PhylogeneticGraph
User Manual
Version 0.1

June 23, 2023

American Museum of Natural History

Program and Documentation
Ward C. Wheeler

Program
Alex Washburn

Documentation
Louise M. Crowley

Louise M. Crowley, Alex Washburn, Ward C. Wheeler
Division of Invertebrate Zoology, American Museum of Natural History, New York, NY, U.S.A.

The American Museum of Natural History
©2023 by The American Museum of Natural History,
All rights reserved. Published 2023.

Available online at `https://github.com/amnh/PhyGraph`
Comments or queries relating to the documentation should be sent to wheeler@amnh.org or crowley@amnh.org

Contents

1 What is PhyG? 5
1.1 Introduction 5
1.2 Quick Start 5
1.2.1 Requirements: software and hardware 5
1.2.2 Obtaining and Installing `PhyG` 5
1.3 Overview of program use 6
1.3.1 Executing Scripts 7
1.3.2 Creating and running `PhyG` scripts 7
1.3.3 Execution in Parallel 7
1.4 Input Data Formats 8
1.4.1 fasta 8
1.4.2 fastc 8
1.4.3 `TNT` 9
1.5 Input Graph Formats 9
1.6 Output Graph Formats 9

2 PhyG Commands 11
2.1 `PhyG` Command Structure 11
2.1.1 Brief description 11
2.1.2 Command order and processing 12
2.1.3 Notation 12
2.2 Command Reference 12
2.2.1 Build 12
2.2.2 Fuse 15
2.2.3 Read 16
2.2.4 Reblock 21
2.2.5 Refine 21
2.2.6 Rename 24
2.2.7 Report 24
2.2.8 Run 29
2.2.9 Search 30
2.2.10 Select 31
2.2.11 Set 32
2.2.12 Support 35
2.2.13 Swap 36
2.2.14 Transform 38

Chapter 1

What is PhyG?

1.1 Introduction

PhylogeneticGraph (`PhyG`) is a multi-platform program designed to produce phylogenetic graphs from input data and graphs via heuristic searching of general phylogenetic graph space. `PhyG` is the successor of **POY** [13, 47, 34, 35, 48, 49], containing much of its functionality, including the optimization of *unaligned* sequences, and the ability to implement search strategies such as random addition sequence, swapping, and tree fusing. As in **POY**, `PhyG` can generate outputs in the form of implied alignments and graphical representations of cladograms and graphs. What sets `PhyG` apart from **POY**, and other phylogenetic analysis programs, is the extension to broader classes of input data and phylogenetic graphs. The phylogenetic graph inputs and outputs of `PhyG` include trees, as well as softwired and hardwired networks.

This is the initial version of documentation for the program.

1.2 Quick Start

1.2.1 Requirements: software and hardware

`PhyG` is an open-source program that can be compiled for Mac OSX and Linux (see information relating to Windows machines below). Some utility programs (such as TextEdit for Mac, or Nano for Linux) can help in preparing `PhyG` scripts and formatting data files, while others (such as Adobe Acrobat and TreeView [26]) can facilitate viewing the outputted graphs and trees.

`PhyG` runs on a variety of computers, including desktops, laptops and cluster computers. By default, `PhyG` is a multi-threaded application and will use the available resources of the computer during the analysis (see Execution in Parallel 1.3.3).

1.2.2 Obtaining and Installing `PhyG`

`PhyG` source code, precompiled binaries, test data, and documentation in pdf format, as well as tutorials, are available from the `PhyG` GitHub website.

Installing from the binaries

Download the PhyG binary from the GitHub website. Binaries are available for Mac OSX computers with either Intel or M1 processors, and Linux machines (see information relating to Windows machines below). The user can go directly to the website and click on the appropriate link for the binary. On most systems this will download to either your Desktop or Downloads folder.

Alternatively, open a *Terminal* window (located in your Applications folder) and type the following for either the Mac Intel, Mac M1 or Linux binary:

curl -LJ –output phyg https://github.com/amnh/PhyGraph/blob/main/bin/OSX/phyg-Intel?raw=true

or

curl -LJ –output phyg https://github.com/amnh/PhyGraph/blob/main/bin/OSX/phyg-M1?raw=true

or

curl -LJ –output phyg https://github.com/amnh/PhyGraph/blob/main/bin/linux/phyg?raw=true

The binary should either be moved into your $PATH or referred to its absolute when executing a script.

