TreeScaper CLV Manual

Version 2.0.0-alpha.2

November 21, 2021

Jeremy Ash[2,4],
Jeremy M. Brown[2],
David Morris[2],
Guifang Zhou[2],
Wen Huang[1],
Melissa Marchand[3],
Paul Van Dooren[1],
Kyle A. Gallivan[3],
and Jim Wilgenbusch[5]

[1]Department of Mathematical Engineering, ICTEAM,
Université catholique de Louvain, Belgium,
wen.huang@uclouvain.be

[2] Department of Biological Sciences and Museum of Natural Science,
Louisiana State University, Baton Rouge, LA, USA

[3] Department of Mathematics,
Florida State University, Tallahassee, FL, USA

[4] Current Address: Bioinformatics Research Center,
North Carolina State University, Raleigh, NC, USA

[5] Minnesota Supercomputing Institute,
University of Minnesota, Minneapolis, MN, USA

TABLE OF CONTENTS

1 INTRODUCTION

Phylogenetic trees are now routinely inferred from enormous genome-scale data sets, revealing extensive variation in phylogenetic signal both within and between individual genes. This variation may result from a wide range of biological phenomena, such as recombination, horizontal gene transfer, or hybridization. It may also indicate stochastic and/or systematic error. However, current approaches for summarizing the variation in a tree set typically condense it into point estimates, such as consensus trees, resulting in extensive loss of information.

We have written TreeScaper to provide a set of visual and quantitative tools for exploring and characterizing the full complement of phylogenetic information contained in a tree set. These tools can be broadly categorized into three types: (1) utilities for calculating basic information about topologies and bipartitions, (2) visualization of treespace in $2-$ or $3-$dimensional space through non-linear dimensionality reduction (NLDR), and (3) detection and delineation of distinct communities of trees.

Tree objects – Much of TreeScaper's functionality requires calculating distances between trees, transforming distances into affinities, translating trees into their component bipartitions, and summarizing how these bipartitions are distributed across trees (i.e., their variances and covariances). However, this information can also be useful in its own right. Therefore, TreeScaper provides a set of built-in utilities to calculate a range of useful tree- and bipartition-related summaries. Once calculated, these values may be used for other tasks in TreeScaper or may be written to file for use in other applications.

NLDR – One way to visually explore tree sets is to plot a $2-$ or $3-$dimensional representation of treespace using non-linear dimensionality reduction (NLDR; Fig. 1). This approach was first suggested for the visualization of phylogenetic trees by Amenta and Klingner, 2002 and Hillis et al., 2005, and recently extended by Wilgenbusch et al., 2017. The general idea behind NLDR performed on distance matrix $D \in \mathbb{R}^{n \times n}$ is to find a lower dimensional representation of the relationships among objects that best preserves the true distances between them, resulting in coordinate matrix $X \in \mathbb{R}^{n \times k}$ which represents n points in $k-$ dimensional Euclidean space. TreeScaper implements several stress functions to assess how the input distances should be optimally represented in lower dimensional space [e.g., Normalized stress, Kruskal-1 stress, nonlinear mapping (NLM) stress, and Curvilinear Components Analysis (CCA) stress], and several optimization algorithms for finding the best low-dimensional representation given a chosen stress function (e.g., Gauss-Seidel-Newton, stochastic gradient descent, and simulated annealing).

Community Detection – When tree sets are summarized by condensing them into a single point estimate, one of the key pieces of lost information is whether distinct phylogenetic "signals" exist in the set. Distinct signals can be created by a variety of biological processes like coalescence within a species, incomplete lineage sorting between species, horizontal gene transfer, hybridization, and migration. Artificial signals can also be created by systematic error during the process of phylogenetic inference. The process of detecting distinct signals in a tree set and assigning trees to one or more groups can be formalized in many different ways (e.g., [Gori et al., 2016; Lewitus and Morlon, 2016]). TreeScaper uses a graph-theoretic approach known as community detection. Roughly speaking, communities are parts of a graph with dense, positive connections between nodes within a community and sparse or negative connections between nodes in different communities. By formalizing the problem of detecting distinct phylogenetic signals as a community detection problem,

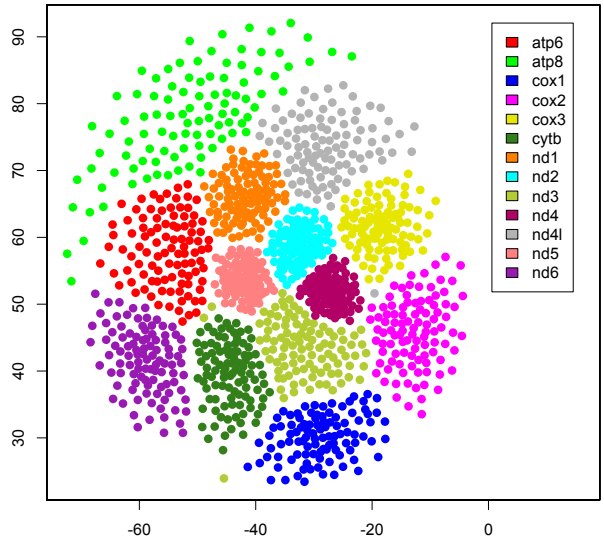


Figure 1: A 2−dimensional representation of treespace generated by non-linear dimensionality reduction (NLDR) for a set of $1,300$ trees sampled from individual Bayesian analyses of 13 mitochondrial protein-coding genes in squamates [Castoe et al., 2009]. Each point represents a tree sampled from the posterior distribution of one gene. One hundred trees were sampled from each posterior distribution. Points are colored by gene. Plot created using R.

we can draw from a large body of existing work in graph theory.

TreeScaper implements several models of community detection methods on weighted graph stored as adjacency matrix [e.g., Configuration Null Model (CNM), Constant Potts Model (CPM), Erdos-Rényi Null Model (ERNM) and No Null Model(NNM)]. The general idea behind community detection performed on adjacency matrix $\mathrm{Adj} \in \mathbb{R}^{n\times n}$ is to find a sparse adjacency matrix $\widetilde{\mathrm{Adj}} \in \mathbb{R}^{n\times n}$ and permutation matrix $P \in \mathbb{R}^{n\times n}$ that best preserves the true graph as well as obtaining $P\widetilde{\mathrm{Adj}}P^T$ as block diagonal matrix, where the blocks are communities found. TreeScaper assumed the input weighted graph to be two distinct types of networks. In the first, nodes in the graph correspond to individual trees in the tree set and the edges between these nodes are weighted by the affinity between these trees (Fig. 2).

