (see Fig. 8.2): Each taxon's name (now in the new order) is added to the newly created Newick tree string as well as its trivial split's edge information. Then each split is processed at the corresponding column: If there is a 0-1 transition from this column to the column of the next taxon, a closing parenthesis ')' and that split's edge information is added to the Newick string (see b), d), and f)). If there is a 1-0 transition, the opening parenthesis '(' count variable is increased by 1 (see d)). If there are no 0-1 or 1-0 transitions, only the taxon names are added (see a), c), e), and g)). After each split was processed this way, a comma ',' and as many opening parentheses as counted by the count variable are added to the Newick string.

By keeping track of the opening and closing parentheses, one can add the missing ones to the Newick string (see h)). Finally, the outermost enclosing parentheses and the semicolon ';' are added (see i)).



```
a) Unsorted matrix:
 A  B  C  D  E  F  G  H  I  J  K  L    A compatible set of splits.
 1  1  1  1  1  1  1  1  1  1  0  0
 1  1  1  0  0  0  0  0  0  0  0  0
 1  1  1  0  0  0  0  0  0  0  1  1
 1  1  1  1  1  1  0  0  0  0  1  1


b) Rows sorted by value:
 A  B  C  D  E  F  G  H  I  J  K  L    Sort rows descending by binary value.
 1  1  1  1  1  1  1  1  1  1  0  0 <- highest value
 1  1  1  1  1  1  0  0  0  0  1  1
 1  1  1  0  0  0  0  0  0  0  1  1
 1  1  1  0  0  0  0  0  0  0  0  0 <- lowest value


c) Columns sorted by value:
 J  I  H  G  D  E  F  K  L  A  B  C    Sort columns ascending by binary value.
 1  1  1  1  1  1  1  0  0  1  1  1    (Note that the taxon order changed.)
 0  0  0  0  1  1  1  1  1  1  1  1    Matrix is now sorted.
 0  0  0  0  0  0  0  1  1  1  1  1
 0  0  0  0  0  0  0  0  0  1  1  1
 ^                                ^
 lowest value                     highest value
```

Figure 8.1: Sorting the matrix for Newick string creation: Rows correspond to splits, columns to taxa. The leftmost bit and the bottommost bit are the most significant when the rows and columns are interpreted as a binary value, respectively. Note that the final taxon order may have changed.

```
a) Sorted matrix:
 J   I   H   G   D   E   F   K   L   A   B   C     No 0-1 or 1-0 transitions from
 1   1   1   1   1   1   1   0   0   1   1   1     J, I and H columns to the next
 0   0   0   0   1   1   1   1   1   1   1   1     ones. Add taxon names and
 0   0   0   0   0   0   0   1   1   1   1   1     commas.
 0   0   0   0   0   0   0   0   0   1   1   1

                      Newick string:   J,I,H,
```

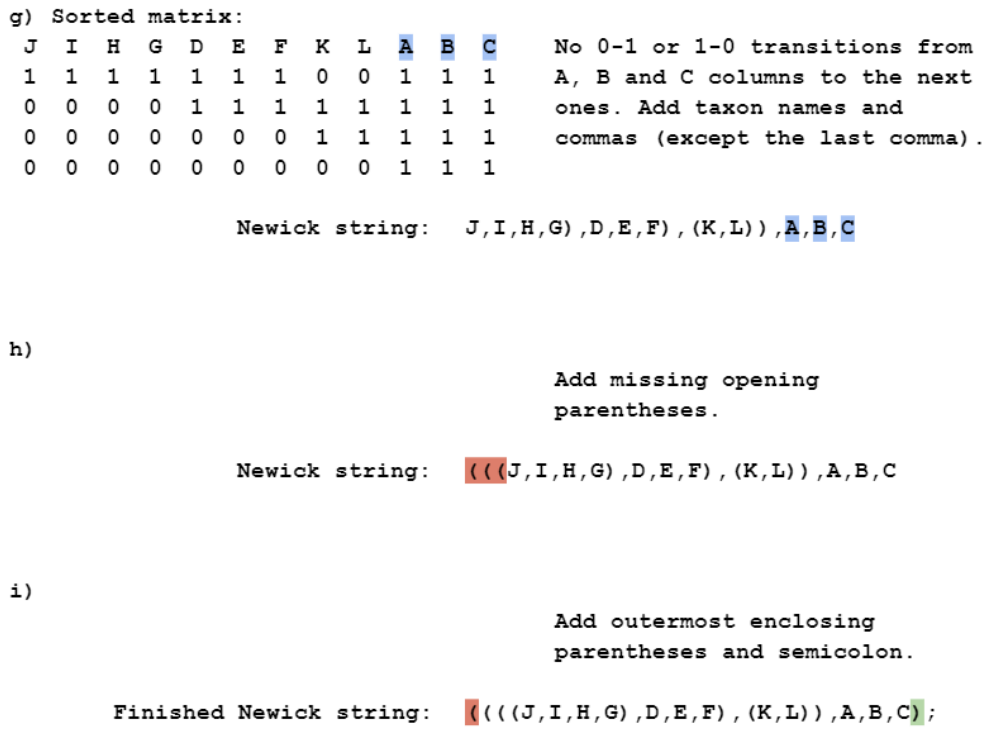
```
b) Sorted matrix:
 J   I   H   G   D   E   F   K   L   A   B   C     0-1 transition from G column to
 1   1   1   1   1   1   1   0   0   1   1   1     the next. Add taxon name,
 0   0   0   0   1   1   1   1   1   1   1   1     closing parenthesis and comma.
 0   0   0   0   0   0   0   1   1   1   1   1
 0   0   0   0   0   0   0   0   0   1   1   1

                      Newick string:   J,I,H,G),
```

```
c) Sorted matrix:
 J   I   H   G   D   E   F   K   L   A   B   C     No 0-1 or 1-0 transitions from
 1   1   1   1   1   1   1   0   0   1   1   1     D and E columns to the next
 0   0   0   0   1   1   1   1   1   1   1   1     ones. Add taxon names and
 0   0   0   0   0   0   0   1   1   1   1   1     commas.
 0   0   0   0   0   0   0   0   0   1   1   1

                      Newick string:   J,I,H,G),D,E,
```

```
d) Sorted matrix:
 J   I   H   G   D   E   F   K   L   A   B   C     1-0 and 0-1 transition from F
 1   1   1   1   1   1   1   0   0   1   1   1     column to the next. Add taxon
 0   0   0   0   1   1   1   1   1   1   1   1     name, closing parenthesis,
 0   0   0   0   0   0   0   1   1   1   1   1     comma and opening parenthesis.
 0   0   0   0   0   0   0   0   0   1   1   1

                      Newick string:   J,I,H,G),D,E,F),(
```

```
e) Sorted matrix:
 J   I   H   G   D   E   F   K   L   A   B   C     No 0-1 or 1-0 transitions from
 1   1   1   1   1   1   1   0   0   1   1   1     K column to the next ones. Add
 0   0   0   0   1   1   1   1   1   1   1   1     taxon name and comma.
 0   0   0   0   0   0   0   1   1   1   1   1
 0   0   0   0   0   0   0   0   0   1   1   1

                      Newick string:   J,I,H,G),D,E,F),(K,
```

```
f) Sorted matrix:
 J   I   H   G   D   E   F   K   L   A   B   C     0-1 transitions from L column
 1   1   1   1   1   1   1   0   0   1   1   1     to the next. Add taxon name,
 0   0   0   0   1   1   1   1   1   1   1   1     closing parentheses and comma.
 0   0   0   0   0   0   0   1   1   1   1   1
 0   0   0   0   0   0   0   0   0   1   1   1

                      Newick string:   J,I,H,G),D,E,F),(K,L)),
```



```
g) Sorted matrix:
 J  I  H  G  D  E  F  K  L  A  B  C    No 0-1 or 1-0 transitions from
 1  1  1  1  1  1  1  0  0  1  1  1    A, B and C columns to the next
 0  0  0  0  1  1  1  1  1  1  1  1    ones. Add taxon names and
 0  0  0  0  0  0  0  1  1  1  1  1    commas (except the last comma).
 0  0  0  0  0  0  0  0  0  1  1  1

                Newick string:   J,I,H,G),D,E,F),(K,L)),A,B,C


h)
                                       Add missing opening
                                       parentheses.

                Newick string:   (((J,I,H,G),D,E,F),(K,L)),A,B,C


i)
                                       Add outermost enclosing
                                       parentheses and semicolon.

       Finished Newick string:   ((((J,I,H,G),D,E,F),(K,L)),A,B,C);
```

Figure 8.2: Example of creating a Newick string: This figure begins on the previous page. 0-1 transitions induce a closing parenthesis (highlighted in green). 1-0 transitions induce an opening parenthesis (highlighted in red). Taxon names (highlighted in blue) are always added to the string. Colons are added between closing and opening parentheses (or between taxon names if there are none). The Newick string is finished after the missing opening parentheses, the outermost enclosing parentheses, and the semicolon are added. The splits' edge information can be added to the Newick string after a taxon name or a closing parenthesis (not shown in this example).

8.3 Results and Discussion

Figs. 8.3a and 8.3b show the runtime of generating a M_{50} and a M_{semi} consensus tree, respectively. For the latter, 8 threads were used. The runtimes remain roughly constant because the size of the set of compatible splits does not change much with the number of trees, and thus the creation of the tree from a set of splits takes roughly equal time.

Fig. 8.4a shows the combined runtime of sorting the splits by their support (if necessary; cf. column 2 of Tab. 8.1 on page 82) and finding the compatible set of splits (see Fig. 8.4b) for the six different consensus trees for the RAN-400 dataset. One can see that almost the entire runtime of up to 5 hours for M_{rel}, M_{glob}, and M_{ext} is due to sorting the more than 250 million unique nontrivial splits of the 1 million RAN-400 trees (see Fig. 3.2b) by their support.