For those users with Windows machines, a Windows Subsystem for Linux (WSL) can be installed. This system will allow you to run the Linux binary directly on your machine, without having to install a virtual machine or dual-boot setup. The WSL, along with directions for installation, can be found here.

Compiling from the source

For the majority of users, downloading the binaries will suffice. Should the user prefer to compile PhyG directly from the source, the source code can be downloaded from the GitHub website. PhyG is largely written in Haskell. In order to compile PhyG from the source, the user must install Cabal, a command-line program for downloading and building software written in Haskell (GHC). Information on its installation can be found here. To compile an optimized version of PhyG, open a *Terminal* and run the following:

cabal build PhyGraph:phyg –flags=super-optimization

1.3 Overview of program use

At present, PhyG is operated solely via command-line in a *Terminal* window and cannot be executed interactively. Commands are entered via a script file containing commands that specify input files, output files and formats, graph type and search parameters.

1.3.1 Executing Scripts

The program is invoked from the command-line as in:

phyg commandFile

For example, typing the following in a *Terminal* window will invoke `PhyG` to run the script `mol.pg`, which is located in the Desktop folder `phygfiles`:

phyg /Users/Ward/Desktop/phygfiles/mol.pg

This is the equivalent of typing the following from any location on your computer:

cd ("/Users/Ward/Desktop/phygfiles")

1.3.2 Creating and running `PhyG` scripts

A script is a simple text file containing a list of commands to be performed. This script can also include references to other script files (Figure 1.1).

Scripts can be created using any conventional text editor such as *TextEdit*, *TextWrangler*, *BBEdit*, or *Nano*. Do not use a word processing application like *Microsoft Word* or *Apple Pages*, as these programs can introduce hidden characters in the file, which will be interpreted by `PhyG` and can cause unpredictable parsing of the data. Comments that describe the contents of the file and provide other useful annotations can be included. Comment lines are prepended with '--' and can span multiple lines, provided each line begins with '--'.

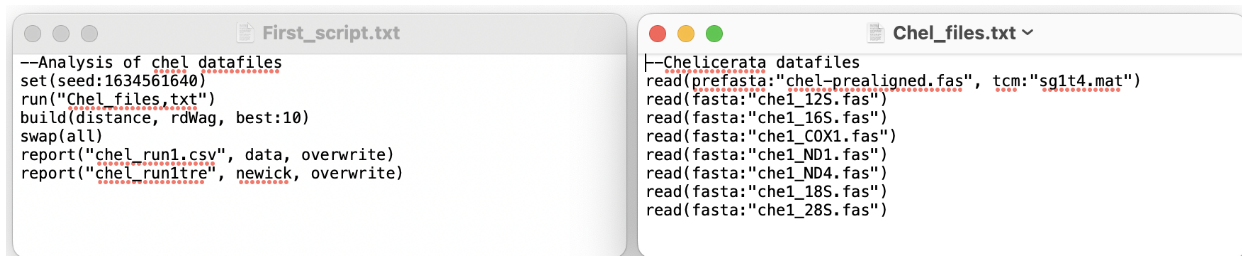

Figure 1.1: `PhyG` scripts. The headers in the scripts are comments, leading with '--', which is ignored by `PhyG` . This first script "**First_script.pg**" includes a reference to the second script "**Chel_files.txt**", which includes a group of data files to be read by the program.

1.3.3 Execution in Parallel

`PhyG` is a multi-threading application and will, by default, use all available cores. Should the user wish to limit or specify the number of processors used by `PhyG` this can be achieved by including the options '**+RTS -NX -RTS**', where '**X**' is the number of processors offered to the program, when executing the script. Should the user wish to use a single processor, this can be specified by typing:

phyg fileName +RTS -N1

This will execute the program sequentially.

1.4 Input Data Formats

`PhyG` can analyze a number of different data types, including qualitative, nucleotide and other sequences (aligned and unaligned), in TNT, fasta, and fastc formats. Any character names in input files are (for now) ignored and internal names are created by appending the character number in its file to the filename as in "**fileName:0**". Qualitative data, and prealigned data include their indices within input files and unaligned data are treated as single characters.