Affinity can be calculated in different ways, but it broadly corresponds to the converse of distance – a pair of trees separated by a small distance have high affinity, while a pair separated by a large distance have low affinity. Communities in these networks should intuitively correspond to sets of trees that are topologically similar to one another and topologically dissimilar to trees in other communities. Topological affinity networks have received some previous attention in attempts to define distinct phylogenetic signals [Stockham et al., 2002; Gori et al., 2016; Lewitus and Morlon, 2016].

The other type of network assumed by TreeScaper uses nodes to represent individual bipartitions, with edge weights corresponding to the covariance in presence/absence of bipartition pairs

across trees in the tree set (Fig. 3). When bipartitions are very common or very rare in the tree set, they tend to have weak covariances with all other bipartitions. However, if two bipartitions are present at intermediate frequencies and they are always found in the same trees or always found in different trees, they will have strong positive or strong negative covariances, respectively. Communities can be identified in bipartition covariance networks just like in topological affinity networks, with the distinction that bipartition covariance networks may contain negative edge weights. In this case, communities should consist of sets of bipartitions that tend to have strong positive covariances, while bipartitions in separate communities should tend to have strong negative covariances (Fig. 3).

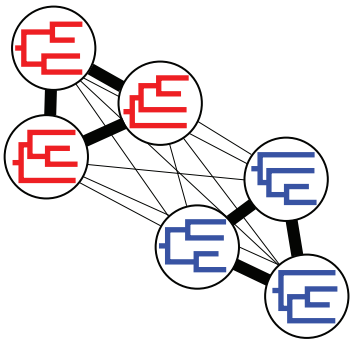


Figure 2: A cartoon topological affinity network. Circles are nodes in the network, each of which corresponds to one tree from a tree set. Edges represent the affinity (or similarity) between the trees, with thicker lines indicating greater affinity. Tree colors correspond to one intuitive definition of communities in this network.

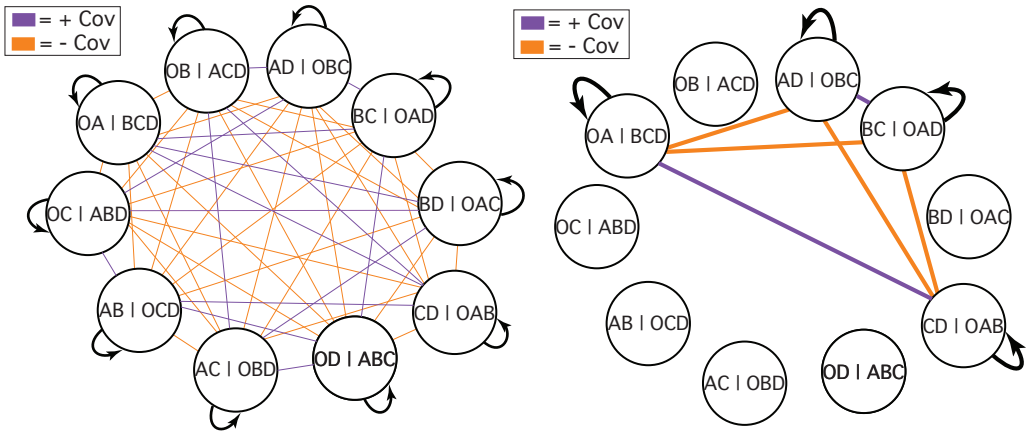


Figure 3: Two example bipartition covariance networks for sets of unrooted, 5−taxon trees. The network on the left corresponds to a tree set with a uniform distribution of frequencies across all possible tree topologies. Some weak covariances exist in this network, because some pairs of bipartitions are mutually exclusive and are therefore found together less often than would be expected based on their frequencies alone. The network on the right corresponds to a tree set with only two topologies present at equal frequencies.

2 QUICKSTART TUTORIAL with CLVTreeScaper

2.1 Getting Started

2.1.1 Installation

The Quickstart Tutorial uses a command line version(CLV) of TreeScaper with bash interface, which requires users to compile source code for their own machine. There are also pre-compile executable binary for the Mac and Linux abailable from the TreeScaper website at:

https://github.com/TreeScaper/TreeScaper

After compiled, put the executable binary `CLVTreeScaper` under the folder with necessary parameter files `nldr_parameters.csv` and `dimest_parameters.csv`.

2.1.2 Example Tree Set

The tree set that will be used throughout this tutorial is titled `1000bp1L.nex`, placed under subfolder `/test/`. The alignment used to generate this set of bootstrap trees was simulated such that half the sites were generated using one topology, while the other half were generated using another. "guide_tree1.pdf" and "guide_tree2.pdf" show the two topologies, corresponding to the first and second halves of the alignment, respectively. Bipartitions that conflict between these topologies are in color. A bootstrap analysis was then performed in Garli, to produce the 100 trees in this tree set. In this tutorial, we will analyze this tree set in TreeScaper to explore the two conflicting signals present in the data.

2.2 Basic Computations

To obtain the bipartitions information of the tree set. Run:

```
./CLVTreeScaper -trees -f test/1000bp1L.nex -w 0 -r 0
-o Cova -post test
```

The output file `Bipartition_Count_test.out` is the list of all bipartitions appeared in the tree set with their bitstring representation and appeared times.

The output file `Bipartition_test.out` is the listed form of the sparse Bipartition matrix.

The output file `Covariance_test.out` is the covariance matrix of all appeared bipartitions.

To obtain certain kind of tree distance, run:

```
./CLVTreeScaper -trees -f test/1000bp1L.nex -w 1 -r 0 -dm URF
-o Dist -post test
```

The output file `Distance_test.out` is the unweighted Robinson-Foulds distance matrix of all trees.

To obtain the consensus trees, run: `./CLVTreeScaper -trees -f test/1000bp1L.nex -w 1 -r 0 -o Consensus -post test`

The output file `Consensus_test.out` is the Majority consensus trees in Newick format.

