

TreeScaper CLV Manual

Beta Version

August 27, 2020

Jeremy Ash^{2,4},
Jeremy M. Brown²,
David Morris²,
Guifang Zhou²,
Wen Huang¹,
Melissa Marchand³,
Paul Van Dooren¹,
Kyle A. Gallivan³,
and Jim Wilgenbusch⁵

¹Department 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	2
2	QUICKSTART TUTORIAL with CLVTreeScaper	5
2.1	Getting Started	5
2.1.1	Installation	5
2.1.2	Example Tree Set	5
2.2	Basic Computations	5
2.3	Visualizing Tree Space	6
3	INSTALLATION	7
4	COMMANDS	9
A	UTILITY DETAILS	15
A.1	Computing bipartitions and their distribution	15
A.2	Computing tree distances	16
A.2.1	Robinson Foulds distance	16
A.2.2	Matching distance	17
A.2.3	subtree-Prube-and-Regraft distance	17
A.3	Computing NLDR of distance matrix	17
A.3.1	Classic multidimensional scaling(MDS)	18
A.3.2	Linear iteration	18
A.3.3	Gauss-Seidel	19
A.3.4	Stochastic	19
A.3.5	Metropolis	20
A.3.6	Majorization	20
A.3.7	Stress functions	21
A.4	Processing adjacency matrix	21
A.5	Performing community detection	22
A.5.1	No Null Model	22
A.5.2	Erdős-Rényi Model	22
A.5.3	Configuration Null Model	23
A.5.4	Constant Potts Model	23
	References	25

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,

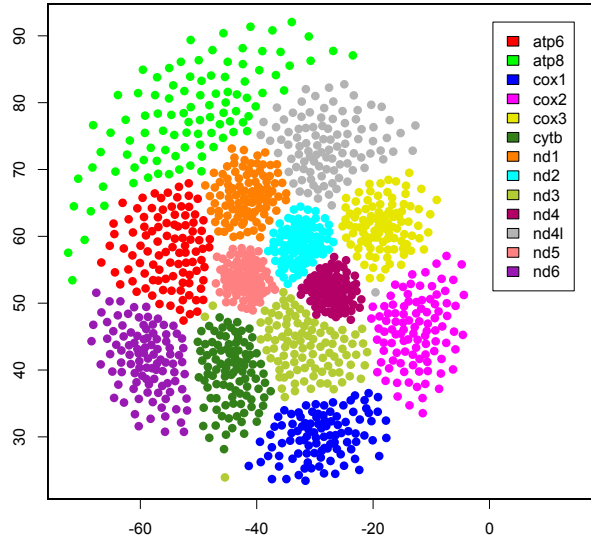


Figure 1: A 2–dimensional representation of treespace generated by non-linear dimensionality reduction (NLDR) for a set of 1,300 trees sampled from individual Bayesian analyses of 13 mitochondrial protein-coding genes in squamates [Castoe et al., 2009]. Each point represents a tree sampled from the posterior 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-Rnyi Null Model (ERNM) and No Null Model(NNM)]. The general idea behind community detection performed on adjacency matrix $\text{Adj} \in \mathbb{R}^{n \times n}$ is to find a sparse adjacency matrix $\widetilde{\text{Adj}} \in \mathbb{R}^{n \times n}$ and permutation matrix $P \in \mathbb{R}^{n \times n}$ that best preserves the true graph as well as obtaining $P\widetilde{\text{Adj}}P^T$ 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).

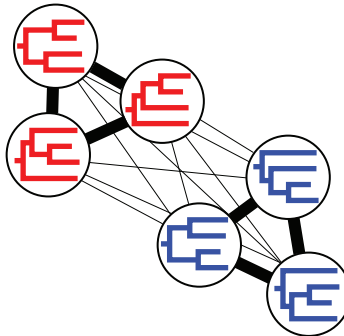


Figure 2: A cartoon topological affinity network. Circles are nodes in the network, each of which corresponds to one tree from a tree set. Edges represent the affinity (or similarity) between the trees, with thicker lines indicating greater affinity. Tree colors correspond to one intuitive definition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

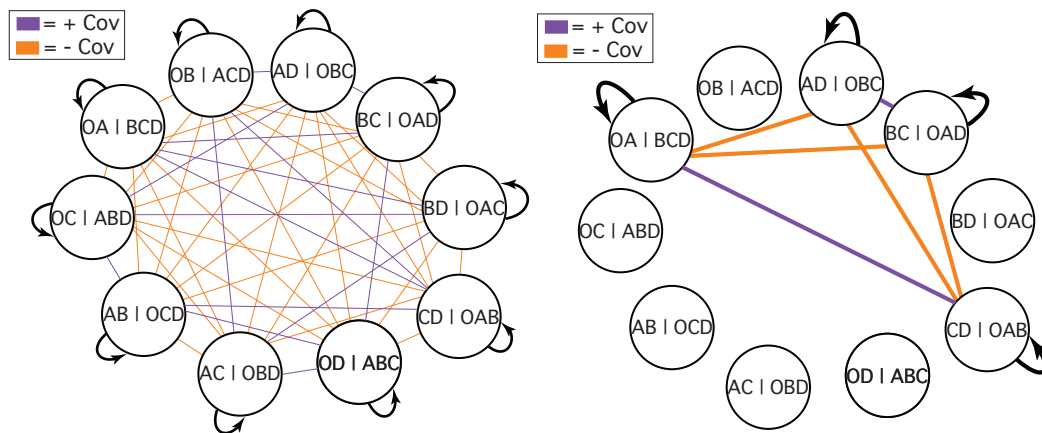


Figure 3: Two example bipartition covariance networks for sets of unrooted, 5-taxon trees. The network on the left corresponds to a tree set with a uniform distribution of frequencies across all possible tree topologies. Some weak covariances exist in this network, because some pairs of bipartitions 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 available 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 sub-folder `/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 Euclidean 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 -nlldr -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-space.