`PhyG` allows the user to comment out portions of a taxon name in the imported data file. This is achieved by inserting a dollar sign ('$') before the region of text that the user wishes to comment out. As an example, placing a '$' before the GenBank information in the taxon name **Lingula_anatina$_AB178773_16S** will comment out this information and the taxon name will be read as **Lingula_anatina** by the program. This can be useful for housekeeping purposes, when it is desirable to maintain long verbose taxon names (such as catalog or NCBI accession numbers) associated with the original data files but avoid reporting these names on the graphs. Moreover, it allows the user to provide a single name for a terminal in cases where the corresponding data are stored in different files under different terminal names.

1.4.1 fasta

Single character sequence input [27] (see Figure 1.2).

1.4.2 fastc

Fastc is a file format for multi-character sequence input [51]. This format is derived from the fasta format and the custom alphabet format of **POY** [34, 48]. Multi-character sequence elements are useful for the analysis of data types such as developmental, gene synteny, and comparative linguistic data (see Figure 1.2). In this format, individual sequence elements are separated by a space.

Figure 1.2: Fasta and Fastc file formats. The file "**chel_cox1aln1.fasta**" represents a fasta file with a greater than sign ('>') preceding taxon names and nucleotide sequence data following on a new line. The file "**woman.fastc**" is a fastc file, with a greater than sign ('>') preceding taxon names and the linguistic data following on a new line.

flu_net1.tre

```
(1466_H7N2_Avian_chicken_pa_1490921_2002:8513.5,((21_H1N1_Swine_swine_france_wvl4_1985:5154.0,
(((2593_H1N1_Human_california_04_2009:4176.5,
(2279_H1N1_Human_iowa_ceid23_2005:7619.5,#HTU15:3152.5)HTU13:7224.0)HTU19:4358.5,
(2083_H1N1_Human_fortmonmouth_1_47:4477.0,(2096_H1N1_Human_newjersey_1976:4300.0,
(2828_H3N2_Human_westernaustralia_40_2003:130.5)#HTU15:130.5)HTU11:3464.0)HTU20:4139.5)HTU12:3593.5,
(2649_H2N2_Human_netherlands_56_1963:88.5)#HTU14:88.5)HTU16:3448.5)HTU17:3685.5,
(36_H1N1_Swine_swine_ohio_23_1935:8273.0,#HTU14:3157.0)HTU18:7410.5)HTU10:4893.0)HTU9[13976.0];
```

Figure 1.3: The file **"flu_net1.tre"** in Enhanced Newick (ENewick) graph format. The values associated with the taxon names and HTUs are branch lengths. The cost of the graph(s) can be found in square brackets at the end of each graph. A semi-colon follows the cost of each graph as a comment.

1.4.3 TNT

The TNT [14] format is accepted here for specification of qualitative, measurement, and pre-aligned molecular sequence data. PhyG can parse all of the basic character settings including activation/deactivation of characters, making characters additive/non-additive, applying weight to characters, step-matrix costs, interleaved data files and continuous characters (see the argument `tnt` in 2.2.3). Continuous characters can only be in the format of integers or floats. Moreover, they must be declared as "additive", otherwise, they will be treated as non-additive character states (see **TNT**). The characters '-' and '?' can be used to indicate that characters or character states are missing or inapplicable. Only one set of ccode commands is allowed per line (e.g. to make everything additive: ccode +.;). Default values of step-matrix weights are not implemented, all step-matrix values must be individually specified. Ambiguities or ranges are denoted in square brackets ('`[ ]`') with a period, as in [X.Y]. Continuous characters are denoted in square brackets with a dash or hyphen, as in [X-Y].

1.5 Input Graph Formats

Graphs can be input in a number of formats, including newick, enhanced newick and dot. Newick tree file format is the parenthetical representation of trees as interpreted by Olsen (linked here). Enhanced Newick [6] is an extension of Newick file format, with the addition of tags ('#') that indicate hybrid zones or reticulation events in the phylogenetic network. Dot is a graph description language and well suited to represent graphs and networks.