2.3 Visualizing Tree Space

In order to visualize the tree set, we perform NLDR to $\mathbb{R}^{n\times 3}$ so that we obtain n points with $3-$tuple of coordinates. Those points generate the Eucldean distance matrix that best approximate the tree distance matrix we obtain from the previous step. To perform the NLDR on tree distance matrix, run `./CLVTreeScaper -nldr -f test/Distance_test.out -d 3 -post NLDR_test`

The output file `Coordinates_NLDR_test.out` is the coordinates matrix $X \in \mathbb{R}^{n\times 3}$. Each row is a point in $\mathbb{R}^3$ representing one tree. We then may use any convenient plotting software to plot each row of X in $3-$ sapce.

The output file `Distance_NLDR_test.out` is the Euclidean distance matrix generated by X.

3 INSTALLATION

Source code of **TreeScaper v2.0.0-alpha.2** for Mac or Linux can be downloaded from

https://github.com/TreeScaper/TreeScaper
under `new_version` branch.

CLVTreeScaper requires a CLAPACK properly installed and linked on your machine. CLAPACK-3.2.1 has been attached to this repository. You may also download here. See detailed instruction on using BLAS library optimized for your machine in CLAPACK/README.install at step (4).

For a fast default installation, you will need to

- Clone TreeScaper repository from GitHub	(see step 1 below)
- Relocate CLAPACK-3.2.1 and modify CLAPACK make.inc file	(see step 2 below)
- Modify TreeScaper makeCLVTreeScaper.inc file	(see step 2 below)
- Make CLAPACK library	(see step 3 below)
- Make CLVTreeScaper binary	(see step 3 below)

Procedure for installing CLAPACK.

(1) `git clone https://github.com/TreeScaper/TreeScaper.git` to build the following directory structure:

`TreeScaper/README.install`	
`TreeScaper/makeCLVTreeScaper.inc`	compiler, compile flags and library definitions for TreeScaper.
`TreeScaper/CLAPACK-3.2.1/`	CLAPACK attached in TreeScaper.
`TreeScaper/CLAPACK-3.2.1/make.inc`	compiler, compile flags and library definitions, for TreeScaper.

(2) Move `/CLAPACK-3.2.1` outside TreeScaper and modify `/CLAPACK-3.2.1/make.inc`. For default installation, you need to only modify the OS postfix name `PLAT` in `/CLAPACK-3.2.1/make.inc`. For advanced installation, please refer to `/CLAPACK-3.2.1/README.install`

Update the path of CLAPACK: `CLAPPATH` in `makeCLVTreeScaper.inc` and make sure the OS postfix name is consistent with `CLAPACK` setting, i.e. `PLAT` in `/CLAPACK-3.2.1/make.inc` and in `makeCLVTreeScaper.inc` must be the same.

==========

(2)' If there is a `CLAPACK` already built in your machine. Make sure it has the following directory structure:

`CLAPACK/BLAS/`	C source for BLAS
`CLAPACK/F2CLIBS/`	f2c I/O functions (libI77) and math functions (libF77)
`CLAPACK/INSTALL/`	Testing functions and pre-tested make.inc files for various platforms.
`CLAPACK/INCLUDE/`	header files - clapack.h is including C prototypes of all the CLAPACK routines.
`CLAPACK/SRC/`	C source of LAPACK routines

Update the path of `CLAPACK`: `CLAPPATH` in `makeCLVTreeScaper.inc` and check the OS postfix name of `lapack_XXX.a` and `blas_XXX.a` and modify `PLAT` in `makeCLVTreeScaper.inc`

For example, if the naming is lapack_MAC.a and lapack_MAC.a then, modify
`PLAT = _LINUX`
in `makeCLVTreeScaper.inc`. If the naming is `lapack.a` and `blas.a`, modify
`PLAT =`
in `makeCLVTreeScaper.inc`.

==========

(3) Go to TreeScaper directory. To install the CLAPACK, run `make CLAPACK`

To compile the TreeScaper, run `make` or `make CLVTreeScaper`.

You may move the binary CLVTreeScaper to other location for your, convenience. Make sure you also move the default parameters files nldr_parameters.csv and dimest_parameters.csv to maintain the structure:

`/CLVTreeScaper`	the CLVTreeScaper binary
`/nldr_parameters.csv`	parameters for nldr routines
`/dimest_parameters.csv`	parameters for dimension estimation routines

To obtain the latest stable version of CLVTreeScaper, update the 'master' GitHub branch,

1) Keep your customized makeCLVTreeScaper.inc file.

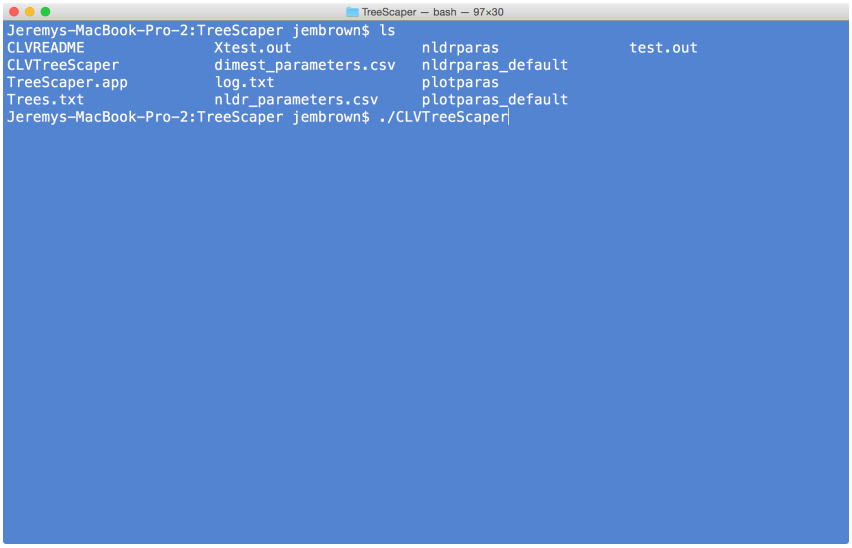
2) Run `git pull`

3) If the makeCLVTreeScaper.inc is overwritten, restore your customized version.

4) If there is no change on CLAPACK side, which is usually the case, run `make` to get the new binary.

Note. There is also a 'test' branch developing new features, working on improvements and fixing certain bugs. The 'test' branch is not guaranteed to be stable.

4 COMMANDS



TreeScaper v2.0.0-alpha.2 only has the tree module completely rewritten and available to users. Therefore, we only include the commands related to tree module. For other functionality, users can refer to the command available on **TreeScaper v1.2.1**.