The output file `Distance.NLDR.test.out` is the Euclidean distance matrix generated by X .

3 INSTALLATION

Source code of TreeScaper command-line version for Mac or Linux can be downloaded from

<https://github.com/TreeScaper/TreeScaper>

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:

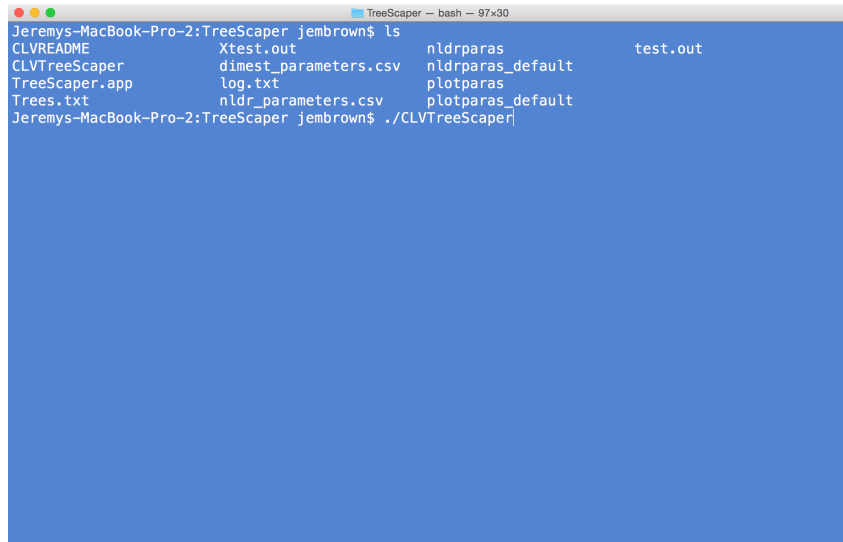
<code>/CLVTreeScaper</code>	the <code>CLVTreeScaper</code> binary
<code>/nldr_parameters.csv</code>	parameters for <code>nldr</code> routines
<code>/dimest_parameters.csv</code>	parameters for dimension estimation routines

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

A terminal window titled "TreeScaper -- bash -- 97x30" showing the output of the 'ls' command in the directory "Jeremys-MacBook-Pro-2:TreeScaper jembrown\$". The output lists files and subdirectories: CLVREADME, CLVTreeScaper, TreeScaper.app, Trees.txt, Xtest.out, dimest_parameters.csv, log.txt, nldr_parameters.csv, nldrparas, nldrparas_default, plotparas, plotparas_default, and test.out. The prompt then changes to "Jeremys-MacBook-Pro-2:TreeScaper jembrown\$./CLVTreeScaper".

```
Jeremys-MacBook-Pro-2:TreeScaper jembrown$ ls
CLVREADME      Xtest.out      nldrparas      test.out
CLVTreeScaper  dimest_parameters.csv  nldrparas_default
TreeScaper.app log.txt         plotparas
Trees.txt      nldr_parameters.csv  plotparas_default
Jeremys-MacBook-Pro-2:TreeScaper jembrown$ ./CLVTreeScaper
```

There are three general run modes for the command-line version of TreeScaper ("CLVTreeScaper"). Using one of these five flags as the first command-line argument sets the mode (e.g., CLVTreeScaper -trees). Note that there are legacy code from previous version that are still functional but no longer suggested to be use any more. Commands and options related to those code are therefore omitted here. For more information on previous version, please refer to [TreeScaperManual_v2.pdf](#).

The current command structure is shown in Fig.4, while the previous structure will remain in the TreeScaper binary.

New key arguments `-comm` and `-aff` are added to TreeScaper. The new structure is centered at files with header information (in order to simplify file names and shorten argument list).

`-aff` and `-comm` which stand for affinity and community detection are separated from `-tree`, which now only takes care of computing bipartition matrix, distance matrix and covariance matrix from tree file. Users are now able to perform any modification on output files from `-tree`, for example, send them to `-nldr`. Then `-comm` will take these (modified) files and other necessary information to perform CD methods. The details of these keys are given in below.