1.6 Output Graph Formats

Graph outputs can be in either newick, enewick, dot, and depending on the operating system, eps or pdf formats. Newick tree files can be viewed in other programs like FigTree or Dendroscope. Enewick files can only be viewed in a text editor. PhyG can output a dot file, along with an eps

(on OSX) or pdf (on linux) file that can be viewed in a vector graphics program. The dot file can be viewed (and modified) in *Graphviz*. Note: in order to output pdf files the application *dot* must be installed from the Graphviz website. Graphviz an open-source graph visualization software.

PDFs can also be generated directly from dot files. In a *Terminal*, type the following:

```
dot -Tpdf myDotFile.dot > myDotFile.pdf
```

Multiple 'dot' graphs can be output in a single file. To create pdf and other formats the commandline would be (these files are named and numbered automatically):

```
dot -Tpdf -O myDotFile.dot
```

In OSX the 'pdf' option does not currently seem to work. In this case, type the following:

```
dot -Tps2 myDotFile.dot > myDotFile.ps
```

'-Tps2' will generate a postscript file that *Preview* can read and convert to pdf.

Chapter 2

PhyG Commands

2.1 PhyG Command Structure

2.1.1 Brief description

PhyG interprets and executes scripts coming from an input file. A script is a list of commands, separated by any number of whitespace characters (spaces, tabs, or newlines). Each command consists of a name, followed by an argument or list of arguments separated by commas (‘,’) and enclosed in parentheses. Commands and arguments are case insensitive with the exception of filename specifications, which must match the filename *exactly*, including suffixes and are always in double quotes (“**fileName**”). Most commands, with the exception of reblock, rename and transform, have default values and are provided at the end of each command section below. If no arguments are specified, the command is executed under its default settings.

PhyG recognizes three types of command arguments: *primitive values*, *labeled values*, and *lists of arguments*.

Primitive values can either be an integer (INT), a real number (FLOAT), a string (STRING) or a boolean (BOOL) i.e. True|False.

Labeled arguments consist of an argument and an identifier (the label), separated by the colon (‘:’) character. Examples of identifiers include majority, nopenalty, hardwired, and parsimony.

List of arguments is several arguments enclosed in parenthesis and separated by commas (‘,’). Most arguments are optional, with only a few requiring specification, e.g. the build method of distance must be specified. In cases where an argument must be specified, PhyG will report a warning message in the output of the terminal window. When no arguments are specified, PhyG will use the default values.

The following examples illustrate the structure of valid PhyG commands.

build()

In this simple example, the command build is followed by an open and closed parenthesis. As

no arguments are specified, `PhyG` will use the defaults, so this is equivalent to `build(character, replicates:10)`.

read(nucleotide:"chel-prealigned.fas", tcm:"sg1t4.mat")

In this second example, the command `read` reads data in the file (`STRING`) "**chel-prealigned.fas**", parsing it as nucleotide sequence data. It also read the data in the file "**sg1t4.mat**", parsing it as a transformation cost matrix, with indel and substitution costs set to 1.

swap(drift:3, acceptequal:2.0)

In the third example, the command `swap` is followed by the list of arguments that includes `drift` and `acceptequal`, enclosed in parentheses and separated by a comma. Both of these are labeled-value arguments, ascribed an `INTEGER` and a `FLOAT`, respectively.

set(graphfactor:nopenalty)

In this fourth and final example, the command `set` is followed by the labeled-value argument `graphfactor` with the label `nopenalty`.

2.1.2 Command order and processing

The commands `read`, `rename`, `reblock`, and `set` are executed at the beginning of program execution, irrespective of where they appear in the command script. All other commands are executed in the order they are specified.

2.1.3 Notation

Commands are listed alphabetically in the next section. Commands are shown in `terminal` typeface. Optional items are enclosed in square brackets ('`[ ]`'). Primitive values are shown in `UPPERCASE`.

2.2 Command Reference

2.2.1 Build

Syntax

`build([argument list])`

Description

Builds initial graphs. The arguments of `build` specify the number of graphs to be generated, and whether the build is based on *distance* or *character* methods. Most options, with the exception of `dWag` and `nj`, $O(n^3)$ distance based options, are $O(n^2)$. Distance methods are considerably faster (lower constant factor), but approximate, compared to character-based methods. Refinement, in the form of branch swapping (`none`, `otu`, `spr`, and `tbr`) can be specified within the command for distance builds. Refinement for character-based Wagner builds occurs after the `build` process through `swap` and other refinement commands. Given the large time complexity, distance refinement is usually not worth the effort [44]. `build` does not replace graphs previously stored in memory.