(1) `-trees`

In this mode, users can compute a majority rule/strict consensus tree, distance matrix, bi-partition matrix, covariance matrix, affinity matrix, or detect communities in an affinity or covariance network. Relevant arguments include (default option are given in red):

-f: Provide the name of the file that contains the data

-w: Indicate whether trees are weighted. Options are:

- **'1':** weighted
- **'0':** unweighted

-r: Indicate whether trees are rooted. Options are:

- **'1':** rooted
- **'0':** unrooted

-o: This option is used to indicate what output the user is interested in. Options are:

- **'Consensus'**
- **'Dist'**
- **'Cova'**

-post: Suffix name of file. Options are:

- **'none'**
- **'time'**

'AnyOtherString': filename will be attached with `_AnyOtherString`

When outputting a bipartition matrix (-o BipartMatrix):

-bfm: Bipartition matrix output type. Options are:

'list': Output sparse matrix in the form (row, column, value)

'matrix': Output as if it is a full matrix

When computing a majority-rule or strict consensus tree (-o Consensus), use the -if, -ct, and/or -cfm flags:

-if: The name of a list file. Consensus tree computations will only consider the trees indicated in the file.

-ct: The type of consensus tree to be computed. Options are:

'Majority': Majority consensus tree

'Strict': Strict consensus tree

-cfm: Format of the consensus tree file. Options are:

'Newick'

'Nexus'

When computing a distance matrix (-o Dist):

-dm: Indicates the distance metric. Options are:

'URF': Unweighted Robinson-Foulds distance

'RF': Weighted Robinson-Foulds distance

Examples of command-line runs *Options specified by the are given inside braces. When specific alternatives are available, they are separated by commas (e.g., {option1,option2}). When numbers can be specified anywhere in a continuous range, the bounds of the range are separated by a dash (e.g., {0-1}).*

Compute a Consensus Tree:

```
./CLVTreeScaper -trees -f {trees.txt} -w {1,0} -r {1,0}
-o Consensus -if IndicesFileName -ct {Majority,Strict}
-cfm {Newick,Nexus} -post {none, time, AnyString}
```

Compute Distance Matrix:

```
./CLVTreeScaper -trees -f {trees.txt} -w {1,0} -r {1,0}
-o Dist -dm {URF,RF,Mat,SPR} -post {none, time, AnyString}
```

Compute Covariance Matrix:

```
./CLVTreeScaper -trees -f {trees.txt} -w {1,0} -r {1,0}
-o Cova -post {none, time, AnyString}
```

5 IMPLEMENTATION DETAILS

TreeScaper v2.0.0-alpha.2 targets at reorgainizing all low level data structure into more self-contained and flexible structures. The complicated data strcutre implemented in **TreeScaper v1.2.1** entangled with the big tree objects has been sorted out, partited and reimplemented as self-contained structure. For example, the tree class no longer process the distance matrix and coordinate matrix. They both go to an object implemented for NLDR module or Community Detection module.

TreeScaper v2.0.0-alpha.2 also includes more complicated basic data structure like lower-triangular matrix, column major sparese matrix which fast access in both row and column. These structures significantly improve the code performance.

The following table gives an comparsion bewteen **TreeScaper v2.0.0-alpha.2** and **TreeScaper v1.2.1** with weighted test tree sets

Tree set	Tree numebr	Taxa size	Distinct bipart. #
Plant	2064	52	37198
Cat	23590	27	635
Mammal	1000	116	12734

Table 1: Test Tree Sets

		Plant		Cat		Mammal	
		Old	New	Old	New	Old	New
RF-dist.	Time(s)	1245.22	6.15625	Over 20 hrs	1160.0925	829.531	2.7812
	Memory(MB)	955.8	693.8	8925.5	4488.9	242.5	137.2
Covari.	Time(s)	697.301	21.4219	96.031	1.4843	342.359	2.5313
	Memory(MB)	11483.73	5955.7	606	543.6	1470.2	997.1

5.1 Computation Modules and Command Structure

TreeScaper can be roughly divided into 4 self-contained computation modules: **preprocessing and computation of trees**, denoted tree module, **large scale NLDR**, denoted NLDR module, **community detection**, denoted CD module, and other preprocessing for adjaceny matrix used in CD module, denoted adjaceny module.

A typical workflow invokes the tree module that reads the provided treeset and generated a symmetric matrix representing the distance matrix or covariance network. Then the adjaceny module, if needed, will generate an adjaceny matrix from the output of tree module and send it to CD module. If visualization is specified, then NLDR module will also take the output of tree module and produce a set of points on $\mathbb{R}^2$ or $\mathbb{R}^3$ that represents the provided treeset. Then, graphing module implemented outside **TreeScaper**, for example, the graphing code on CloudForest, will take the NLDR output and CD output for further visualization.

Fig.4 illustrates the relationships between all computation modules.

5.2 Tree Module Implementation

This module is responsible for reading provided tree set and compute its related objects correctly and efficiently.

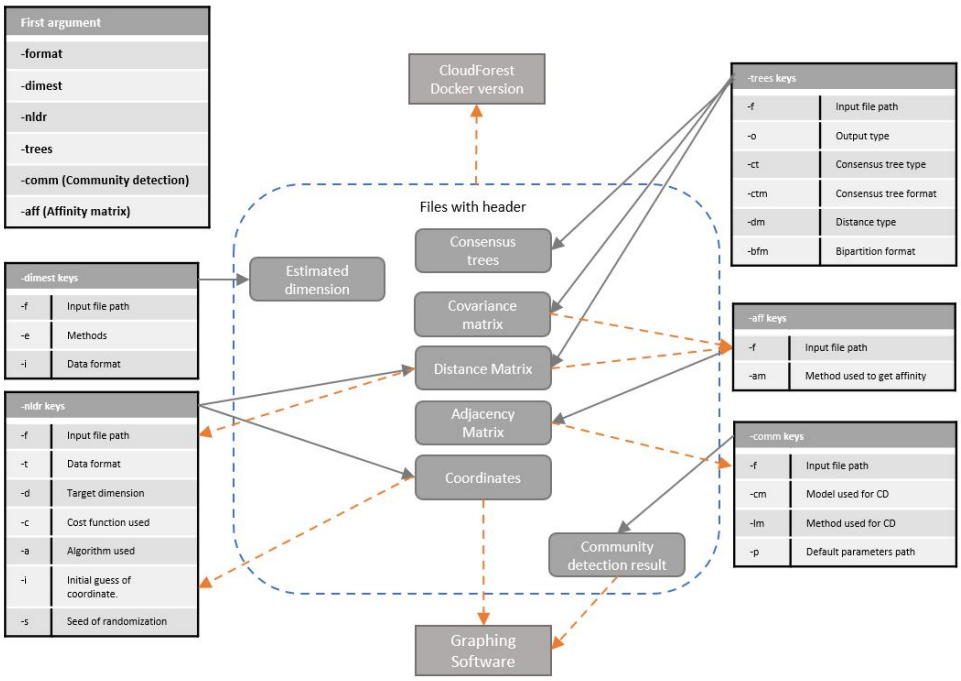


Figure 4: Command List and routine structure based on file.