(1) -trees

In this mode, users can compute a majority rule/strict consensus tree, distance matrix, bipartition 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:

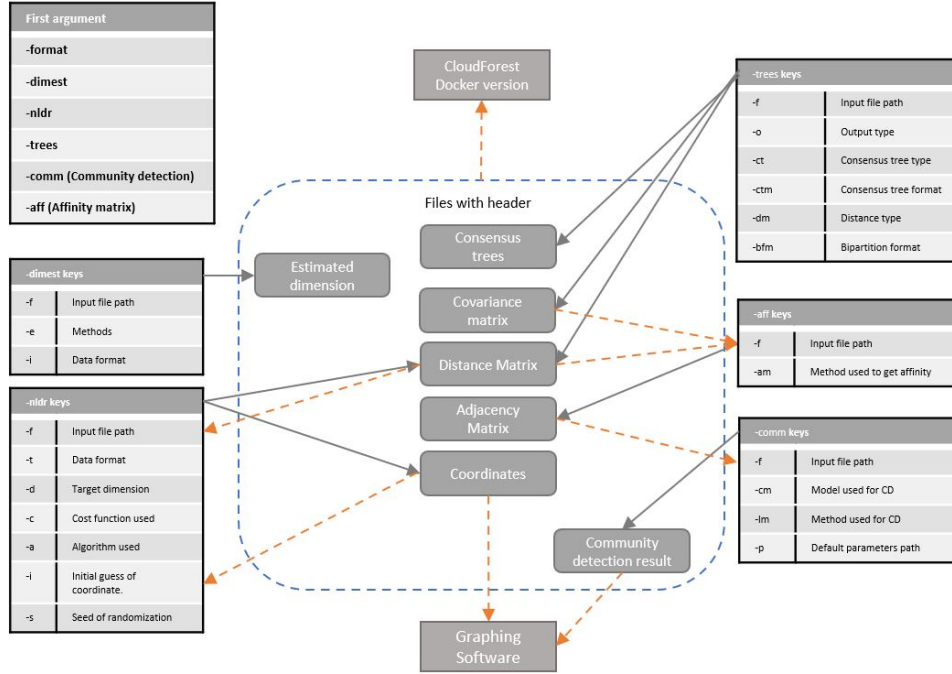


Figure 4: Command List and routine structure based on file.

- '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

'Mat': Matching distance

'SPR': Subtree-Prune-Regraft

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}
```

(2) -nlldr

In this mode, users can project distance matrix into lower dimensional space using non-linear dimensionality reduction (NLDR). Relevant arguments include (default option are given in red):

-f: Name of the file containing distance data.

-t: The type of distances contained in the file. Options are:

'DIS': A lower triangle matrix of original distances

'COR': Low-dimensional Euclidean coordinates (if already computed).

- d: The desired dimension of the Euclidean representation (usually 1, 2, or 3).
- c: The chosen cost function. Options are:
 - 'CLASSIC_MDS'
 - 'KRUSKAL1'
 - 'NORMALIZED'
 - 'SAMMON'
 - 'CCA'
- a: The chosen NLDR algorithm. Options are:
 - 'LINEAR_ITERATION'
 - 'MAJORIZATION'
 - 'GAUSS_SEIDEL'
 - 'STOCHASTIC'
- i: The method for generating initial Euclidean coordinates. Options are:
 - 'RAND': Randomly choose coordinates for each point.
 - 'CLASSIC_MDS': Generate initial coordinates using classic multi-dimensional scaling (MDS).
- s: A random seed, if initial coordinates are generated randomly.
- post: Suffix name of file. Options are:
 - 'none'
 - 'time'
 - 'AnyOtherString': filename will be attached with `_AnyOtherString`

Example command:

```
./CLVTreeScaper -nlldr -f {test.out} -t {DIS,COR} -d {AnyPositiveInteger}
-c {CLASSIC_MDS,KRUSKAL1,NORMALIZED,SAMMON,CCA}
-a {LINEAR_ITERATION,MAJORIZATION,GAUSS_SEIDEL,STOCHASTIC}
-i {RAND,CLASSIC_MDS} -s 1 -post {none, time, AnyString}
```

(3) -aff

In this mode, users can process (pre-)adjacency matrix. Relevant arguments include (default option is colored in red):

- f: Provide the name of the file that contains the data
- am: Indicates the distance to affinity transformation. Options are:
 - 'Rec': Reciprocal
 - 'Exp': Exponential
- post: Suffix name of file. Options are:
 - 'none'
 - 'time'
 - 'AnyOtherString': filename will be attached with `_AnyOtherString`

(4) `-comm`

In this mode, user can perform community detection method on adjacency matrix. Relevant arguments include (default option is given in red):

-cm: Model used to compute communities. Options are:

- 'CNM':** Configuration Null Model
- 'CPM':** Constant Potts Model
- 'ERNM':** Erdos-Renyi Null Model
- 'NNM':** No Null Model

-lm: Method of plateau detection. Options are:

- 'auto':** automatically choose lambdas and find plateaus
- 'manu':** specify intervals by users to find plateaus

-post: Suffix name of file. Options are:

- 'none'**
- 'time'**
- 'AnyOtherString':** filename will be attached with `_AnyOtherString`

The following flags are used to specify values of lambda for manual searches:

- lp:** Specify a fixed value of λ^+ . Must be between 0 and 1. Used when `-lpiv` is zero (see below).
- lps, -lpe, -lpiv:** Starting, ending, and sampling intervals for λ^+ . Used to explore a range of possible values for λ^+ .
- ln:** Specify a fixed value of λ^- . Must be between 0 and 1. Used when `-lniv` is zero (see below).
- lns, -lne, -lniv:** Starting, ending, and sampling intervals for λ^- . Used to explore a range of possible values for λ^- .

Note: Either λ^+ or λ^- must be fixed, because plateau detection is undefined when both vary.

- hf:** Frequency upper bound. A number between 0 and 1. Nodes with frequencies above this value are ignored.
- lf:** Frequency lower bound. A number between 0 and 1. Nodes with frequencies below this value are ignored.

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 Communities with λ^+ Fixed:

```
./CLVTreeScaper -comm -f {Adjacency.out} -cm {CNM, CPM, ERNM, NNM}
```

```
-lm manu -lp {AnyNumber} -lns {AnyNumber} -lne {AnyNumber}
-lniv {AnyNumber} -hf {0-1} -lf {0-1} -post {none, time, AnyString}
```

Compute Communities with λ^- Fixed:

```
./CLVTreeScaper -comm -f {Adjacency.out} -cm {CNM,CPM,ERNM,NNM}
-lm manu -ln {AnyNumber} -lps {AnyNumber} -lpe {AnyNumber}
-lpiv {AnyNumber} -hf {0-1} -lf {0-1} -post {none, time, AnyString}
```

Compute Communities with Automatically Chosen Lambdas:

```
./CLVTreeScaper -comm -f {Adjacency.out} -cm {CNM/CPM/ERNM/NNM}
-lm auto -hf {0-1} -lf {0-1} -post {none, time, AnyString}
```

(5) -dimest

In this mode, users can estimate the intrinsic dimensionality of their data. This estimate can help in deciding on an appropriate number of dimensions to use when performing NLDR projections.

-f: Name of the file containing distance data.

-i: The type of distances contained in the file. Options are:

'DIS': A lower triangle matrix of original distances

'COR': Low-dimensional Euclidean coordinates (if already computed).

-e: The chosen estimator. Options are:

'CORR_DIM': Correlation dimension estimator

'NN_DIM': Nearest neighbor estimator

'MLE_DIM': Maximum likelihood estimator

Example command:

```
./CLVTreeScaper -dimest -f {test.out} -i {DIS,COR}
-e {CORR_DIM,NN_DIM,MLE_DIM}
```

A UTILITY DETAILS

The command-line version of TreeScaper (CLVTreeScaper) has three independent major routines implemented, computing tree-related objects, computing NLDR and performing community detection. A comprehensive task, e.g. visualizing trees grouped by their distance, will need a sequence calls to CLVTreeScaper for activating different routines. These routines communicate with each other and carry out essential information through the output-input files.

Input-Output files:

Command key	Input	Output
-trees	Nexus formatted trees Newick formatted trees	Bipartition list Bipartition covariance K Distance matrix of trees $D \in \mathbb{R}^{n \times n}$ Consensus trees
-nlldr	Distance matrix $D \in \mathbb{R}^{n \times n}$ Coordinates matrix $X \in \mathbb{R}^{n \times m}$ from \mathbb{R}^m	Distance matrix \bar{D} from Euclidean \mathbb{R}^k Coordinates matrix $\bar{X} \in \mathbb{R}^{n \times k}$ from \mathbb{R}^k Information of NLDR quality
-aff ¹	(pre)-Adjacency matrix $\text{Adj} \in \mathbb{R}^{n \times n}$	Adjacency matrix $\widehat{\text{Adj}} \in \mathbb{R}^{n \times n}$
-comm	Adjacency matrix $\text{Adj} \in \mathbb{R}^{n \times n}$	Modularity and grouped nodes' ID

Particular information can be carried out through different calls of CLVTreeScaper by the header information included in files. Note that -trees command does not yet support header information in tree file due to some conflict of formatting.

An header information is a collection of messages surrounded by angle bracket $< >$. Each line inside the bracket separated by $:$ is a message in forms of `item:content`. Some message, about labeling created time or output format, will be updated and the rest will be carried out in the output file generated by CLVTreeScaper.

Bipartitions and related objects are the basis objects for computing further objects. Given a set of trees with exactly n leaves, CLVTreeScaper generates a unique ID for each possible bipartition as n bitstring. See Sul and Williams [2011] about details of the translation.

A.1 Computing bipartitions and their distribution

When CLVTreeScaper -trees is called, it will read trees from file and store all of them in memory.²

Then it scan through the tree set and record every bipartition presented in the tree set. For this tasks, CLVTreeScaper first assigns hash values to each leaf and then performs the following

²In the future, we can develop functionality that allow to store limited trees in case the tree set is to big to store.

subroutine so that

Input: Tree with leaves $\{l_i\}, i = 1, \dots, n$
Output: Critical point p_* of f
Data: Hash values of each leaf $\{h_i^1\}, \{h_i^2\}, i = 1, \dots, n$