Arguments

block Performs independent builds for each 'block' of data. If this option is not specified, the builds are performed combining all the data. Builds are performed according to the other options, i.e. `character` or `distance`. The resulting tree or `graph` is reconciled using the `eun` or `cun` commands. The reconciled graph is resolved into display trees via the `displayTrees`, `first`, and `atRandom` options. This option is especially useful for softwired network search. Associated arguments of `block` include:

cun Reconciles **block** trees into a Cluster-Union-Network [2] before resolution into display trees via `displayTrees` and its associated arguments.

displayTrees[:INT] When the option `block` is specified, this variable returns n display trees specified by this optional argument. If the number of display trees is not specified, up to 2^{63} may be returned.

atRandom When the option `block` is specified, this variable returns the number of display trees as specified by the integer value in `displayTrees[:INT]`, where trees are produced by resolving network nodes uniformly at random. Compare with `first`.

first:INT When the option `block` is specified, this variable specifies that the first number of displays tree resolutions, as specified by the integer value, are chosen for each input graph. Compare with `atRandom`.

eun Reconciles block trees into a Edge-Union-Network [21, 45] before resolution into display trees via `displayTrees` and its associated arguments.

graph When the option `block` is specified, this variable returns the reconciled graph as specified by `eun` or `cun`. The graph may be altered to ensure that it is a 'phylogenetic' graph sensu [24].

character Performs random addition sequence Wagner [9] builds ($O(n^2)$) for tree construction. If the graphtype is specified as softwired or hardwired, the resulting trees are rediagnosed as softwired graphs. This is the default method for tree construction.

distance Causes a pairwise distance matrix to be calculated ($O(n^2)$) and used as a basis for distance tree construction. Specifies how the builds are refined (`none`, `otu`, `spr`, `tbr`), as well as how the tree is constructed (`dWag`, `nj`, `rdWag`, `wpgma`). Distance trees are subsequently rediagnosed as character trees and returned for further analysis. Associated arguments of `distance` include:

dWag Performs distance Wagner build as in [10] choosing the 'best' taxon to add at each step, yielding a single tree. This method has a time complexity of $O(n^3)$.

nj Performs Neighbor-Joining distance build [29], yielding a single tree. This method has a time complexity of $O(n^3)$.

none No refinement (`otu`, `spr`, `tbr`) is performed after distance builds. `none` is the default refinement method.

otu Specifies that OTU refinement [44] is performed after distance builds.

rdWag Performs random addition sequence distance Wagner builds [10, 44], yielding multiple trees determined by the argument `replicates`. This method has a time complexity of $O(m \times n^2)$ for m replicates and n taxa.

best:INT Applies only to `rdWag`. Specifies the number of trees retained after `rdWag` builds, selecting the best trees in terms of distance cost. The options can be used to reduce the number of trees retained.

spr Specifies that SPR refinement [8] is performed after distance builds.

tbr Specifies that TBR refinement [11, 32] is performed after distance builds.

wpgma Performs Weighted Pair Group Method with Arithmetic Mean distance build [31], yielding a single tree. This method has a time complexity of $O(n^2)$.

replicates:[INT | Applies to `rdWag` and `character`. Specifies the number of random addition sequences to be performed, as indicated by the integer value. By default 10 random addition sequences are performed.

return:[INT | Applies to `rdWag` and `character`. This limits the number of Wagner trees returned for further analysis, to the value as specified by the integer. By default all graphs that are built are returned, unless limited by `best` in distance analysis. Limiting the number of returned trees (as opposed to simply generating that number) can result in a larger memory footprint.

Defaults

`build(character, replicates:10)` By default, `PhyG` will build 10 graphs using a random addition of sequences for each of them.

Examples

- `build(replicates:100)`
  Builds 100 graphs using a random addition sequence (the default) for each of them.