5.2.1 Tree Implementation

The fundamental data for tree module is obviously the tree itself. **TreeScaper** supports tree-set data formatted in Newick or Nexus form, which uses one ACSII string to represents a tree.

Here is an example of a tree in Newick form.

```
((1:$w_1$, (3:$w_2$, 0:$w_3$):$w_4$):$w_5$, (4:$w_6$,2:$w_7$):$w_8$).
```

The corresponding tree is given in Fig.5.

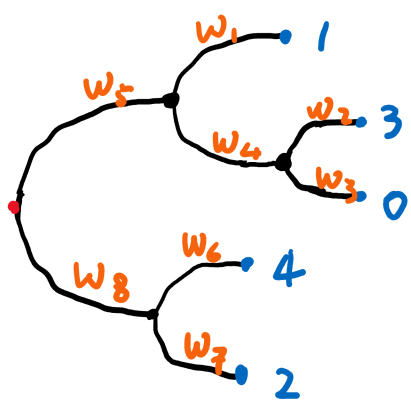


Figure 5: Illustration of a weighted tree in Newick form.

Red: root indicated in Newick form; blue: leaf nodes; orange: weights of edges

It is easy to see that one tree can have multiple Newick form, by exchanging the order of node in the same level:

((4:w_6,2:w_7):w_8, (1:w_1, (3:w_2, 0:w_3):w_4):w_5)
or
((1:w_1, (0:w_3, 3:w_2):w_4):w_5, (4:w_6,2:w_7):w_8)
that both represent the tree in Fig.5.

For a binary rooted tree with N taxa, there will be $N-1$ internal nodes which yields 2^{N-1} Newick representations for one tree, which is unacceptable for computation task. Therefore, we need a unique representation that can be computed from Newick form and possibly be resuable in later tasks.

A comment solution implemented in **TreeScaper 1.1** is to label all internal node and then keep the all "parent-to-child" relationship present in tree. When comparing two trees/subtrees, a routine that examines the "parent-to-child" relations associated to some given node is required.

Bipartition.

A more efficient solution is given as bipartition representation. Note that the comment solution is essentially labelling all edges presents in trees by the linked list structure used to store the tree. Also note that later computation requires the bipartition information of edges.

Definition 5.1. A bipartition of a tree edge is the bipartition on the leaf-set given by the leaf-sets of two resulting trees after removing that edge.

For example, the edge with weight w_5 in Fig.5 corresponds to the bipartition

$$(\{0,1,3\},\{2,4\}).$$

We point out that storing bipartitions of all edges in a tree is effectively labeling all edges and, additionally, this set of bipartitions uniquely represents the tree, i.e., the tree can be recovered from the set of biparitions.

The recovered algorithm is currently not implemented in TreeScaper since no later computation require information other than bipartitions.

Root and dummy leaf.

Please note that the edge correspond to w_5 and the edge corresponds to w_8 seem to represent a same bipartition, but in phylogenetic interpretation, which subtree the root fell into distinguish differnet bipartitions, i.e., w_5 and w_8 may correspond to **different** bipartitions. To handle this case, we intrepret the root as an dummy leaf as illustrated in Fig.6.

Also note that if an unrooted binary tree is provided in Newick form that yield rooted tree like Fig.5, which is usually seen in appliactions, we have $w_5 = w_8$, which is consistent with the intrepretation of same bipartition.

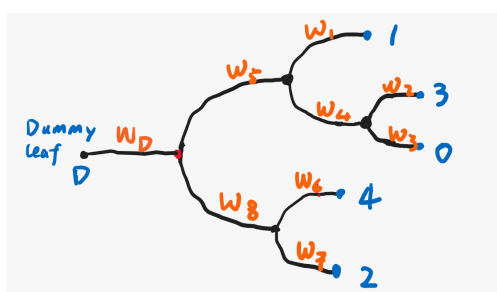


Figure 6: Illustration of a weighted rooted tree with dummy leaf.

Red: root indicated in Newick form; blue: leaf nodes; orange: weights of edges

By inserting a dummy leaf `D`, edge with w_5 represents the bipartition

$$(\{0,1,3\},\{2,4,D\})$$

while edge with w_8 represents bipartition

$$(\{0,1,3,D\},\{2,4\}).$$

Bitstring.

Now, **TreeScaper 2.0.0-alpha.2** can use bitstring which is a set of ordered boolean values to represent a bipartition. For those leaves in the same grouping, we labelled them with the same boolean value. Then, it remains to impose a order on the leaf set so that the boolean values can be aligned as a bitstring.

For the bipartition

$$(\{0,1,3\},\{2,4\}),$$

we have two representations `00101` and `11010` using the natural numerical order. If an dummy leaf is inserted, then the first bit is reserved for the dummy leaf. With the order of leaf set: $D,0,1,2,3,4$, the bipartition

$$(\{0,1,3\},\{2,4,D\}),$$

has representations `011010` and `100101`.

Note that leaves are usually not labelled with integers unless a Nexus form is provided. In this case, **TreeScaper** will generate a label map that convert leaf label to integer. Please refer to Sec.5.2.2 for details.

All bitstrings in a tree can be sequentially constructed in one reverse level order traversal, also referred as reverse breadth first traversal given the first traversal in reading the Newick string. Note that the Newick string is effectively a depth-first traversal. Given a depth-first traversal, **TreeScaper** implemented the reverse level order traversal with sequential computations happening at every edge in $O(n)$ complexity and $O(n)$ storage.