8.4 Conclusion

Subsets of sets of compatible splits

Since consensus trees are formed from a subset of all splits of a given set of trees, it might be interesting to know if their split sets form subsets of each other. These relationships can be summarized by the following statements (where $\subseteq$ means the left consensus tree is less or equally resolved as the right one, but all splits from the left tree are also in the right tree):

$$M_{strict} = M_{100} \subseteq M_l \subseteq M_{50} \subseteq M_{rel} \subseteq M_{glob} \subseteq M_{ext}$$

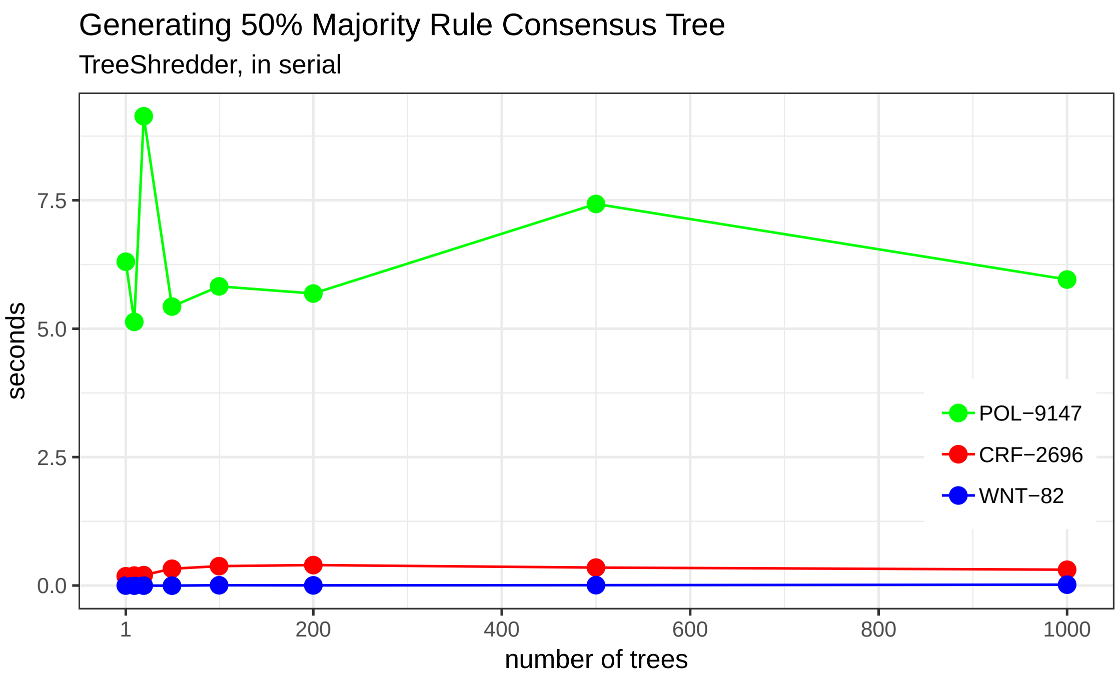
$$M_{strict} \subseteq M_{semi} \subseteq M_{glob}$$

The M_{strict} is the simplest consensus tree and its split set is necessarily a subset of all other consensus trees. It is also clear that the set of compatible splits of a M_l consensus tree is a subset of a M_l consensus tree with an equal or lower threshold.

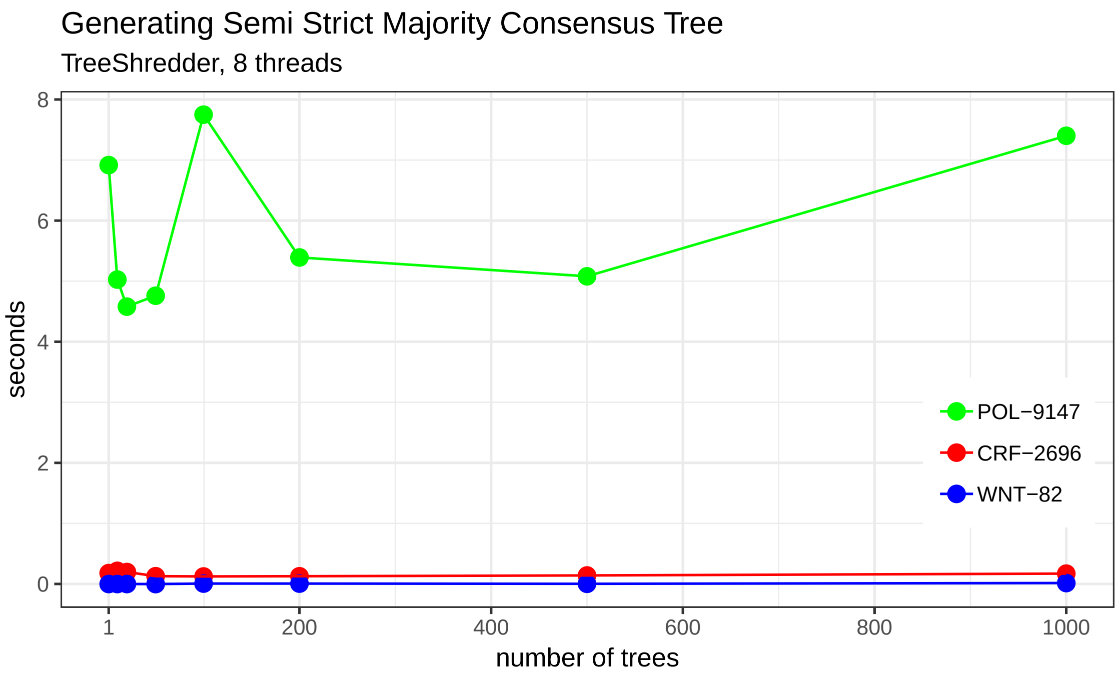
The split set of the M_{rel} is a subset of, both, the split sets of the M_{ext} and M_{glob} consensus trees because their methods continue adding splits when the M_{rel} method has already stopped because an incompatible split was encountered.

The split set of the M_{glob} is a subset of the split set of the M_{ext} because the latter's split set is a subset of the splits against which splits for the M_{glob} would be checked (i.e., all equal or better supported splits) at any stage of finding the compatible set of splits. Therefore a split that was rejected in the M_{ext} method is also rejected in the M_{glob} method, but not necessarily vice versa. By the same logic, the M_{semi} is a subset of the M_{glob}: splits that are added in the stricter M_{semi} are also added in the M_{glob}, but not necessarily vice versa.

If all input trees are fully resolved, then the M_{semi} and M_{strict} are identical. Only splits within unresolved regions (i.e., multifurcations) in one of the input trees could potentially be included in the M_{semi}, but would be missing from the M_{strict} consensus tree (Felsenstein, 2004).

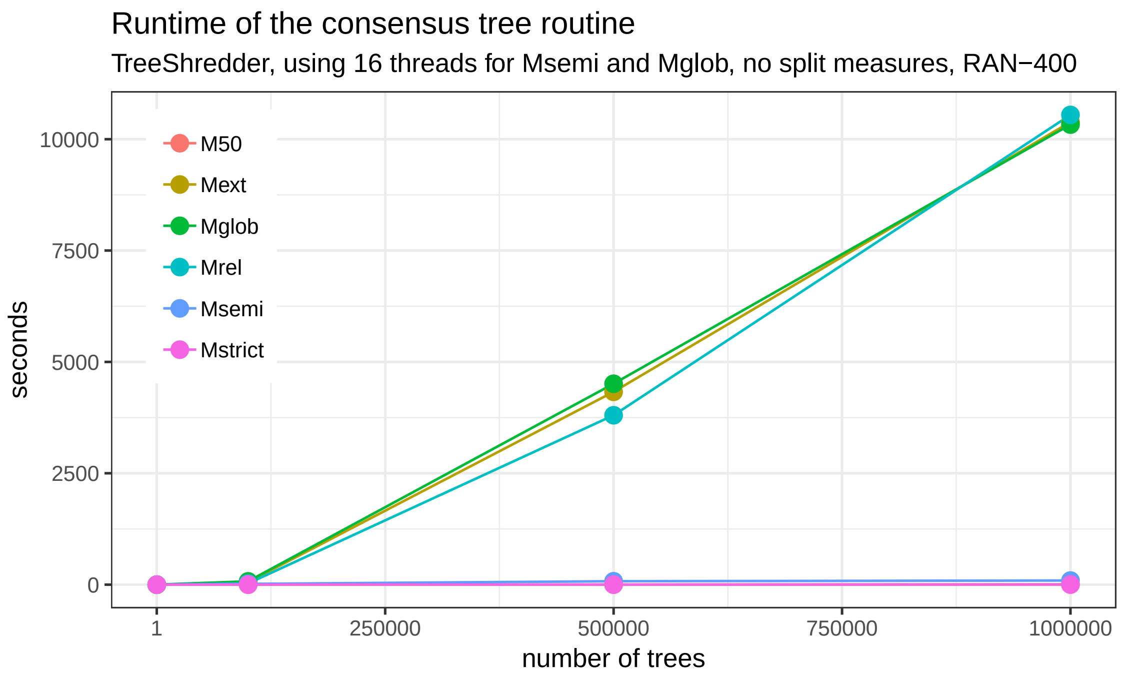


(a) Runtime of the 50% majority rule consensus tree: Runtime is roughly constant.

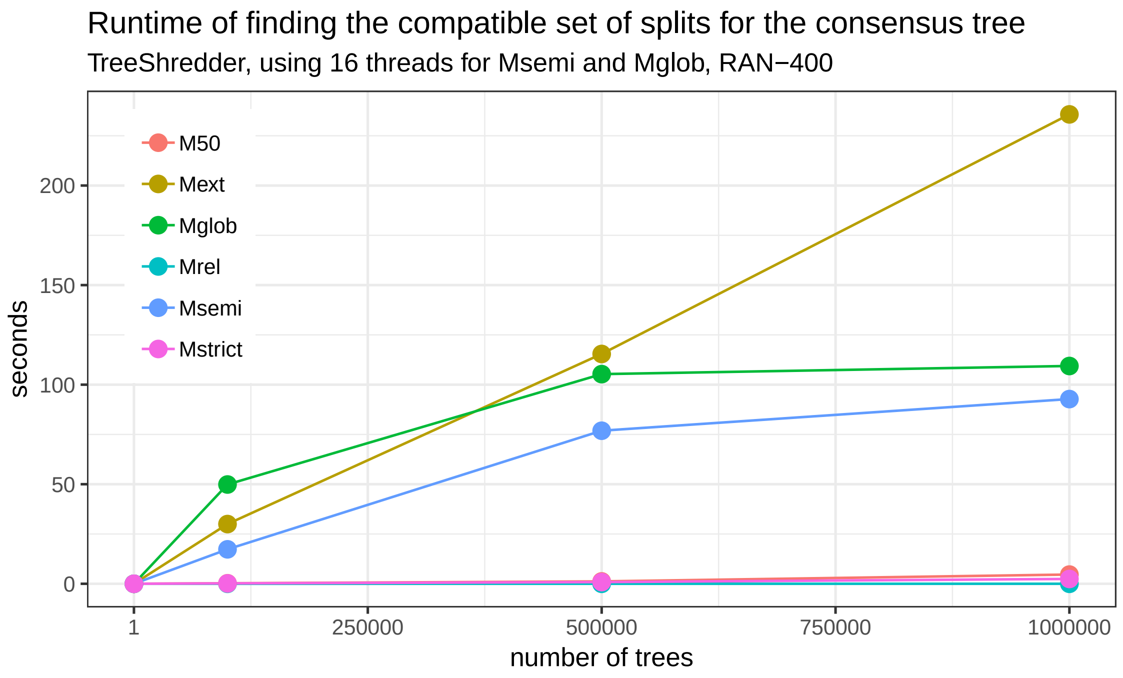


(b) Runtime of the semi strict majority consensus tree: Runtime is roughly constant, 8 threads were used.