- `build(character, block, graph, cun, displaytrees:5, atrandom)`
  Builds 10 (the default) random addition sequence character Wagner builds, for each block of data. The graph is reconciled into a Cluster-Union-Network, before resolution into 5 display trees. The trees are produced by resolving the network nodes uniformly atrandom.

- `build(distance, rdWag, nj, wpgma)`
  Builds a single 'best' random addition sequence distance Wagner build, a Neighbor-Joining tree, and a WPGMA tree. As the option `block` is not specified, the distance trees are built using all the data. A total of three trees is returned.

- `build(distance, dWag, replicates:1000, best:10)`
  Builds 1000 distance Wagner builds and returns 10 of the lowest cost distance trees. The best trees are chosen arbitrarily, but consistently–the first 10 with the lowest cost.

- `build(distance, rdwag, block, eun, displaytrees:3)`
  Builds 10 random addition sequence Wagner builds for each 'block' of data. The graph is reconciled into a Edge-Union-Network, before resolution into the 3 display trees. The trees are produced by resolving the network nodes.

- `build(distance, block, rdWag, wpgma, replicates:100, best:10, otu)`
  Builds 100 random addition sequence distance Wagner builds and a wpgma tree. OTU swapping is performed on the 10 of the lowest cost random addition sequence Wagner trees. These distance searches, and subsequent refinements are performed on each block of data.

2.2.2 Fuse

Syntax

`fuse([argument list])`

Description

Performs Tree Fusing [15, 22, 23] on the graphs in memory. `fuse` operates on a collection of graphs performing reciprocal graph recombination between pairs of graphs. Non-identical subgraphs with identical leaf sets are exchanged between graphs and the results evaluated. This process of exchange and evaluation continues until no new graphs are found. This can be used in concert with other options to perform a Genetic Algorithm refinement [17]. The behavior of `fuse` can be modified by the use of options specifying SPR and TBR-like rearrangement of the combination process.

Arguments

all During branch swapping-type operations, all rearrangement are tried before choosing a new graph.

best Specifies the method for tree selection, which in this case returns the best graphs found during fuse operations.

keep:INT Limits the number of returned graphs to the integer value specified.

none No branch swapping is performed during the fuse. This is the default.

noReciprocal Turns off `reciprocal` (see below).

once Performs a single round of fusing on input graphs and returns the resulting graphs. Alternatively (and by default) fusing continues recursively until no new graphs are found.

pairs:INT Limits the number of graphs to be fused to the number of pairs as specified by the integer value (as oppose to $\binom{m}{2}$ for m graphs).

atRandom Chooses graphs to fuse uniformly at random when `pairs` is specified.

reciprocal By default, fuse takes a subgraph of one graph in a pair and replaces the corresponding subgraph in the other. This argument results in the exchange and evaluation of graphs in both directions—roughly doubling both the time and memory footprint.

spr[:INT] Causes the exchanged subgraphs to be tried at multiple positions (up to n edges away from their initial positions, where n equals the integer value).

steepest During branch swapping-type operations, if a better graph is found, swapping shifts greedily to that graph. This is the default if swapping is specified.

tbr[:INT] Causes the exchanged subgraphs to be tried at multiple positions (up to n edges aware from their initial positions) where n equals the integer value. TBR-style rerooting of the exchanged components occurs.

unique Specifies the method for tree selection, which in this case returns all unique graphs found during fuse operations.

Defaults

`fuse(best, none, reciprocal)` By default, `PhyG` keeps all the best graphs found, and continues fusing until no new graphs are found. No branch swapping style rearrangements are performed. The exchange and evaluation of graphs occurs in a reciprocal manner.

Examples

- `fuse(best, once)`
  Fuses input graphs and returns best graphs after a single round of fusing.

- `fuse(tbr, keep:10)`
  Fuses input graphs and preforms TBR-style replacement and rerooting of pruned components returning up to 10 best cost graphs.

- `fuse(spr:3, pairs:5, unique)`
  Fuses input graphs and performs SPR-style swapping, with the exchange of subgraphs being tried at multiple positions up to 3 edges away from their initial position. The number of graph pairs to be fused is limited to 5. All unique graphs found during this operation are returned.