Leading bit.

Since computation perfer unique bipartition in address space, we choose to store the bitstring with leading `0`.

By assuming all bitstrings in memory have a same leading digit, we can relief from checking complementary bitstring when comparing two bitstrings. The reason of choosing `0` over `1` is a bit artifical. We may argue that the bitstring algorithm , mentioned later in this section, generates more bitstring with leading `0` but there is no actual numerical results support this claim.

Hash table.

Note that for L leaf set, every tree has at most $2L-3$ edges, maximum number of edges obtained at binary tree structure, while L of them are trivially attached to leaves, yet the representation space of $L-3$ nontrivial bipartitions is 2^{L-1}. Therefore, it is necessary to have hash-table structure for fast inquiries and comparison on bitstrings.

Instead of using a random hash id, **TreeScaper v2.0.0-alpha.2** use a deterministic hash id that can be sequentially computed during the computation of bitstring. The hash function is also h designed to be invariant under bit-wise inversion, e.g.,

$$h(\{00101\}) = h(\{11010\}).$$

If we represent the hash id with h_i, the unrooted tree in Fig.5 is uniquely represented by the set of bitstrings

$$\texttt{01111}, w_3, h_3 \quad \texttt{01000}, w_1, h_1 \quad \texttt{00100}, w_7, h_7 \quad \texttt{00010}, w_2, h_2$$
$$\texttt{00001}, w_6, h_6 \quad \texttt{01101}, w_4, h_4 \quad \texttt{00101}, w_5 = w_8, h_5$$

$$\texttt{011111}, w_D, h_D \quad \texttt{010000}, w_3, h_3 \quad \texttt{001000}, w_1, h_1 \quad \texttt{000100}, w_7, h_7 \quad \texttt{000010}, w_2, h_2$$
$$\texttt{000001}, w_6, h_6 \quad \texttt{010010}, w_4, h_4 \quad \texttt{011010}, w_5, h_5 \quad \texttt{000101}, w_8, h_8$$

while the tree with dummy leaf in Fig.6 is uniquely represented by

Due to the limitation of C++ that operates on byte in code level, users will have to choose container size from $\{8, 32, 64\}$ that stores the bitstring in array of {`unsigned char`, `unsigned int`, `unsigned long lon` Fillers will be added to the bitstring on addressive space. For the example we just used, suppose the container is set to be `unsigned char`, bitstring in addressive space will be `00101XXX` where `XXX` are fillers that could be either `0` or `1`.

It gets even more complicated if we look into the address space. Due to how C++ reads the byte location, we actually reverse the bitstring, which really only helps the input output code. For an 13-bitstring `1001101100011` stored in `unsigned char` containers, Tab. 2 gives a close look at address level.

	First byte	Second byte	
Abstract	`10011011`	`00011`	`XXX`
	Bitstring		Redundant
Address	`11011001`	`XXX`	`11000`
	Bitstring	Redundant	Bitstring

Table 2: Bitstring example

Finally, the explicit bitstrings are stored in an separate array to avoid repeated storage and the tree itself only keeps the hash ids and possibly the weights.

For N binary trees with L leaves that produces $M \leq (N+1)(L-3)$ unique bipartition using K-bits container for bitstring and let $C << M$ be the number collided hash-ids, the theoretical memory usage after processing is given below, (actual implementation take a bit more than this amount).

	Size(byte)	Description
Tree set	$4 \cdot (N(2L-2))$ $(+8 \cdot (N(2L-3)))$	$2L-3$ `int` for hash ids, 1 pointer to the tree itself and $(+2L-3)$ `double` for weight.
Hash table	$4 \cdot (2M + C)$	M tuples of `(tree id, bipartition id)` + extra collision id.
Bipartitions	$(K\lceil L/K \rceil)M$	Every bitstring needs $\lceil L/K \rceil$ containers and every container takes K bit.
Total	$(8 + K\lceil L/K \rceil)M + (8L-8)N + 4C(+(16L-24)N)$	

Table 3: Memory usage of tree data

If we are not storing the trivial bipartitions' id, then the term $(8L-8)N$ can be reduced to $(4L-8)N$ and $(16L-24)N$ in weighted tree can be reduced to $(8L-24)N$.

5.2.2 Taxon List Implementation

TreeScaper 2.0 keeps only one record for taxon informations: a size l linear array of `String` with taxon names. The linear array of taxon is either read from Nexus formatted file in block like with taxon {a, b, c}
1, a
2, b
3, c

or is created by labelling taxa with the order of presence in the first trees.

TreeScaper 2.0 has implemented an object `TaxonList` for storing the linear array as well as settings for later computations.

`TaxonList` must be created before reading trees. According to the size of taxon list, TreeScaper will then set the container size for bitstring. `set_bitstr_size(T ele)` will set the bit size as `8 * sizeof(T)`, the bitsize of `T`, it is 8 for `unsigned char`, 32 for `unsigned int`,
64 for `unsigned long long`.

For multiple trees input files, **TreeScaper 2.0** always creates `TaxonList` from the first file and alway maintain it with later input. Depending on whether the file is raw Newick form, i.e., taxon names explicitly presents in Newick string, or Nexus form with a taxon block, i.e., integer labels presents in Newick string, **TreeScaper 2.0** takes the following actions.

1. If the later file is raw Newick form, TreeScaper keeps using the `TaxonList` from the first file.

2. If the later file is Nexus form with conflict taxon labeling, TreeScaper will create additional mapping that convert the taxon labelling in the second file to the labels in `TaxonList`.

The duplicated taxon names in leaf nodes cause a huge waste on memory. A copy of taxon list could dominate the memory usage of a tree and **TreeScaper 1.1** has every tree keeping a copy of taxon list. The leaf node should be identified by a pointer to the linear array of taxon list, rather than the explicit string with taxon name.