Figure 8.3: TreeShredder runtimes of the 50% majority rule consensus tree and the semi strict majority consensus tree: The latter is parallelized using 8 threads. Both take roughly equal time. (Runtimes only include the time to find the set of compatible splits and to create the consensus tree, but exclude the time to read the tree information from a file.)



(a) Consensus routine runtime of the six different consensus trees for the RAN-400 dataset: Runtime is much longer for M_{ext}, M_{glob}, and M_{rel} than the others.



(b) Runtime of finding the compatible set of splits for the six different consensus trees for the RAN-400 dataset: Runtime is longest for M_{ext}, M_{glob} and M_{semi}. This excludes the time to sort the splits by support.

Figure 8.4: TreeShredder runtimes of the six consensus trees for the RAN-400 dataset: M_{semi} and M_{glob} are parallelized using 16 threads. No split measures were calculated. (Runtimes do not include the time to read the tree information from a file.)

Properties of consensus trees

The different consensus tree methods apply different criteria to find the compatible set of splits, which causes the consensus trees to differ in their properties. These different criteria and properties are summarized in Tab. 8.1.

While the M_l consensus tree only contains splits that occur in more than half of all trees, this is not necessarily true of M_{ext}, M_{rel}, M_{glob}, and M_{semi}. Because any split that occurs in $> 50\%$ of the trees is necessarily compatible with any other such split, there need not be any compatibility checks done. These splits can simply be added to the compatible set of splits, and the order in which they are added is irrelevant. Since the M_{semi} only contains splits that are compatible with all splits in all trees, the order in which the compatibility is checked is also irrelevant. If the trees are fully resolved, then M_{semi} and M_{strict} are identical and compatibility checks are not necessary. (TreeShredder does not check whether trees are fully resolved, and therefore checks compatibility for the M_{semi}.)

The M_{ext} method is the only one that could produce different consensus trees depending on the (undefined) order of splits within blocks of equal support. The M_{ext}, M_{glob}, and M_{semi} may contain some – not necessarily all – splits of equal support. This is not the case for the M_l (including the M_{50} and M_{strict}) for obvious reasons, and for the M_{rel} because the method stops adding splits once an incompatible is found and all splits with equal support are removed from the compatible set of splits.

Obviously, M_{50}, M_{rel}, M_{glob}, and M_{ext} are guaranteed to contain all splits that occur in $> 50\%$ of trees. There is no such guarantee for the M_l (apart from the M_{50}), M_{semi}, and M_{strict}, although it may turn out to be true for particular datasets.

In turn, only the M_l (including the M_{50} and M_{strict}) is guaranteed to contain only splits with $> 50\%$ support. There is no such guarantee for the M_{semi}, M_{rel}, M_{glob}, and M_{ext} because for those the search for compatible splits continues below 50% support.

Finally, the M_{ext} is the only consensus tree that is not guaranteed to contain only splits that have no better supported incompatible split in any tree. This is because the M_{ext} method searches through all splits, but only checks for their compatibility

Table 8.1: Summary of relationships between consensus trees: The "+" means that the property is true of that consensus method. The "–" means that the property is false of that consensus method. The "–*" means that the property is generally false of trees of that consensus method but there may be exceptions for particular datasets.

consensus method	requires sorting of splits	requires compatibility checks of splits	tree is independent of order of equally supported splits	all splits of equal support are either in or out	contains all splits with > 50% support	only contains splits with > 50% support	splits have no better supported incompatible splits
M_{strict}	–	–	+	+	–*	+	+
M_{semi}	–	+	+	–*	–*	–*	+
M_l	–	–	+	+	–*	+	+
M_{50}	–	–	+	+	+	+	+
M_{rel}	+	+	+	+	+	–*	+
M_{glob}	+	+	+	–*	+	–*	+
M_{ext}	+	+	–	–*	+	–*	–*

with the splits in the compatible set of splits, not against better supported splits that have been discarded already. This is similar to the M_{rel} method, but here the search for compatible splits is terminated once an incompatible one is found.

Creating a Newick string

The Tree Creation Algorithm creates a Newick tree string from a set of compatible splits by first, sorting a matrix of splits, and second, progressively adding taxon names, parentheses, commas, and edge information to the string.

Runtime of the different consensus trees

The runtimes of generating the different consensus trees are roughly equal and remain constant with increasing number of trees for the biological datasets. This is also true of the M_{semi}, whose set of compatible splits is the most computationally expensive to find. However, generating the M_{rel}, M_{glob}, and M_{ext} for the random dataset takes much longer than the other consensus tree methods due to the necessity of sorting many millions of nontrivial splits by their support.

9 Matrix Representation

9.1 Introduction

There are several ways to compute species trees from alignments of a set of genes. One possibility is to concatenate the gene alignments and analyse this so-called superalignment. Another way is using the gene trees to construct a so-called supertree. If the taxon sets of the trees are identical, one can use consensus methods as described in chapter 8.

However, if the taxon sets of trees are not identical, TreeShredder's consensus tree feature cannot depict the topology and information of those trees in a single tree. Instead, a supertree needs to be constructed that combines diverse phylogenies into a single tree. Input for several of these methods is a Matrix Representation (Ragan, 1992; Baum, 1992), where rows correspond to unique taxa of the trees and columns correspond to the presence or absence status of that unique taxon in the trees' nontrivial splits and on which side of the splits they occur. From this matrix, a supertree can be constructed using diverse methods such as Matrix Representation with Parsimony (MRP; Ragan, 1992; Baum, 1992), Matrix Representation with Flipping (MRF; Chen, Diao, et al., 2003), Matrix Representation with Distances (MRD; Lapointe, Wilkinson, and Bryant, 2003; Levasseur and Lapointe, 2006), Matrix Representation with Compatibility (MRC; Ross and Rodrigo, 2004), and Matrix Representation with Bad Clade Deletion (BCD; Fleischauer and Böcker, 2017). These methods require a Matrix Representation as input but differ, e.g., in complexity, number of output supertrees, and introduction of clades that are contradicted by all input trees (Kupczok, Schmidt, and Haeseler, 2010; Fleischauer and Böcker, 2017).

However, with the advent of the partition model in combination with the superalignment, supertree construction has lost some popularity in the scientific community in recent years. Over the same period, even though theoretical research is still being done in this field, maintenance of and support for programs that allow for generating a Matrix Representation for supertree construction have been discontinued. To fill that gap, TreeShredder offers this feature with the `-mr` flag option.

9.2 Methods

The creation of a Matrix Representation file takes files containing tree information (that may have different taxon sets) as input. In TreeShredder, these can be Newick, Nexus, or TreeShredder files. Let I denote the set of input files, and U denote the union set of (unique) taxa among the input files of I.

First, the union set U of all taxon sets among input files is created to find all unique taxa. Then, in that order (as nested loops), for each unique taxon u in U, for each input file in set I, for each tree t in that file, for each nontrivial split b in that tree, the value of that unique taxon u (at its position) in split b (i.e., 0 or 1) is written to the Matrix Representation file. If the unique taxon does not exist in the taxon set of t then a question mark '`?`' is written instead, indicating the lack of information regarding that taxon. The runtime complexity of this algorithm is $\mathcal{O}(|U| * s)$, where s is the sum of nontrivial splits of all trees of all input files.

TreeShredder creates a Matrix Representation file in Nexus format by default, where the matrix strings corresponding to trees and files are separated by one and three blank spaces, respectively, for better readability. Additionally, the start and end positions of, both, the strings in the matrix corresponding to individual trees, and corresponding to individual files, are stored in a Nexus "Set" block.

Some programs require files in FASTA format for supertree construction. For them, TreeShredder offers the `-FASTA` flag option, which outputs an additional Matrix Representation file in FASTA format. Here, the matrix strings corresponding to trees and files are separated by one or two newlines, respectively.

9.2.1 Presence-Absence file

For a simple overview about which taxon exists in which tree information file, there is the `-PresAbs` flag option. This will output a Presence-Absence matrix file where the first column contains the unique taxon names. The other columns represent each tree in each file and contain a 1 if the unique taxon exists in that tree or a 0 if not. These columns are denoted, e.g., as "f1t1" to denote the indices of the trees and files. The columns are separated by tabs.

9.3 Results and Discussion

Fig. 9.1 shows TreeShredder runtime of reading Newick file and running the Matrix Representation feature for six datasets collected by Rob Lanfear (Lanfear, 2019). The datasets are collected from stinging wasps (Aculeata; Branstetter et al., 2017), ray-finned fishes (Actinopterygii; Faircloth et al., 2013), spurge (Euphorbia; Horn et al., 2014), spiny-rayed fishes (Acanthomorpha; Near et al., 2013), amphibians (Amphibia; Pyron and Wiens, 2011), and mushrooms (Basidiomycota; Varga et al., 2019) with different number of partitions (i.e., trees) and unique taxa (see Tab. 9.1). The runtime tends to increase with the product of the number of unique taxa and the sum of nontrivial splits. This is because the Matrix Representation output file grows linearly in size with this product. Overall, runtime is in the range of a few seconds.