```

1  $i = 0$ 
2 while Scanning with in-order traversal do
3   if The current node is  $l_i$  then
4     Assign hash value  $h_i^1, h_i^2$  to the node
5     Generate bitstring " $\delta_{1i} \dots \delta_{ni}$ " and assigned to the node
6   end
7   else
8     Assign the sum of hash values accordingly of its children to the node
9     Generate bitstring by bit-wise OR operation to its children's bitstring and assigned
      to the node
10  end
11 end
12 Label the bitstring of the root by its two hash values and stored it in hash table

```

Algorithm 1: Compute hash value of bipartition

After performing Alg. 1, the root has hash value that label the corresponding to the bitstring for its bipartition structure.

Once every tree has been processed, CLVTreeScaper will scan through tree set again but only access every trees root's hash value. By storing and sorting hash values of the root, CLVTreeScaper is then able to count how many times bipartitioned are appeared in this tree set (via looking through hash table).

Complexity concern: sorting is not necessary especially when number of bipartition is far smaller than the number of trees.

Data storage concern: The structure of hash table is poorly managed. It is a 3— dimensional array by construction, i.e., uses 3 labels, yet only two indices are labeled on the data, bipartitions labeled by bitstring and trees labeled by their ID. It costs unnecessary and complicated code as well as inconsistent coding when the extra label is flatten for the computation of weighted RF-distance.

A.2 Computing tree distances

CLVTreeScaper supports computation of 3 kinds of tree distance: Robinson Foulds distance³, (URF) for unweighted tree and (RF) for weighted tree, Matching distance (Mat) and subtree-Prune-and-Regraft (SPR) distance. CLVTreeScaper implements RF-distance and Matching distance with its original code and use the SPR-tree library developed by for the computation of SPR-distance.

A.2.1 Robinson Foulds distance

For unweighted trees t_i and t_j , let A_i , and A_j , be the number of all bipartitions that the structure of t_i , and t_j accordingly, can obtain. Let B_{ij} be the number of bipartition appeared in both t_i and t_j . Then the RF-distance is given by

$$d^{\text{URF}}(t_i, t_j) = \frac{A_i + A_j - 2B_{ij}}{2}. \quad (1)$$

³wiki page of Robinson Foulds distance

For weighted trees t_i and t_j , let $\{b_k\}$ be all bipartitions appeared in at least one of them. Let $w_i(b_k)$ and $w_j(b_k)$ be the weight of root which correspond to an instance of bipartition b_k in tree t_i and t_j . (If there is no instance of b_k , then it is zero.) Then the RF distance is given by

$$d^{\text{RF}}(t_i, t_j) = \frac{\sum_{b_k} |w_i(b_k) - w_j(b_k)| + |w_j(b_k) - w_i(b_k)|}{4} \quad (2)$$

Algorithmic concern: why not compute the equivalent $\sum_{b_k} |w_i(b_k) - w_j(b_k)|/2$ instead?

The computation is done on single precision floating point system.

A.2.2 Matching distance

In order to compute matching distance, an combinatorial optimization problem will be formed for each pair of trees (t_i, t_j) first, then Hungarian algorithm⁴ implemented in Stachniss [2004] is called. The matching distance is the solution to the optimization problem.

The cost matrix C is describing the bitwise distance between bipartition b_k and b_l that appeared in at least one of the two trees. Let Δ_{kl} be the number of different bits in bitstring of b_k and bitstring of b_l (for bipartition never appear in a tree, make the bitstring all 1). C_{kl} is given by

$$C_{kl} = \min(\Delta_{kl}, N_{\text{taxa}} - \Delta_{kl}) \quad (3)$$

where N_{taxa} is the number of taxa. Then distance is given as

$$d_{ij}^{\text{Mat}} = \min_{P \text{ is Permutation}} \text{Tr}(PCP^T). \quad (4)$$

Note that the problem scale M is the number of all possible bipartitions and complexity of Hungarian algorithm is at $O(M^3)$. There are $n(n-1)/2$ distance needed to be computed, leading to overall complexity $O(n^2M^3)$

Algorithmic Error: In handling the case where b_k is not in tree t_i or b_l is not in tree t_j , the code calling `onebitstrXOR` is not comparing to 1 bitstring.

A.2.3 subtree-Prune-and-Regraft distance

TreeScaper implements routine to translate the tree structure it created to SPR tree structure implemented in Chris Whidden's library⁵. In addition, CLVTreeScaper calls the routines built-in that library for computing SPR distance. Details of the algorithm implemented in Whidden's library can be found in Whidden and Matsen [2018].

Algorithmic concern: TreeScaper may implement SPD distance computation itself in the future.

A.3 Computing NLDR of distance matrix

For the task of NLDR, there are 5 iterative methods implemented for nontrivial stress function and 1 deterministic method implemented for MDS cost function.

Data storage concern: All symmetric matrix are stored in dense form.

Complexity concern: The update is on the coordinate matrix $X \in \mathbb{R}^{n \times k}$ at the scale of $O(nk)$ but the stress function evaluated at $D \in \mathbb{R}^{n \times n}$ at the scale of $O(n^2)$, which implies forming D_i with complexity $O(nk^2)$ and evaluating stress function $O(n^3)$ is necessary at each step. Is there a way to work directly on X , which get rid of $O(nk^2)$ of forming D and get less computation $O(nk^2)$ for evaluating cost function.

⁴wiki of Hungarian algorithm

⁵The library can be found in his homepage

A.3.1 Classic multidimensional scaling(MDS)

This method, also known as standard scaling method⁶, only applies to Euclidean distance matrix. However, this method also used as a lousy solution for initial guess of other problem.

For given distance matrix $D \in \mathbb{R}^{n \times n}$ or coordinates matrix $C \in \mathbb{R}^{n \times m}$ from \mathbb{R}^1 , compute the squared proximity matrix $D^{(2)} = [d_{ij}^2]$ and CLVTreeScaper performs the following algorithm.

Input: Squared proximity matrix $D^{(2)}$, targeted dimension k

Output: Coordinate matrix $X \in \mathbb{R}^{n \times k}$

- 1 Perform singular value decomposition of $D^{(2)} = U\Sigma V^T$
- 2 Selete k -th positive largest eigenvalue λ_i and their eigenvectors U_i
- 3 Form coordinate matrix by lining up scaled eigenvectors $\begin{bmatrix} \lambda_1^{1/2}U_1 & \cdots & \lambda_k^{1/2}U_k \end{bmatrix}$

Algorithm 2: Multidimensional Scaling

Note that negative eigenvalue only appear when distance matrix is not Eculidean.

Algorithmic concern: It perform SVD on dense symmetric matrix, instead of eigenvalue decomposition.

A.3.2 Linear iteration

Linear iteration method is straight-forward. For different cost functions, there are subroutines built in CLVTreeScaper to obtain a descent direction (usually the negative gradient) and then CLVTreeScaper performs the following algorithm

Input: Initial guess $X_0 \in \mathbb{R}^{n \times k}$

Output: Coordinate matrix $X \in \mathbb{R}^{n \times k}$

Data: Subroutine f for computing descent direction, step size λ

- 1 $i \leftarrow 0$
- 2 **while** *not converged* **do**
- 3 $V_i \leftarrow f(X_i)$
- 4 $X_{i+1} \leftarrow X_i + \lambda V_i$
- 5 $i \leftarrow i + 1$
- 6 **end**

Algorithm 3: Linear Iteration

Algorithmic concern: Linear iteration uses fixed stepsize. There are better stepsizes, e.g., Armijo stepsize.

⁶wiki of multidimensional scaling.

A.3.3 Gauss-Seidel

Gauss-Seidel methods is similar to linear iteration, except that it perform changes on entry-wise basis of the coordinate matrix X_i .

Input: Initial guess $X_0 \in \mathbb{R}^{n \times k}$
Output: Coordinate matrix $X_* \in \mathbb{R}^{n \times k}$
Data: Subroutine f for computing descent direction, step size λ