5.2.3 Computation Tasks in Tree Module

Depth-First Traversal from Newick String.

```
Input: Newick formatted ASCII string s.
Output: Tree in linked list form that is ready for reverse level order traversal
Data: Array<Array<int>*> L, Array<Array<double>*> weights
Array<int> Parent_level, Parent_pos, unlabelled, active
i ← 0
j ← 0
L.push(new Array<int>)
while Scan string by index j do
    if s[j] == '(' then
        L[i]->push(-1)                                  // Unlabelled internal node
        L.push(new Array<int>)                                          // New level
        Parent_level.push(i)
        Parent_pos.push(L[i]->size() - 1)                // Record the parent info
        unlabelled.push(0)
        unlabelled[i]--                                 // Update the unlabel info
        i = L.size() - 1                                      // Go to child-level
        j++
    else if s[j] == ')' then
        i = Parent_level[i]                                  // Go to parent-level
        j++
    else if s[j] == ',' then
        j++
    else if s[j] == ':' then
        Read the followed weight and push it into weights[i]
        j increase accordingly
    else
        Read the followed leaf id n
        L[i]->push(n)
        active.push(i)                                // Record level id for leaves
        j increase accordingly
    end
end
return L, weights, parent_level, parent_pos, unlabelled, active
```

Algorithm 1: Depth First Traversal from Newick String

The DFT only performs one scan of the given ASCII string and for L leave size, it creates $(2L-3)\cdot 4 + A$ `int`, $2L-3$ `Array<int>*` and exrta $(2L-3)$ `weights` for weighted tree, where $L/2 \leq A \leq L-1$ is the number of levels that have leaf nodes.

Note that this algorithm handle general tree with possibly polytomy node. A more efficient implementation is possible for binary tree.

Reverse Level Order Traversal with Bitstring and Hash ID Computation.

We use the unweighted version for simplicity.

```
Input: Tree from Alg.1.
Output: Set of Hash IDs and updated Bitstring Array.
Data: Array<Bitstring> B
i= 0
L.push(new Array<int>)
Assign leaf node with bitstring corresponding bitstring b and hash id h
while active[i] != 0                                  // Not labelling the root level
 do
    if unlabelled[i] != 0                    // There is unlabelled node in this level
     then
        continue
    else
        l = active[i]
        for j in 0:L[i]->size() - 1 do
            Bitstring b[parent_level[l]][parent_pos[l]] &= b[l][j]      // Bitwise AND
            Hash ID h[parent_level[l]][parent_pos[l]] += b[l][j]
        end
        h[parent_level[l]][parent_pos[l]] /= bound     // bound isfor complementary
         invariance.
        Store bitstring and update hash tabel
    end
end
return Set of h
```

Algorithm 2: Reverse Level Order Traversal with Bitstring and Hash ID computation.

Note that every node carrys its ID, bitstring, hash value and possibly weights. Every swap in `active` scans only the level that has all node labelled and then label the parent nodes of those levels. The bitstring and hash computation is done during the scan of the levels. Overall, we only scan the tree in one traversal and the temperary dynamic arrays `L, weights, parent_level, parent_pos, unlabelled, act` will be reused in the next tree scan. Therefore, overall the tree set preprocessing only produce new bitstrings, $2L - 3$ hash id and weights and the tuples in hash table.

RF Distance Matrix Computation.

We use the unweighted version for simplicity.

Input: n Trees in form of sets of bipartitions
Output: Lower triangular part of distance matrix $D \in \mathbb{R}^{n \times n}$
Data: Sparse Bipartition-to-Tree matrix `B2T`

```
i= 0
j= 0
D = LowerTri<double>(n)
for i = 1:n do
    for k = Bipart. ID in Tree i do
        for j = 1:i do
            if B2T[k][j] == 0 then
                D[i][j] += 1
        end
    end
end
return D
```

Algorithm 3: Reverse Level Order Traversal with Bitstring and Hash ID computation.

Note that it is an inefficient algorithm since it take an outer product formula with very limited rank-1 update. A more efficient algorithm has been proposed.

Covariance Matrix Computation.

This is straight forward outer product of the sparse Bipartition-to-Tree matrix `B2T` and we will not include technical details of how we efficiently perform both row and column access to the column-major sparse matrix.

5.3 NLDR Module

This computation module is not yet rewritten for **TreeScaper v2.0.0-alpha.2**. Please refer to user manual of **TreeScaper v1.2.1** for implementation details.

NLDR module is an optimization library with various solvers implemented. In previous version, the routines involved with NLDR module are not organized well and the solver framework is very compact and deeply relies on a dense matrix input. In order to make the NLDR module a self-contained library that can handle flexible input structure while maintianing the efficiency of previous implementation, it is important to build a self-contained library for general optimization solver and partite previous code into different subroutines which adapts the new solver strcture. **TreeScaper v2.0.0-alpha.2** has implemented the optimization library.

5.3.1 Optimization Library

This is an optimization library operates on customized super type objects that is consist of a set of `void*` pointer converted from any data required in routines.

Super Type Arguments

Basic `KwArg`. The library use a self-implemented super type structure, named `KwArg`, to carry general arguments list for routines. The type `KwArg` stores a list of `void *` pointers and users are supposed to define their interpretations upon routine calls. The member function `arg(unsigned ind)` and the operator `void *operator()` are used to retrieve the pointer at index `ind` and `set_arg(T *x, ind)` is used to set the pointer at index `ind`.

It is strongly recommended to adopt the following general routine structure:

```
    return_type routine(KwArg &kws, unsigned int *ind)
```

in which the interpretations of pointers in kws at indices ind are explicitly given. Note that the `ind` array specifies the variables `routine` uses as well as their ordering. Users are expected to manage this ordering structure.

For example, let us define a behavior of computing $\frac{f(x)}{\text{sum}(x)}$.

```
    double eval(KwArg &kws, unsigned int *ind)
    {
        double *x = reinterpret_cast<double*>(kws.arg(ind[0]));
        unsigned int n = *reinterpret_cast<unsigned int*>(kws.arg(ind[1]));
        double f = *reinterpret_cast<double*>(kws.arg(ind[2]));\\
        double s = 0;
        for(auto i = 0; i < n; i++)
            s += x[i];
        return f / x;
    }
```

Then, for `kws` stores sequential pointers to: cost function `double (*cost)(double *)`, dimension `unsigned int* n`, variable `double *x` and cost value `double* f`. Use the following lines to call `eval`.

```
    unsigned int ind[3] = {2, 1, 3}; \\ {index of x, index of n, index of f}
    eval(kws, ind);
```

Wrapped `Optim_KwArg`. In the use of OptimLib, there are commonly used routines and data shared among different process. Therefore, a warpped super class `Optim_KwArg` is defined to create place holder for those default routines and data.