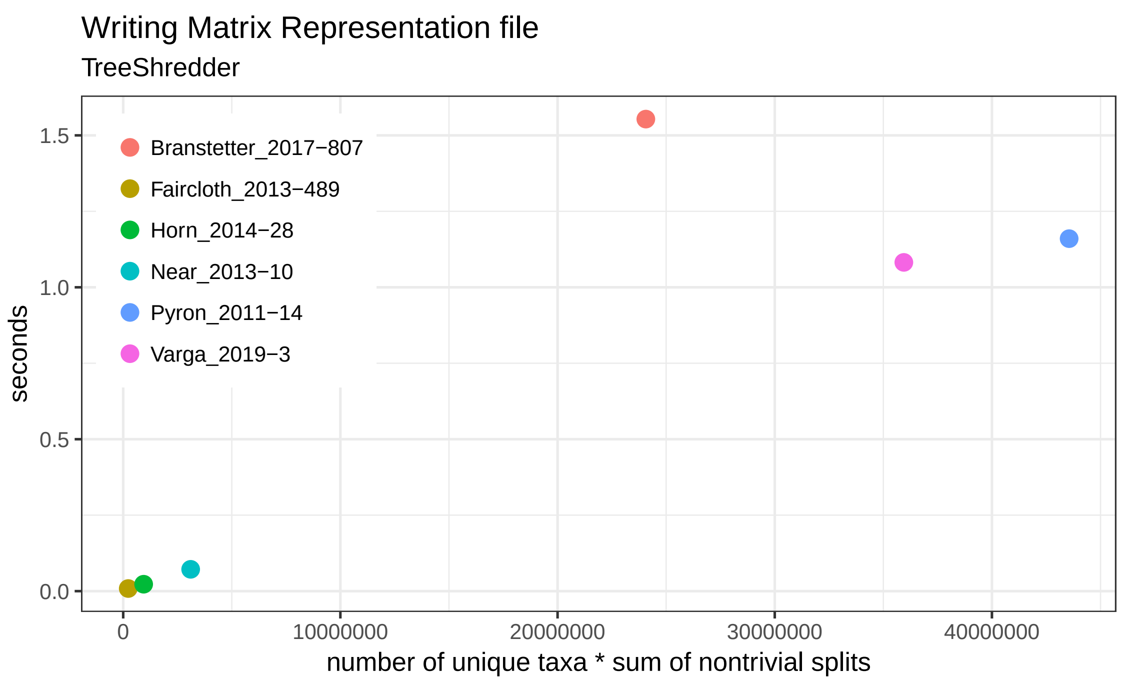


Figure 9.1: TreeShredder runtimes of Matrix Representation for six datasets: They are encoded by first author, followed by year of publication and finally by the number of DNA dataset partitions (mostly genes) extracted from the original superalignment, i.e., this is also the number of trees for that dataset.

Table 9.1: Datasets for Matrix Representation: The datasets are named as they appear in Fig. 9.1. The number of trees corresponds to the number of partitions. The "taxa" column contains the number of unique taxa among the trees of the dataset. The datasets were collected by Rob Lanfear (Lanfear, 2019).

dataset	**trees**	**taxa**	**clade**	**dataset DOI**
Branstetter_2017-807	807	187	Aculeata (stinging wasps)	10.5061/dryad.r8d4q
Faircloth_2013-489	489	27	Actinopterygii (ray-finned fishes)	10.5061/dryad.j015n
Horn_2014-28	28	197	Euphorbia (spurge)	10.5061/dryad.sb1j1
Near_2013-10	10	608	Acanthomorpha (spiny-rayed fishes)	10.5061/dryad.d3mb4
Pyron_2011-14	14	2872	Amphibia (amphibians)	10.5061/dryad.vd0m7
Varga_2019-3	3	5285	Basidiomycota (mushrooms)	10.5061/dryad.gc2k9r9

9.4 Conclusion

TreeShredder reintroduces the Matrix Representation feature for the construction of supertrees from input trees with different taxon sets. This is in reaction to the discontinuation of software for generating Matrix Representation files (see, e.g., Page, 2002; Chen, Eulenstein, and Fernández-Baca, 2004). The user can choose between Nexus and FASTA format output. Generating a Matrix Representation file takes less than a few seconds for a diverse selection of datasets.

The Presence-Absence file contains a simple matrix showing which unique taxon exists in which tree of the input files.

10 Robinson-Foulds Distances

10.1 Introduction

When comparing two given trees, one might be interested in the similarities and differences between them. A convenient way is to count the number of splits that occur only in one tree or the other, but not in both. This metric is called Robinson-Foulds (RF) distance and was described by David Robinson and Leslie Foulds (Robinson and Foulds, 1981). The RF distance is biased because trees with more taxa (and splits) tend to have a higher distance than trees with fewer taxa (and splits). This bias can easily be corrected by normalizing the RF distance to a range between 0 and 1 through division by the sum of nontrivial splits of the trees.

The following definitions and annotation are taken from Steel (2016). The Robinson-Foulds (RF) distance, $d_{RF}(T, T')$, is a measure to assess the difference of two trees T and T' with identical taxon sets. It is the sum of splits occurring in tree T but not in tree T' and vice versa. If they have identical taxon sets the maximum Robinson-Foulds distance is $2(n-3)$, which is two times the maximum possible number of nontrivial splits (which is $n-3$ in bifurcating trees) where n is the number of taxa. The trivial splits must necessarily exist in both trees and do not contribute to the sum (Steel, 2016).

In short, the Robinson-Foulds distance is the symmetric difference of the split sets of two trees, denoted as

$$d_{RF}(T, T') = |\Sigma(T) \ \Delta \ \Sigma(T')|,$$

where Σ denotes the split set of a tree (Steel, 2016).

10.2 Methods

The Robinson-Foulds distance routine in TreeShredder uses `dynamic_bitset`s to represent the set of nontrivial splits of a tree (0 means the split is absent in the tree, 1 means it is present). Using `DataDepot`'s vector of pointers to all unique splits, each tree's bits are set to 1 at the position of the tree's nontrivial splits in the vector (which is also that split's ID; see also "Nontrivial Splits" block in subsection 5.5.1).

With the `dynamic_bitset`s now representing the presence or absence status of the nontrivial splits of the trees, pairwise XOR operations can be done between them. Counting the resulting bits that are equal to 1 gives the Hamming Distance (see also subsection 6.1.5) between the set of nontrivial splits of two trees, i.e., how many splits would hypothetically need to be removed from either tree such that their sets of nontrivial splits are identical. This is equivalent to the Robinson-Foulds distance.

This part of the algorithm is parallelized with a `static` schedule with block size of 1 (see subsection 4.2.1), which allocates a roughly equal number of pairwise comparisons to each thread by interleaving. Because the number of comparisons for each tree T to each tree T' increases according to a half matrix (Robinson-Foulds distances are symmetric) failing to interleave would lead to a skewed load balance of the threads. This is because the first thread would receive the tip of the half matrix and the last thread would receive the base of the half matrix, which has more pairwise comparisons.

The pairwise Robinson-Foulds distances are written to a file where each line contains a single tree combination (including a tree with itself, which has $d_{RF} = 0$).

There are four columns separated by tabs in the output file:

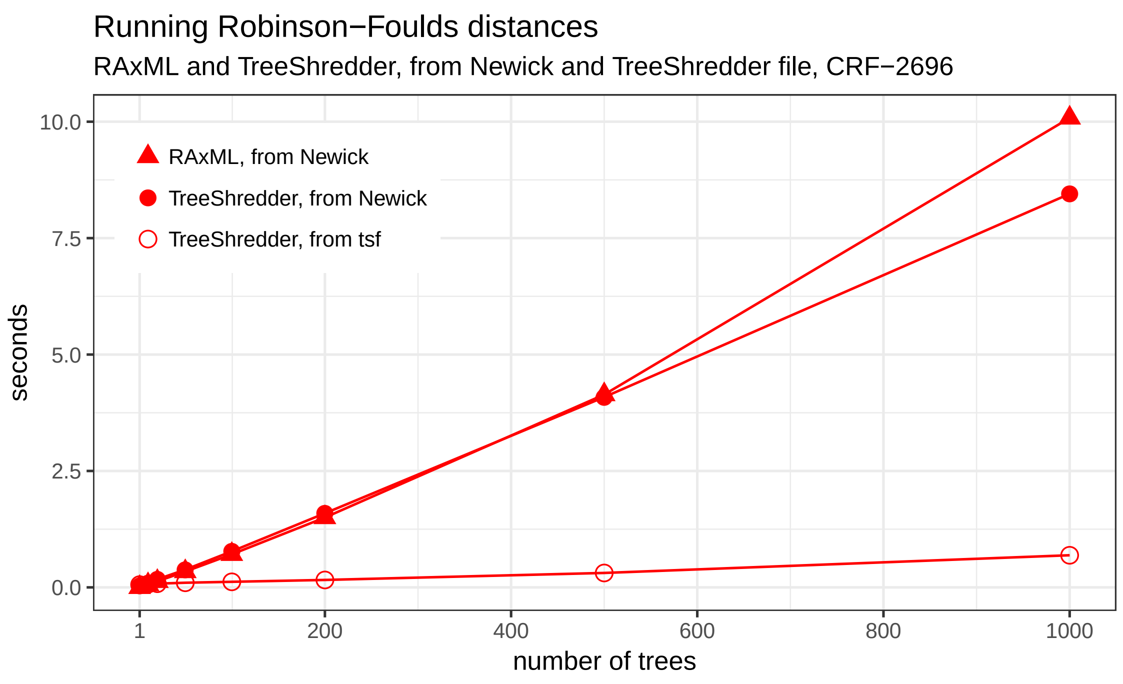
- The 1^{st} column contains the indices of Tree T,
- The 2^{nd} column contains the indices of Tree T',
- The 3^{rd} column contains the absolute RF distances $d_{RF}(T, T')$,
- The 4^{th} column contains the relative RF distances. These are normalized to the range of $[0, 1]$ through division by the sum of existing nontrivial splits in both trees (which, for partially resolved trees, is smaller than $2(n - 3)$).