```

1  $l \leftarrow 0$ 
2 while not converged do
3    $X_{l+1} \leftarrow X_l$  (No computation exists in here)
4    $i \leftarrow 1$ 
5    $j \leftarrow 1$ 
6   for  $i \leq n$  do
7     for  $j \leq k$  do
8       Compute descent direction  $V_{l,ij}$  along  $X_{l,ij}$ 
9       Find Armijo stepsize  $\lambda_{l,ij}$ 
10       $X_{l+1,ij} \leftarrow X_{l+1,ij} + \lambda_{l,ij} V_{l,ij}$ 
11       $j \leftarrow j + 1$ 
12    end
13     $i \leftarrow i + 1$ 
14  end
15   $l \leftarrow l + 1$ 
16 end

```

Algorithm 4: Gauss-Seidel

A.3.4 Stochastic

Stochastic methods refers to the stochastic descent method⁷, which selects random set of coordinates $X_{i_l j_l}$ and compute descent direction along them. The ratio between number of selected indices and number of all indices is control by parameter epochs. The stepsize is control by parameter

⁷wiki of stochastic descent method

STO_alphan.

<p>Input: Initial guess $X_0 \in \mathbb{R}^{n \times k}$</p> <p>Output: Coordinate matrix $X_* \in \mathbb{R}^{n \times k}$</p> <p>Data: Subroutine f for computing descent direction, stepsize $\{\lambda_m\}_{m=1}^M$</p> <pre> 1 $l \leftarrow 0$ 2 while <i>not converged</i> do 3 $X_{l+1} \leftarrow X_l$ (No computation exists in here) 4 Sample $\{i_m, j_m\}_{m=1}^M$ 5 $m \leftarrow 0$ 6 for $m \leq M$ do 7 Compute descent direction $V_{l,m}$ along selected indices $\{i_m, j_m\}$ 8 $X_{l+1} \leftarrow X_{l+1} + \lambda_m V_{l,m}$ 9 $l \leftarrow l + 1$ 10 $m \leftarrow m + 1$ 11 end 12 end </pre>

Algorithm 5: Stochastic

Algorithmic concern: fixed stepsize is used.

A.3.5 Metropolis

This algorithm refers to Metropolis-Hastings algorithm⁸, which samples descent vector V around the gradient descent direction, accepts/rejects it by the decrease in f obtained along V .

<p>Input: Initial guess $X_0 \in \mathbb{R}^{n \times k}$</p> <p>Output: Coordinate matrix $X_* \in \mathbb{R}^{n \times k}$</p> <p>Data: Subroutine f for computing gradient descent direction</p> <pre> 1 $i \leftarrow 0$ 2 $j \leftarrow 0$ 3 while X_j <i>not converged</i> do 4 Compute $\nabla_j = -\text{grad}f(X_j)$ 5 Compute stepsize $\lambda_j = 1/\ \nabla_j\ _F^2$ 6 while <i>Not enough decrease in f at $X_{j,i}$ and $i < 2000$</i> do 7 Sample V_i around ∇_j 8 $X_{j,i} \leftarrow \lambda_j V_i$ 9 $i \leftarrow i + 1$ 10 end 11 $X_{j+1} \leftarrow X_{j,i}$ 12 $j \leftarrow j + 1$ 13 end </pre>
--

Algorithm 6: Metropolis

A.3.6 Majorization

This algorithm refers to Majorize-Minimization algorithm⁹, which construct a surrogate function represented by $B_i \in \mathbb{R}^{n \times n}$ from $D_i = \text{Dist}(X_i)$ the distance matrix of coordinates X_i . The optimal

⁸wiki of Metropolis-Hastings algorithm

⁹wiki of MM algorithm.

solution for B_i is known and given by f .

Input: Initial guess $X_0 \in \mathbb{R}^{n \times k}$
Output: Coordinate matrix $X_* \in \mathbb{R}^{n \times k}$
Data: Subroutine f for forming related problem B_i , g for solving of related problem