Please note that not all default pointers must be assigned with something, or corresponds to a legit routine, but it is users' responibility to avoid illegal memery access if there are undefined default pointers not being handled. Therefore, it is recommended to assign a pointer to some dummy variable or nullfunction provided in `OptimLib`:

```
    void nullfunction(Optim_KwArg &, unsigned int *){};
```

The wrapped super class contains the following members.

```
public:
    KwArg routine;
    KwArg arg;
    unsigned int CUS_IND;
    void *carg(unsigned int ind) { return this->arg(CUS_IND + ind);};
```

`routine` is used to stored pointers to default routine only, while `arg` stores pointers to default variables followed by pointers to customized variables, starting at index `CUS_IND`. `carg` is used to access the `ind`-th customized variable. Note that customized routines should also be kept in `KwArg arg`.

The large object in `OptimLib` usually have its own `Optim_KwArg` setting, see next part fore more details. To initialize to correct `Optim_KwArg`, use the `enum OPTIM_TYPE` as label in constructor.

```
    Optim_KwArg(OPTIM_TYPE ot, unsigned int cvn);
```

where `unsigned int cvn` is the number of customized variables.

Basic Objects and Predefined Dictionary for Routine Calls.

Here is the list of implemented optimization objects.

```
        class Optim_StepSize;
                class Optim_Const_StepSize : public Optim_StepSize;
                class Optim_Computed_StepSize : public Optim_StepSize;
                class Optim_Search_Stepsize : public Optim_StepSize;
```

kws index	Explicit pointer	Default Variable List/ Notion/ Indices reference	Notes
(R) 0	C	x, f $[4, 5]$	Compute cost value $f = f(x)$ at x.
(R) 1	D	x, Δ, n $[4, 6, 8]$	Compute update direction $\Delta^{\cdot}$
(R) 2	U	x, Δ, t, f, e $[4, 6, 9, 5, 2]$	Inplace update x along Δ with stepsize t.
(R) 3	SL	x, Δ, s, $[4, 6, 7]$	Compute the slope of f along Δ at x.
(A) 0	iter	i	Current iteration i
(A) 1	acc_time	T	Time used T
(A) 2	err	e	Current error e with respect to last point.
(A) 3	diff	δ	Change δ of x with respect to last point.
(A) 4		x	Variable x.
(A) 5		f	Cost value f of x.
(A) 6		Δ	Update direction Δ.
(A) 7		s	Slope of f along Δ.
(A) 8	nD	n	Norm of direction Δ
(A) 9	cur_step	t	Stepsize t.

Table 4: Default pointers of Optim_KwArg kws in Optim_Iter.

```
            class Wolfe_1st_StepSize : public Optim_Search_Stepsize;

    class Optim_Iter;

    class Optim_Paras;

    class Optim_Solver;
            class Optim_Update_Solver : public Optim_Solver;
            class Optim_Stepping_Solver : public Optim_Solver;
            class Optim_Trial_Stepping_Solver : public Optim_Stepping_Solver;
```

Certain objects only serve as an variable carrying static data, like Optim_Paras and Optim_Const_StepSize while others may call various routine provided super class Optim_KwArg &kws.

Only Optim_Iter, Optim_Search_Stepsize and Wolfe_1st_StepSize carry a native Optim_KwArg.

During the routine call, all data in kws are visable in forms of void* pointer. However, not all of them are needed in one routine call, for example, a routine that computes the current objective function value probably never use the update direction data. Therefore, the library also provide a dictionary that select the most used data in corresponding routine call in unsigned* ind.

The native Optim_KwArg routines, data and dictionaries are provided in Tab.

5.4 Community Detection Module

This computation module is not yet rewritten for **TreeScaper v2.0.0-alpha.2**. Please refer to user manual of **TreeScaper v1.2.1** for implementation details.

kws index	Explicit pointer	Default Variable List	Notes
(R) 0	S	x, Δ, t	Compute stepsize t at x along direction Δ.
(A) 0		x	Variable x
(A) 1		Δ	Direction Δ
(A) 2		t	Stepsize t
(A) 3		n	Norm of Δ.

Table 5: Default pointers of Optim_KwArg kws in Optim_Search_Stepsize.

kws index	Explicit pointer	Default Variable List/ Notion	Notes
(R) 0	C	x, f $[6, 7]$	Compute cost value $f = f(x)$ at x.
(R) 1	Update	x, Δ, t, y, f, f_y, e $[1, 3, 5, 6, 2, 7, 0]$	General Update x stored at y.
(R) 2	CP	x, y $[1, 6]$	Copy x to y.
(A) 0		e	Current Error e
(A) 1		x	Variable x
(A) 2		f	Cost value f
(A) 3		Δ	Update direction Δ
(A) 4		s	Slope s along Δ
(A) 5		t	Stepsize t
(A) 6		y	Temporary variable y
(A) 7		f_y	Cost value $f(y)$ of temporary y

Table 6: Default pointers of Optim_KwArg kws in Wolfe_1st_StepSize.

References

Amenta, N. and Klingner, J. (2002). Case study: visualizing sets of evolutionary trees. In *IEEE Symposium on Information Visualization, 2002. INFOVIS 2002.*, pages 71–74.

Castoe, T. A., de Koning, A. P. J., Kim, H.-M., Gu, W., Noonan, B. P., Naylor, G., Jiang, Z. J., Parkinson, C. L., and Pollock, D. D. (2009). Evidence for an ancient adaptive episode of convergent molecular evolution. *Proceedings of the National Academy of Sciences*, 106(22):8986–8991.

Gori, K., Suchan, T., Alvarez, N., Goldman, N., and Dessimoz, C. (2016). Clustering genes of common evolutionary history. *Molecular Biology and Evolution*, 33(6):1590.

Hillis, D. M., Heath, T. A., John, K. S., and Anderson, F. (2005). Analysis and visualization of tree space. *Systematic Biology*, 54(3):471.

Lewitus, E. and Morlon, H. (2016). Characterizing and comparing phylogenies from their laplacian spectrum. *Systematic Biology*, 65(3):495.

Stockham, C., Wang, L.-S., and Warnow, T. (2002). Statistically based postprocessing of phylogenetic analysis by clustering. *Bioinformatics*, 18(suppl_1):S285.

Wilgenbusch, J. C., Huang, W., and Gallivan, K. A. (2017). Visualizing phylogenetic tree landscapes. *BMC Bioinformatics*, 18(1):85.