10.3 Results and Discussion

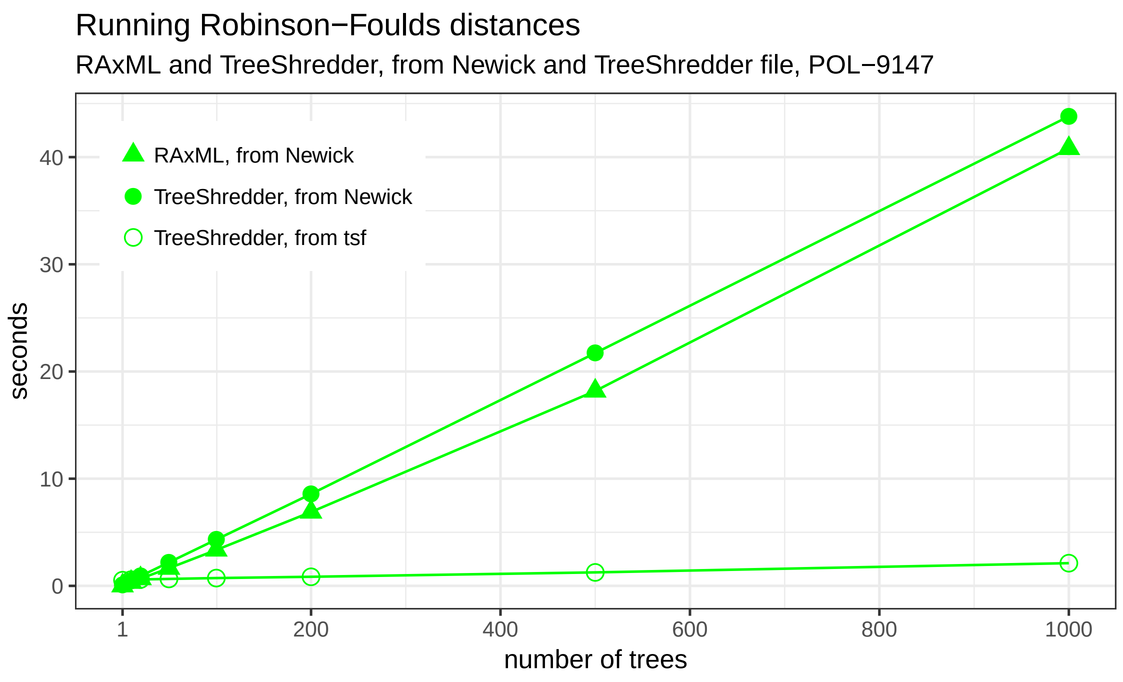
Figs. 10.1a and 10.1b compare the Robinson-Foulds distances runtimes of TreeShredder and RAxML for CRF-2696 and POL-9147, respectively. (WNT-82 was omitted because the runtime was too short to be usefully compared.) For both software, the analysis is started from Newick file input, and for TreeShredder additionally from TreeShredder file input. TreeShredder with TreeShredder file input is fastest by a factor of about 20 for the larger datasets CRF-2696 and POL-9147. The reason is that TreeShredder file input does not require the costly parsing of Newick strings. RAxML and TreeShredder take roughly equal time when started from Newick file input for both datasets. It can also be seen that the runtime for TreeShredder's RF distances routine increases linearly with the number of trees.

10.4 Conclusion

TreeShredder offers the well-established absolute and relative Robinson-Foulds distances feature, which measures the difference between sets of nontrivial splits of two trees. TreeShredder's implementation is as fast or faster than RAxML when started from a Newick file, but orders of magnitude faster when started from a TreeShredder file.



(a) Generating RF distances comparing different programs and file input for CRF-2696: TreeShredder with TreeShredder file input is fastest by an order of magnitude. RAxML and TreeShredder with Newick file input take roughly equal time.



(b) Generating RF distances comparing different programs and file input for POL-9147: TreeShredder with TreeShredder file input is by far the fastest. RAxML and TreeShredder with Newick file input take roughly equal time.

Figure 10.1: Generating RF distances comparing different programs and file input

11 Split Co-Occurrences

11.1 Introduction

As described in chapter 6, counting how often a given nontrivial split occurred in a set of input trees is an established method to assess the robustness of the analysis regarding that split. If the split occurs in many trees derived from multiple sequence alignments created by bootstrapping or jackknifing (see, e.g., Felsenstein, 1986), this is a good indication that the information regarding that split really was present in the original alignment.

But what if the researcher is interested in not only one, but two nontrivial splits? In the case of the above bootstrap values, splits are regarded in a strictly individual way. I.e., having two bootstrap values does not tell (exactly) how often the splits occurred together in the same trees. For two splits A and B with occurrences a and b, respectively, lower and upper bounds for their co-occurrence can be given. It is clear that they co-occur in at least 0 and at most $\min(a, b)$ trees. Only if the sum of bootstrap values of two splits is greater than the number of trees t, the lower bound is greater than 0 and given by $(a + b) - t$ instead. But to narrow down this broad range of possible values, I define and implemented the Split Co-Occurrences (SCO) measure, $SCO(A, B)$, which gives the exact number of co-occurrences of two splits in the trees.

A potential area of application for SCO in phylogenetics is to study the reason why two nontrivial splits do not co-occur (or do not co-occur more often) in the trees even though they are congruent. This could be extended to outright exclude certain splits from consideration for, e.g., consensus trees if they do not co-occur with the other splits with a custom-defined threshold. I.e., the consensus tree would then consist only of splits that really co-occurred in a given minimum number of trees. Similarly, a set of trees could be restricted to only contain those trees that have a high co-occurrence rate of their splits. These potential applications for SCO are out of scope of this master's thesis and are left for others to develop further.

Since there are $(n^2 - n)/2$ pairwise comparisons (all to all), where n is the number of unique nontrivial splits, the Split Co-Occurrences output file can grow very large. For this reason, among others, I implemented the output compression feature in TreeShredder (see also subsection 5.6.2).

11.2 Methods

The Split Co-Occurrences routine in TreeShredder uses a `dynamic_bitset` to represent the set of trees a unique nontrivial split occurs in. Here, the bits of a `dynamic_bitset` correspond to the trees and if the split occurs in the tree the bit is set to 1, if not the bit is 0. To find this relationship of trees and their splits, `DataDepot`'s vector of vectors of a tree's nontrivial splits is used.

With the `dynamic_bitsets` now representing all unique nontrivial splits' set of trees they occur in, pairwise AND operations determine in which trees two unique nontrivial splits co-occur. Counting the bits that are equal to 1 obtains the number of trees they co-occur in.

The pairwise Split Co-Occurrences are written to a file where each line contains a single combination of two unique nontrivial splits (including each split with itself, which is equal to the number of trees the split occurs in for absolute SCO, and equal to 1 for relative SCO and SCO ratios).

There are six columns separated by tabs in the output file:

- The 1^{st} column contains the indices of unique nontrivial split A,
- The 2^{nd} column contains the indices of unique nontrivial split B,
- The 3^{rd} column contains the absolute Split Co-Occurrences $SCO(A, B)$,
- The 4^{th} column contains the relative Split Co-Occurrences. These are normalized to the range of $[0, 1]$ through division by the number of trees.
- The 5^{th} column contains the SCO ratio of split A "in" split B. This is the ratio of the number of trees split A and B co-occur in divided by the number of trees split B occurs in overall.
- The 6^{th} column contains the SCO ratio of split B "in" split A. This is the ratio of the number of trees split A and B co-occur in divided by the number of trees split A occurs in overall.

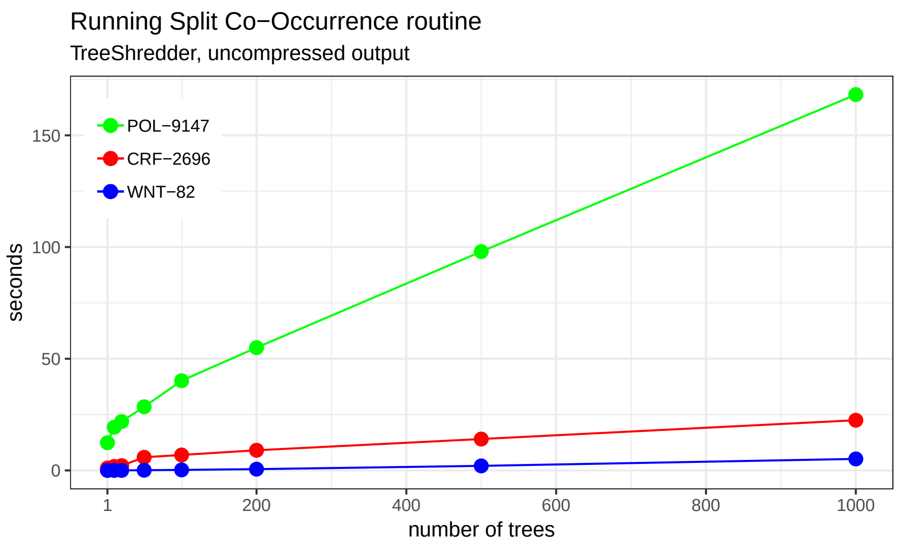
11.3 Results and Discussion

Fig. 11.1a shows the runtime of the Split Co-Occurrences routine. The runtime increases roughly linearly with the number of trees. This is because it is the sum of the linearly increasing runtime of calculating the SCO values (see Fig. 11.1b) and the sub-linearly increasing time of writing those values to the output file (see Fig. 11.1c).

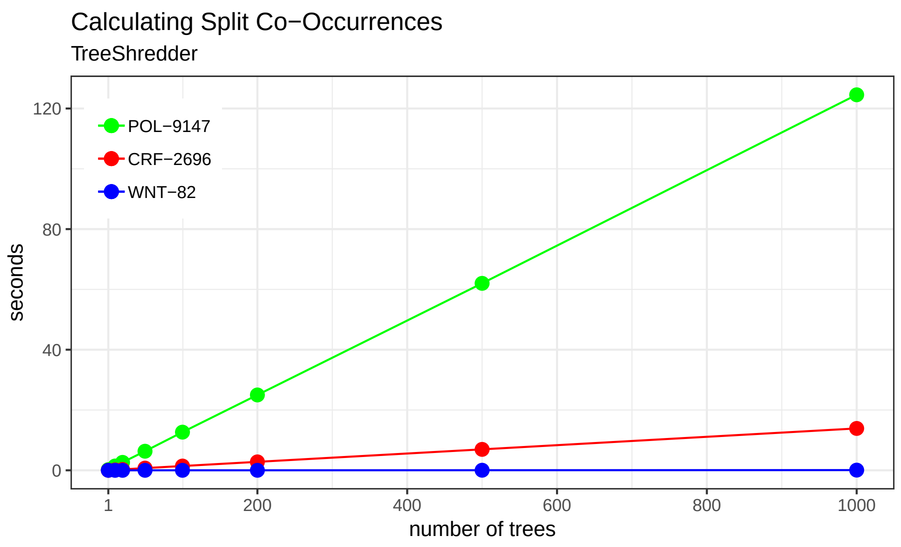
Fig. 11.1b shows the calculation time for the Split Co-Occurrences values. It grows linearly with the number of trees because the `dynamic_bitsets` grow with the number of trees (see section 11.2).