```

1  $i \leftarrow 0$ 
2 while not converged do
3   Compute  $B_i = f(D_i)$ 
4   Compute  $V_i = g(B_i)$ 
5   Update  $X_{i+1} \leftarrow X_i + V_i$ 
6   Update  $D_{i+1}$ 
7    $i \leftarrow i + 1$ 
8 end
```

Algorithm 7: Majorization

Algorithmic concern: D_i is never updated for NORMALIZED cost function. It seems never converge.

A.3.7 Stress functions

TreeScaper implement 4 kinds of stress functions evaluated at distance matrix D . NLDR is done by applying specific algorithms above to minimize these cost functions. Under the sense of these stress functions, the resulted coordinates $X_* \in \mathbb{R}^{n \times k}$ is said to be the best approximation of the points that generate the original distance matrix. The stress functions are

1. MDS stress function, that has closed form formula of X_* given in Alg. 2 (when the original distance matrix is Euclidean).
2. KRUSKAL1 stress function, that accepts all 5 iterative optimization algorithms.
3. NORMALIZED stress function, that accepts 4 iterative optimization algorithms except for Linear-Iteration method.
4. SAMMON stress function, that accepts 4 iterative optimization algorithms except for Linear-Iteration method.
5. CCA stress function, that accepts 4 iterative optimization algorithms except for Linear-Iteration method.

A.4 Processing adjacency matrix

The key `-aff` is now only used for converting distance matrix $D = [d_{ij}] \in \mathbb{R}^{n \times n}$ to an affinity matrix considered as an adjacency matrix $\text{Adj} = [a_{ij}] \in \mathbb{R}^{n \times n}$ of a graph. In the future, this command will be expanded for the purposes of preprocessing (pre-)adjacency matrix. For example, it can set threshold of the weights and zero out all edge weight with magnitude under that threshold.

For converting distance matrix to adjacency matrix, the larger distance between node i and node j , the weaker they are considered connected. Therefore, CLVTreeScaper performs the following two convert functions.

1. `-Exp`:

$$a_{ij} = \exp(-d_{ij}).$$

2. -Rec:

$$a_{ij} = \frac{1}{d_{ij} + 0.1 \max_{k,l} d_{kl}}.$$

Data storage concern: All symmetric matrix are stored in dense form.

A.5 Performing community detection

Community detection includes a broad class of methods that attempt to find structure in networks, by identifying groups of nodes that are more densely or tightly connected to each other than they are to other nodes in the network [Newman, 2010]. These methods do not require the number and size of groups (known as communities) to be identified in advance, in contrast to graph partitioning approaches. Each of the community detection methods implemented in TreeScaper employ a quantity known as the Hamiltonian (\mathcal{H}). Roughly analogous to the use of this term in quantum mechanics, the Hamiltonian represents the energy imposed by a given community structure. The structure with the minimum energy represents the most natural division of nodes. Below we provide definitions of \mathcal{H} for each of the methods in TreeScaper, as well as some explanation of how these definitions influence the detected communities, in order to help users efficiently explore parameter space and properly interpret model output. We also direct those interested to more in-depth explanations of these methods in papers by Fortunato, 2010; Reichardt and Bornholdt, 2006; Raghavan et al., 2007; Traag and Bruggeman, 2009; Traag et al., 2011; Traag, 2014.

A.5.1 No Null Model

In the first case, which we term No Null Model (NNM, also known as the label propagation method), \mathcal{H} is defined for the set of all communities, $\{\sigma\}$, and is given by

$$\mathcal{H}(\{\sigma\}) = - \sum_{i,j} A_{i,j} \delta(\sigma_i, \sigma_j), \quad (5)$$

where the sum is over all nodes i and j , $A_{i,j}$ is the adjacency between nodes i and j (i.e., is there exists an edge connecting nodes i and j), and σ_i is the community to which bipartition i belongs. $\delta(\sigma_i, \sigma_j)$ is defined as 1 when i and j are in the same community, and 0 otherwise. The NNM contains no tunable parameters and is generally of the least interest, since its Hamiltonian has only one global optimum with all nodes in a single community. However, local optima could be of some interest.

One way to refine our approach to community detection beyond the NNM involves first defining an expectation for structure based on a stochastic model of network construction. In these cases, the existence of communities can be revealed by comparison between the actual density of edges in a subgraph and the density one would expect to have in the null subgraph without community structure. The expected edge density depends on the chosen null model. The two following methods are based on different choices of null model.

A.5.2 Erdős-Rényi Model

For the Erdős-Rényi Model (ERM), \mathcal{H} is given by

$$\mathcal{H}(\{\sigma\}) = - \sum_{i,j} \left[A_{i,j} - \left(p_{i,j}^+ \lambda^+ - p_{i,j}^- \lambda^- \right) \right] \delta(\sigma_i, \sigma_j), \quad (6)$$