Fig. 11.1c shows the runtime of writing the Split Co-Occurrences values to a file. This should grow quadratically with the number of unique nontrivial splits. However, there is no clear relationship with the number of trees. This likely reflects the non-linear relationship of the number of unique nontrivial splits as a function of the number of trees (see Fig. 3.2a).

(a)



(b)



(c)

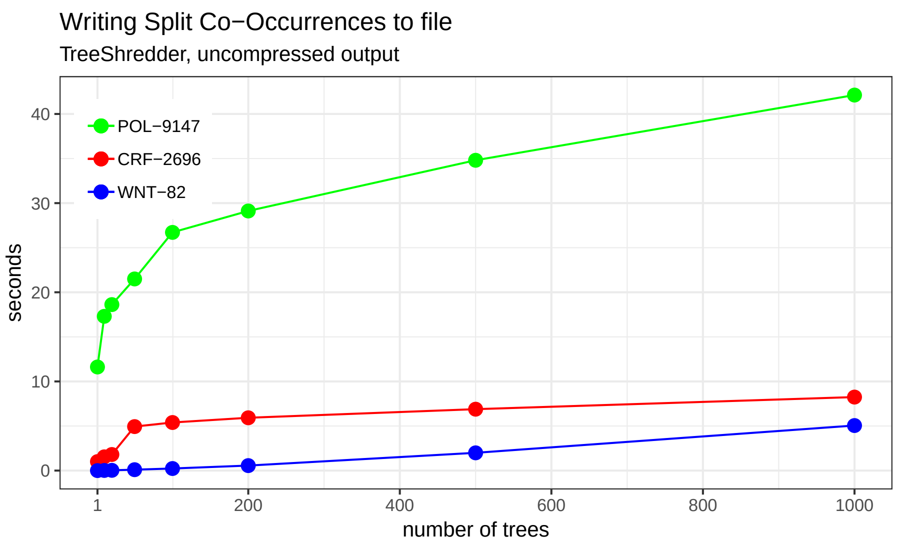
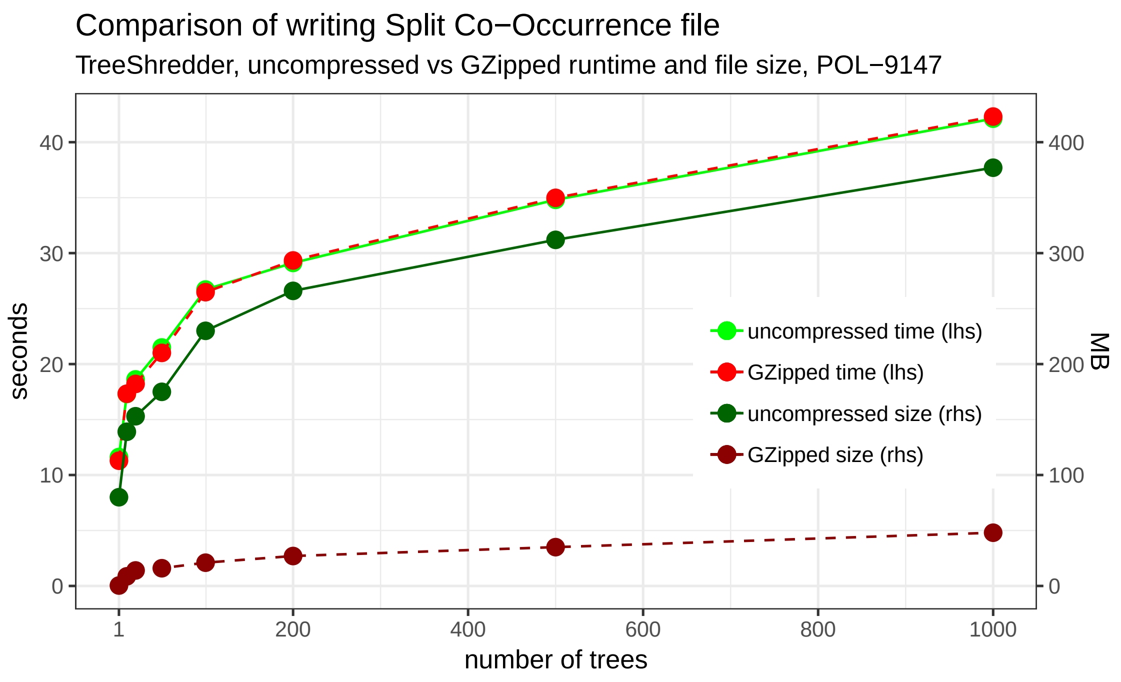
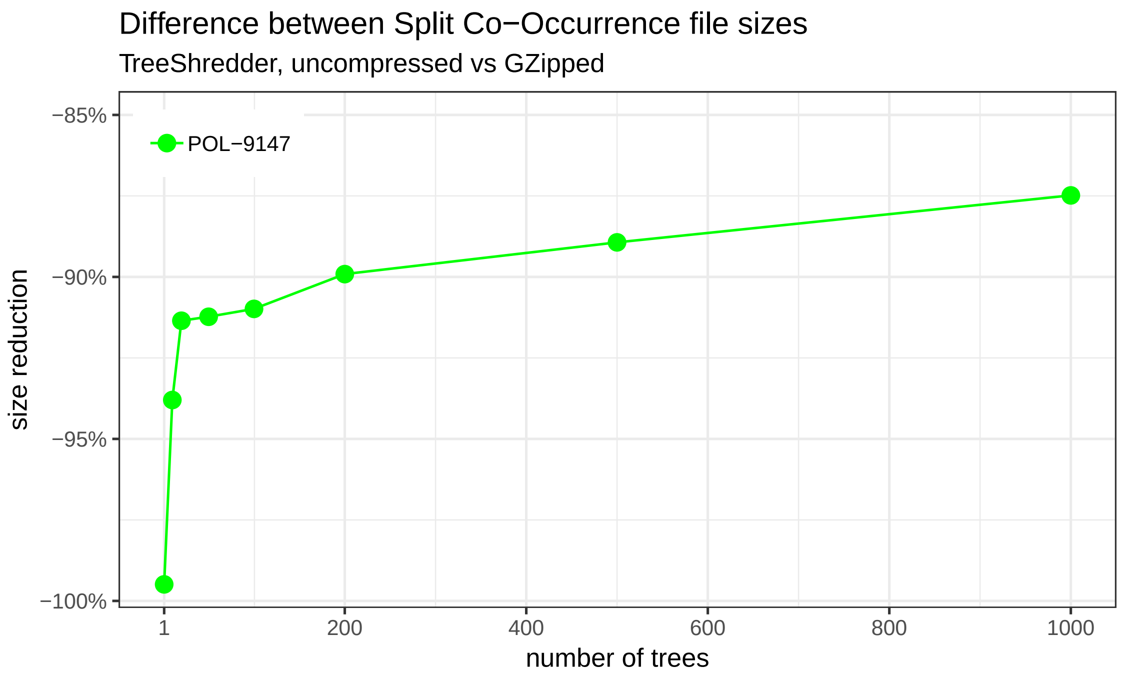


Figure 11.1: TreeShredder runtimes of calculating and writing Split Co-Occurrences to file: The plots show the runtime of (a) the total SCO routine, (b) calculating the SCO values, and (c) writing the SCO values to the uncompressed output file.



(a) Comparison of Split Co-Occurrences file writing time and file sizes: Note the left-hand side (lhs) and right-hand side (rhs) scales. GZipped and uncompressed runtimes (lhs) are virtually identical, while their output file sizes (rhs) are not.



(b) Differences of Split Co-Occurrences file sizes: GZipped files are about one tenth the size of uncompressed files.

Figure 11.2: Comparison of Split Co-Occurrences runtime and output compression

11.3.1 GZip compression

Split Co-Occurrences output files can grow very large as the number of unique nontrivial splits grows large. In this case, it is advised to compress the file output using the `-gz` flag. This is shown in Fig. 11.2, where uncompressed vs GZipped runtime and file size for POL-9147 Split Co-Occurrences is compared. Writing GZipped file output comes without noteworthy runtime changes but decreases output file size substantially by about 90%, e.g., 9 MB vs 139 MB for 10 POL-9147 trees or 48 MB vs 377 MB for 1,000 POL-9147 trees (Fig. 11.2a). The runtime curve in Fig. 11.2a flattens because the number of unique nontrivial splits saturates with an increasing number of trees (compare with Fig. 3.2a).

11.4 Conclusion

I introduced a new feature called Split Co-Occurrences, which shows how often the splits occur with other splits in the trees. Potential fields of research are, e.g., investigating the reason why two congruent splits do not co-occur more often or restricting a set of trees to only those whose splits co-occur sufficiently. Concrete applications are left to be developed by others.

GZipping the Split Co-Occurrences output does not significantly change the runtime but saves about 90% of space (48 MB vs 377 MB for 1,000 POL-9147 trees).

Part IV

Summary

12 Conclusion and Outlook

Here, I summarize TreeShredder's diverse features described in their respective chapters and present an outlook for future work to extend TreeShredder's versatile toolbox even further.

12.1 Summary

If certain precautions are taken, such as minimizing the number of creations and copies of splits by using pointers, speedup by parallel computing can be achieved with the `boost::dynamic_bitset` class. This class offers bitwise standard logical operations such as AND, OR, and XOR, as well as dynamic memory allocation at runtime. However, speedup also depends on the operating system TreeShredder is run on (see chapter 4).