where $p_{i,j}$ is the probability of a positive ($p_{i,j}^+$) or negative ($p_{i,j}^-$) edge between i and j in the null case, while λ^+ and λ^- are tunable parameters. All other definitions are as Equation (5) for the NNM. For Erdős-Rényi's random graph model, the probability of occurrence for any particular positive edge ($p_{i,j}^+$) is m^+/n^2 , while the corresponding probability for any particular negative edge ($p_{i,j}^-$) is m^-/n^2 , where m^+ is the sum of positive edge weights, m^- is the sum of the absolute value of the negative edge weights, and n is the number of nodes.

If community detection is performed on a bipartition covariance network and there are no polytomies in the tree set, then for each bipartition, the sum of its covariances with all other bipartitions equals zero. Correspondingly, the sum of the absolute value of the positive covariances is equal to the sum of the absolute value of the negative covariances. In this case, Equation (6) simplifies to

$$\mathcal{H}(\{\sigma\}) = - \sum_{i,j} \left[A_{i,j} - p_{i,j}^+ (\lambda^+ - \lambda^-) \right] \delta(\sigma_i, \sigma_j). \quad (7)$$

For this model, changing the λ tuning parameters affects the preferred community size by adjusting the reward and penalty for including positive and negative edges, respectively, in a community. In the simplified equation, if $\lambda^+ > \lambda^-$, then small communities are preferred. If $\lambda^- > \lambda^+$, then large communities are preferred.

A.5.3 Configuration Null Model

For the Configuration Null Model (CNM), \mathcal{H} is given by

$$\mathcal{H}(\{\sigma\}) = - \sum_{i,j} \left[A_{i,j} - \left(\frac{k_i^+ k_j^+}{2m^+} \lambda^+ - \frac{k_i^- k_j^-}{2m^-} \lambda^- \right) \right] \delta(\sigma_i, \sigma_j), \quad (8)$$

where k_i^+ is the sum of all positive edges connecting nodes i , k_i^- is the sum of the absolute value of all negative edges connecting to node i , m^+ is the sum of the positive edges in community σ , and m^- is the sum of the absolute value of all negative edges in community σ . All of the other terms are as above.

If bipartition-covariance community detection is performed and there are no polytomies in the tree set the above equation simplifies to:

$$\mathcal{H}(\{\sigma\}) = - \sum_{i,j} \left[A_{i,j} - \frac{k_i^+ k_j^+}{2m^+} (\lambda^+ - \lambda^-) \right] \delta(\sigma_i, \sigma_j). \quad (9)$$

As with ERM, changing the λ tuning parameters affects the preferred community size by adjusting the reward and penalty for including positive and negative edges, respectively. In the simplified equation, if $\lambda^+ > \lambda^-$, then communities are more likely to include negative edges. If $\lambda^- > \lambda^+$, then communities are less tolerant of negative edges and more strongly favor only positive edges.

A.5.4 Constant Potts Model

For the Constant Potts Model (CPM), \mathcal{H} is given by

$$\mathcal{H}(\{\sigma\}) = - \sum_{i,j} [A_{i,j} - (\lambda^+ - \lambda^-)] \delta(\sigma_i, \sigma_j), \quad (10)$$

where all terms are as above. As with the other models, changing the values of the λ tuning parameters affects the preferred community size. If $\lambda^+ > \lambda^-$, small communities are preferred. If $\lambda^- > \lambda^+$, large communities are preferred.

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.
- Fortunato, S. (2010). Community detection in graphs. *Physics Reports*, 486(3–5):75 – 174.
- 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.
- Newman, M. E. J. (2010). *Networks: An Introduction*. Oxford University Press, Inc., New York, NY, USA.
- Raghavan, U. N., Albert, R., and Kumara, S. (2007). Near linear time algorithm to detect community structures in large-scale networks. *Phys. Rev. E*, 76:036106.
- Reichardt, J. and Bornholdt, S. (2006). Statistical mechanics of community detection. *Phys. Rev. E*, 74:217 – 224.
- Stachniss, C. (2004). C implementation of the hungarian method. *Last accessed October, 27:2015*.
- Stockham, C., Wang, L.-S., and Warnow, T. (2002). Statistically based postprocessing of phylogenetic analysis by clustering. *Bioinformatics*, 18(suppl_1):S285.
- Sul, S.-J. and Williams, T. L. (2011). Big cat phylogenies, consensus trees, and computational thinking. *Journal of Computational Biology*, 18(7):895–906.
- Traag, V. (2014). *Algorithms and Dynamical Models for Communities and Reputation in Social Networks*. Springer, Heidelberg.
- Traag, V. and Bruggeman, J. (2009). Community detection in networks with positive and negative links. *Physical review. E, Statistical, nonlinear, and soft matter physics*, 80:036115.
- Traag, V. A., Van Dooren, P., and Nesterov, Y. (2011). Narrow scope for resolution-limit-free community detection. *Phys. Rev. E*, 84:016114.
- Whidden, C. and Matsen, F. A. (2018). Calculating the unrooted subtree prune-and-regraft distance. *IEEE/ACM transactions on computational biology and bioinformatics*, 16(3):898–911.
- Wilgenbusch, J. C., Huang, W., and Gallivan, K. A. (2017). Visualizing phylogenetic tree landscapes. *BMC Bioinformatics*, 18(1):85.