TreeShredder can read the common file formats for tree information input in phylogenetic software: Newick and Nexus files. Additionally, I developed the TreeShredder file, which holds tree information that is useful to TreeShredder in a condensed way (see chapter 5). Investing the time to convert Newick and Nexus files to TreeShredder files once saves time in subsequent TreeShredder analyses because the Newick tree strings need only be parsed once. This is especially true for large datasets, e.g., the GEN-404 dataset comprising millions of trees, where reading the TreeShredder file (in serial) is four times faster than reading the Newick file in parallel using 16 threads.

The Splits Extraction Algorithm extracts nontrivial splits from Newick tree strings along with additional information such as branch lengths and split measures. I extended the algorithm to also extract Nexus metacomments and determine the rootedness of the tree (see chapter 5). Conversely, the Tree Creation Algorithm produces a Newick tree string from a set of compatible splits. With this, TreeShredder can create diverse consensus trees: the strict, semi-strict, majority rule, majority rule extended, relative majority, and the newly introduced global relative majority consensus trees (see chapter 8).

Besides consensus trees, TreeShredder can also map tree information onto reference trees. The user can choose between diverse split measures: abs./rel. support, abs./rel. best incompatible support, abs./rel. difference to best incompatible support, Internode Certainty, and Transfer Bootstrap Expectation (see chapter 6). The abs./rel. best incompatible support and abs./rel. difference to best incompatible support split measures offer an additional way to gauge the quality of a split: weakly supported splits could be less contradicted by incongruent splits than splits that are better supported. I introduced a way to extend the calculation of Internode Certainty to reference trees, where there may be splits that do not occur in the set of splits that are to be mapped onto it. TBE calculation is time consuming and lends itself to parallelization. It uses a gradual indicator function to measure the presence of splits in the trees instead of a binary one. This leads to higher values than simple relative support in deep branches of large phylogenies that could be disrupted by a single taxon with uncertain placement (i.e., rogue taxon). TreeShredder compares favorably against its competitors RAxML and BOOSTER in the calculation of TBE, especially if the analysis is started from a TreeShredder file.

Uniquely among competitor programs, TreeShredder can deal with queries of incomplete splits, where some taxa may occur on either side of the split (see chapter 7). Given a set of incomplete splits, TreeShredder can find all trees from a set that are compatible with those splits and calculate the maximum or sum of support among supportive or incongruent splits in the trees. Additionally, it can determine the congruency status of the splits in the trees with those incomplete splits, i.e., whether they are congruent and supportive, congruent but not supportive, or incongruent.

To support scientists in the field of supertree construction, TreeShredder can generate a Matrix Representation, which has seen decreasing software maintenance and support in recent years (see chapter 9).

TreeShredder calculates the well-established Robinson-Foulds distance (see chapter 10). TreeShredder is as fast or faster than RAxML, depending on whether the analysis was started from a TreeShredder file.

Additionally, I introduced the Split Co-Occurrence, which measures how often each pair of unique splits occurs together in the trees (see chapter 11). Potential applications for this new measure include studying the reason why two congruent splits do not co-occur (more often) in the trees or to outright exclude splits with low co-occurrence from consideration in further analyses.

Any file input or output of TreeShredder can be compressed. Surprisingly, output compression not only saves disk space, it is also faster than writing uncompressed output (see subsection 5.6.2).

TreeShredder can transform IQ-TREE's branch annotation values that are separated by slashes to Nexus metacomments with custom defined names as keys (see subsubsection 6.2.2.1).

12.2 Outlook

In the future, TreeShredder could be extended to annotate specific splits by using (incomplete) splits to add key:value annotation to all splits matching. Additionally, the congruency measures for incomplete splits could be extended to include additional measures, such as the minimum occurrence or the shortest and longest branch among matching splits.

Another feature that could be usefully extended is the consensus tree feature. As described in section 8.4, the compatible sets of splits of the different consensus tree methods are subsets of each other. This means that generating multiple consensus trees by finding their respective compatible sets of splits and calculating their split measures in one go, exploiting their subset relationship, is feasible.

Furthermore, the TreeShredder file feature could be extended to write the nontrivial splits in a custom-defined order, e.g., sorted by their support. This could usefully speed up subsequent analyses that require (many millions of) nontrivial splits in a specific order.

12.3 Concluding Remarks

In this thesis, I introduced features that are unique to TreeShredder. These are: extended Nexus metacomments, incomplete splits and related queries, new support measures (best incompatible and difference to best incompatible support), a way to speed up phylogenetic analyses using the TreeShredder file as convenient starting point, dealing with several million input trees, and Split Co-Occurrence. These unique features, together with a repertoire of established ones, make TreeShredder a versatile and powerful addition to the analysis toolbox in phylogenetics.

References

Adams III, Edward N. (1972). “Consensus Techniques and the Comparison of Taxonomic Trees”. In: *Syst. Zool.* 21, pp. 390–397. DOI: 10.1093/sysbio/21.4.390.

Baum, Bernard R. (1992). “Combining trees as a way of combining data sets for phylogenetic inference, and the desirability of combining gene trees”. In: *Taxon* 41, pp. 3–10. DOI: 10.2307/1222480.

BEAST developers (2017). *NEXUS format metacomments.* URL: https://beast.community/nexus_metacomments (accessed: 2022-6-15).

Boost developers (2017). *Boost 1.66.0 Libraries Documentation.* URL: https://www.boost.org/doc/libs/1_66_0 (accessed: 2022-5-5).

Branstetter, MG, BN Danforth, JP Pitts, BC Faircloth, PS Ward, ML Buffington, MW Gates, RR Kula, and SG Brady (2017). “Phylogenomic Insights into the Evolution of Stinging Wasps and the Origins of Ants and Bees”. In: *Current Biology* 27.7, pp. 1019–1025. DOI: dx.doi.org/10.1016/j.cub.2017.03.027.

Bremer, Kåre (1990). “Combinable component consensus”. In: *Cladistics* 6, pp. 369–372. DOI: 10.1111/j.1096-0031.1990.tb00551.x.

Bryant, David (1997). “Hunting for trees, building trees and comparing trees: Theory and method in phylogenetic analysis”. PhD thesis. Canterbury: Dept. Mathematics, University of Canterbury. DOI: -.

— (2003). “A classification of consensus methods for phylogenetics”. In: *Bioconsensus: Proceedings of Tutorial and Workshop on Bioconsensus II.* DIMACS Series in Discrete Mathematics and Theoretical Computer Science. DIMACS-AMS, pp. 55–66. DOI: 10.1090/dimacs/061/11.

Chen, Duhong, Lixia Diao, Oliver Eulenstein, David Fernández-Baca, and Michael J. Sanderson (2003). “Flipping: A Supertree Construction Method”. In: *DIMACS Series in Discrete Mathematics and Theoretical Computer Science.* Ed. by M. F. Janowitz, F.-J. Lapointe, F. R. McMorris, B. Mirkin, and F. S. Roberts. Vol. 61. Providence, Rhode Island: American Mathematical Society, pp. 135–160. DOI: 10.1090/dimacs/061/10.

Chen, Duhong, Oliver Eulenstein, and David Fernández-Baca (2004). “Rainbow: a toolbox for phylogenetic supertree construction and analysis”. In: *Bioinformatics* 20, pp. 2872–2873. DOI: 10.1093/bioinformatics/bth313.

Chor, Benny and Tamir Tuller (May 2005). “Maximum Likelihood of Evolutionary Trees Is Hard”. In: *Proceedings of the 9th Annual International Conference on Res earch in Computational Molecular Biology (RECOMB 2005).* Vol. 3500. Lecture Notes in Computer Science. New York, USA: ACM Press, pp. 296–310. DOI: 10.1007/11415770_23.

Dagum, Leonardo and Ramesh Menon (1998). “OpenMP: An Industry-Standard API for Shared-Memory Programming”. In: *IEEE Comput. Sci. Eng.* 5, pp. 46–55. DOI: 10.1109/99.660313.

Day, William H. E., D. Johnson, and David Sankoff (1986). “The computational complexity of inferring rooted phylogenies by parsimony”. In: *Math. Biosci.* 81, pp. 33–42.

Faircloth, BC, L Sorenson, F Santini, and Alfaro ME (2013). “A phylogenomic perspective on the radiation of ray-finned fishes based upon targeted sequencing of ultraconserved elements (UCEs)”. In: *PLoS ONE* 8.6, e65923. DOI: `dx.doi.org/10.1371/journal.pone.0065923`.

Felsenstein, Joseph (1978). “The number of evolutionary trees”. In: *Syst. Zool.* 27, pp. 27–33. DOI: `10.2307/2412810`.

— (1985). “Confidence Limits on Phylogenies: An Approach Using the Bootstrap”. In: *Evolution* 39, pp. 783–791. DOI: `10.1111/j.1558-5646.1985.tb00420.x`.

— (1986). “Discussion: Jackknife, Bootstrap and Other Resampling Methods in Regression Analysis”. In: *Ann. Stat.* 14, pp. 1304–1305. DOI: `10.1214/aos/1176350146`.

— (1993). *PHYLIP (Phylogeny Inference Package) version 3.5c.* Distributed by the author. Department of Genetics, University of Washington. Seattle.

— (2004). *Inferring Phylogenies.* Sunderland, Massachusetts: Sinauer Associates, pp. 521–538. ISBN: 0878931775.

Felsenstein, Joseph, James Archie, William H.E. Day, Maddison Wayne, Christopher Meacham, F. James Rohlf, and David Swofford (1986). *The Newick tree format.* URL: `https://evolution.genetics.washington.edu/phylip/newicktree.html` (accessed: 2022-6-26).

Fleischauer, Markus and Sebastian Böcker (2017). “Bad Clade Deletion Supertrees: A Fast and Accurate Supertree Algorithm”. In: *Mol. Biol. Evol.* 34, pp. 2408–2421. DOI: `10.1093/molbev/msx191`.

Gailly, Jean-loup and Mark Adler (2022). *zlib 1.2.12.* URL: `https://zlib.net` (accessed: 2022-5-9).

Horn, JW, Z Xi, R Riina, JA Peirson, Y Yang, BL Dorsey, PE Berry, CC Davis, and KJ Wurdack (2014). “Evolutionary bursts in Euphorbia (Euphorbiaceae) are linked with photosynthetic pathway”. In: *Evolution* 68.12, pp. 3485–3504. DOI: `dx.doi.org/10.1111/evo.12534`.

Kerrisk, Michael (2010). “The Linux Programming Interface - A Linux and UNIX System Programming Handbook”. In: San Francisco: No Starch Press, p. 40.

Kobert, Kassian, Leonidas Salichos, Antonis Rokas, and Alexandros Stamatakis (2016). “Computing the Internode Certainty and Related Measures from Partial Gene Trees”. In: *Mol. Biol. Evol.* 33, pp. 1606–1617. DOI: `10.1093/molbev/msw040`.

Kupczok, Anne, Heiko A. Schmidt, and Arndt von Haeseler (2010). “Accuracy of phylogeny reconstruction methods combining overlapping gene data sets”. In: *Algorithms Mol. Biol.* 5, 37, p. 37. DOI: `10.1186/1748-7188-5-37`.

Lanfear, Rob (2019). *BenchmarkAlignments.* URL: `https://github.com/roblanf/BenchmarkAlignments` (accessed: 2022-6-15).

Lapointe, François-Joseph, Mark Wilkinson, and David Bryant (2003). “Matrix Representations with Parsimony or with Distances: Two Sides of the Same Coin?” In: *Syst. Biol.* 53, pp. 865–868. DOI: `10.1080/10635150390252297`.

Lemoine, Frederic, Jean-Baka Domelevo Entfellner, Eduan Wilkinson, Miraine Dávila Felipe, Tulio De Oliveira, and Olivier Gascuel (2018). “Renewing Felsenstein’s phylogenetic bootstrap in the era of big data”. In: *Nature* 556, pp. 452–456. DOI: 10.1038/s41586-018-0043-0.

Lengfeld, Tobias, Hiroshi Watanabe, Oleg Simakov, Dirk Lindgens, Lydia Gee, Lee Law, Heiko A. Schmidt, Suat Özbek, Hans Bode, and Thomas W. Holstein (2009). “Multiple Wnts are involved in Hydra organizer formation and regeneration”. In: *Dev. Biol.* 328, pp. 186–199. DOI: 10.1016/j.ydbio.2009.02.004.

Levasseur, Claudine and François-Joseph Lapointe (2006). “Total Evidence, Average Consensus and Matrix Representation with Parsimony: What a Difference Distances Make”. In: *Evol. Bioinform. Online* 2, pp. 1–5. DOI: 10.1177/117693430600200018.

Lewis, Paul O. (Nov. 2003). “NCL: a C++ class library for interpreting data files in NEXUS format”. In: *Bioinformatics* 19.17, pp. 2330–2331. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btg319. eprint: https://academic.oup.com/bioinformatics/article-pdf/19/17/2330/539234/btg319.pdf. URL: https://doi.org/10.1093/bioinformatics/btg319.

Maddison, David R., David L. Swofford, and Wayne P. Maddison (1997). “NEXUS: An Extensible File Format for Systematic Information”. In: *Syst. Biol.* 46, pp. 590–621. DOI: 10.1093/sysbio/46.4.590.

Margush, Tim and Fred R. McMorris (1981). “Consensus n-trees”. In: *Bull. Math. Biol.* 43, pp. 239–244. DOI: 10.1016/S0092-8240(81)90019-7.

Minh, Bui Quang, Minh Anh Thi Nguyen, and Arndt von Haeseler (2013). “Ultrafast Approximation for Phylogenetic Bootstrap”. In: *Mol. Biol. Evol.* 30, pp. 1188–1195. DOI: 10.1093/molbev/mst024.

Near, TJ, A Dornburg, RI Eytan, BP Keck, WL Smith, KL Kuhn, JA Moore, SA Price, FT Burbrink, M Friedman, and PC Wainwright (2013). “Phylogeny and tempo of diversification in the superradiation of spiny-rayed fishes”. In: *Evolution* 110.31, pp. 12738–12743. DOI: dx.doi.org/10.1073/pnas.1304661110.

Nguyen, Lam-Tung, Heiko A. Schmidt, Arndt von Haeseler, and Bui Quang Minh (2015). “IQ-TREE: A fast and effective stochastic algorithm for estimating maximum likelihood phylogenies”. In: *Mol. Biol. Evol.* 32, pp. 268–274. DOI: 10.1093/molbev/msu300.

Object Management Group (2017). *OMG® Unified Modeling Language® Version 2.5.1.* URL: https://omg.org/spec/UML/2.5.1/PDF.

OpenMP Architecture Review Board (2018). *OpenMP Application Programming Interface Version 5.0.* URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

Page, Roderic D. M. (2002). “Modified Mincut Supertrees”. In: *Proceedings of the 2nd Workshop on Algorithms in Bioinformatics (WABI 2002).* Vol. 2452. Lecture Notes in Computer Science. New York: Springer, pp. 537–551. DOI: 10.1007/3-540-45784-4_41.

— (2003). “Visualising Phylogenetic Trees Using TreeView”. In: *Current Protocols in Bioinformatics.* Ed. by Andreas D. Baxevanis, Daniel B. Davison, Roderic D. M. Page, Gary Stormo, and Lincoln Stein. Supplement 1. New York, USA: Wiley and Sons, pp. 6.2.1–6.2.15. DOI: 10.1002/0471250953.bi0602s01.

Penny, D., M. D. Hendy, and B. R. Holland (2007). “Phylogenetics: Parsimony, Networks, and Distance Methods”. In: *Handbook of Statistical Genetics*. Ed. by D. J. Balding, M. Bishop, and C. Cannings. John Wiley & Sons, Ltd. Chap. 16, pp. 489–532. ISBN: 9780470061619. DOI: `https://doi.org/10.1002/9780470061619.ch16`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470061619.ch16`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470061619.ch16`.

Pyron, RA and JJ Wiens (2011). “A large-scale phylogeny of Amphibia including over 2800 species, and a revised classification of extant frogs, salamanders, and caecilians”. In: *Molecular Phylogenetics and Evolution* 61.2, pp. 543–583. DOI: `dx.doi.org/10.1016/j.ympev.2011.06.012`.

Ragan, Mark A. (1992). “Phylogenetic inference based on matrix representation of trees”. In: *Mol. Phylogenet. Evol.* 1, pp. 53–58. DOI: `10.1016/1055-7903(92)90035-F`.

Rambaut, Andrew (2010). *FigTree v1.3.1.* URL: `http://tree.bio.ed.ac.uk/software/figtree` (accessed: 2022-6-10).

Ranwez, V, F Delsuc, S Ranwez, K Belkhir, MK Tilak, and EJ Douzery (2007). “OrthoMaM: a database of orthologous genomic markers for placental mammal phylogenetics”. In: *BMC Evol. Biol.* 7, 241, p. 241. DOI: `10.1186/1471-2148-7-241`.

Rauber, T. and G. Rünger (2012). *Parallele Programmierung.* eXamen.press. Springer Berlin Heidelberg. ISBN: 9783642136047. URL: `https://books.google.at/books?id=sQydgXyf3kMC`.

Reddy, Sushma, Rebecca T. Kimball, Akanksha Pandey, Peter A. Hosner, Michael J. Braun, Shannon J. Hackett, Kin-Lan Han, John Harshman, Christopher J. Huddleston, Sarah Kingston, Ben D. Marks, Kathleen J. Miglia, William S. Moore, Frederick H. Sheldon, Christopher C. Witt, Tamaki Yuri, and Edward L. Braun (2017). “Why do phylogenomic data sets yield conflicting trees? Data type influences the avian tree of life more than taxon sampling”. In: *Syst. Biol.* 66, in press. DOI: `10.1093/sysbio/syx041`.

Robinson, David F. and Leslie R. Foulds (1981). “Comparison of phylogenetic trees”. In: *Math. Biosci.* 53, pp. 131–147. DOI: `10.1016/0025-5564(81)90043-2`.

Ross, Howard A. and Allen G. Rodrigo (2004). “An assessment of matrix representation with compatibility in supertree construction”. In: *Phylogenetic Supertrees: Combining Information to Reveal the Tree of Life.* Ed. by Olaf R. P. Bininda-Emonds. Dordrecht, The Netherlands: Kluwer Academic, pp. 35–63.

Salichos, Leonidas, Alexandros Stamatakis, and Antonis Rokas (2014). “Novel Information Theory-Based Measures for Quantifying Incongruence among Phylogenetic Trees”. In: *Mol. Biol. Evol.* 31, pp. 1261–1271. DOI: `10.1093/molbev/msu061`.

Sanderson, Michael J., Michelle M. McMahon, and Mike Steel (2011). “Terraces in Phylogenetic Tree Space”. In: *Science* 333, pp. 448–450. DOI: `10.1126/science.1206357`.

Schmidt, Heiko A. (2003). “Phylogenetic Trees from Large Datasets”. PhD thesis. Düsseldorf, Germany: Universität Düsseldorf. DOI: `-`.

— (2007). *GenerateTrees, version 0.5.* URL: `http://www.cibiv.at/software/generatetrees` (accessed: 2023-4-8).

Semple, Charles and Mike Steel (2003). *Phylogenetics.* Vol. 24. Oxford Lecture Series in Mathematics and Its Applications. Oxford, UK: Oxford University Press. ISBN: 0198509421.

Shannon, Claude Elwood (1948). "A Mathematical Theory of Communication". In: *The Bell System Technical Journal* 27, pp. 379–423. DOI: `10.1002/j.1538-7305.1948.tb01338.x`.

Siek, Jeremy and Chuck Allison (2017). "Dynamic Bitset". In: URL: `https://www.boost.org/doc/libs/1_66_0` (accessed: 2022-5-5).

Stamatakis, Alexandros (2006). "RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models". In: *Bioinformatics* 22, pp. 2688–2690. DOI: `10.1093/bioinformatics/btl446`.

Steel, Mike (2016). *Phylogeny: Descrete and Random Processes in Evolution.* Philadelphia: SIAM, p. 25.

Varga, T., K. Krizsán, C. Földi, B. Dima, M. Sánchez-García, S. Sánchez-Ramírez, et al. (2019). "Megaphylogeny resolves global patterns of mushroom evolution". In: *Nature Ecology and Evolution.* DOI: `dx.doi.org/10.1038/s41559-019-0834-1`.

Whelan, S, PI de Bakker, E Quevillon, N Rodriguez, and N Goldman (2006). "PANDIT: an evolution-centric database of protein and associated nucleotide domains with inferred trees". In: *Nucleic Acids Res.* 34, pp. D327–D331. DOI: `10.1093/nar/gkj087`.