2.2.3 Read

Syntax

`read(argument list)`

Description

Imports file-based information, including data files and graph files. `read` commands must contain an input file (`STRING`). Supported data formats include FASTA, FASTC and TNT files, and graph formats include Dot, Enewick, and Newick. Filenames must be included in quotes. Filenames must include the appropriate suffix (e.g. .fas, .ss, .mat). The exclusion of these suffixes will result in an error. The filename must match exactly, including capitalization. `PhyG` will attempt to recognize the type of input and parse appropriately. Otherwise, the type of file can be explicitly indicated, using one of arguments below. The argument is followed by a colon (':') and the data file name(s), enclosed in quotes, and separated by commas. It is possible to import more than one data file on the same input line of the command script, but only of the same data type. Reading in files of different data types, e.g. amino acid and nucleotide in the same command, will result in an error. Prepending the file type prevents any ambiguity when the file is parsed (e.g. `read(nucleotide:"Chel.fas")`). If the data type is not specified, it is important to verify that the data was interpreted properly (using the command `report("STRING", data)` or by checking the output display in the *Terminal* window). `read` can also use wildcard expressions ('*' and '?'), which can be useful when reading

in multiple files of the same type. For example, `read(preaminoacid:"*.fas*")` imports all files of the FASTA format in the current directory (in this case this will include files that end in both .fas and .fasta). This files will be interpreted as prealigned amino acid sequences. Terminal names should not have spaces in the imported data file, otherwise the names can be incorrectly interpreted by the program.

Arguments

aminoacid:STRING Specifies that the file contents are parsed as IUPAC coded amino acid data in FASTA [27] format. Sequence gaps are removed.

> Note
>
> `PhyG` recognizes the characters '**x**' as representing any IUPAC character in amino acid data, and '**n**' as representing any nucleotide base in nucleotide sequence data files. A question mark character ('**?**') represents either an '**x**' or a gap character '**-**' in amino acid data or an '**n**' or a '**-**' nucleotide sequence data.

block:STRING Specifies that the string contains block information. Each line contains the new block name followed by names of input files to be assigned to that data block. Blocks are initially named as the input file name with "#0" appended. In the examples, data from files "b" and "c" will be assigned to block "a". There can be no spaces in file or block names. This argument is only intended for use with softwired networks. Characters in the same block have the same display tree in a softwired network [52].

```
"a" "b#0" "c#0"
```

dot:STRING Specifies that the file contains a graph in 'dot' format for use with graph rendering software such as GraphViz.

enewick:STRING Specifies that the file contains Enhanced Newick format graph(s) as specified here [6].

exclude:STRING Specifies that the file contains the names of terminal taxa to be excluded from an analysis. Taxa appear in the form of a list, with a single taxon per line. Thus, taxa not included in the list and present in input files, will be included in analysis. Compare with `include`.

fasta:STRING Ensures that file contents are parsed in FASTA [27] format. This is used for single character sequences such as binary streams, IUPAC nucleotide and amino acid sequence data. Sequence gaps are removed.

fastc:STRING Ensures that file contents are parsed in FASTC [51] format. This is used for multi-character sequences such as gene synteny, developmental, or linguistic data. Sequence gaps are removed.

include:STRING Specifies the names of terminal taxa to be included in the analysis. Taxa appear in the form of a list, with a single taxon per line. It is possible to specify terminals that have no data. This may be done in order to diagnose a large graph on partial data. If there are no

data for a leaf taxon, a warning will be printed to `stderr`. Taxa not included in this list, but present in the inputted data files, will be excluded from the analysis. Compare with `exclude`.

newick:STRING Specifies that the file contains Newick format graph(s) as specified here.

nucleotide:STRING Ensures that file contents are parsed as IUPAC coded nucleotide data in FASTA [27] format. Sequence gaps are removed.

> Note
>
> Sequences can be divided into smaller fragments using an assigned character (default '#'). This character can be chosen by the user (unlike in POY, where the pound sign ('#') was the only character used to partition datasets). Each fragment is treated as an individual character. When partitioning the data in this way, the number of partitions must be the same across homologous sequences. The character should be set with the command `set:partitioncharacter` (see Section 2.2.11).

preaminoacid:STRING Specifies that the file contents are parsed as IUPAC coded amino acid data in prealigned FASTA [27] format. Gap characters ("-") in the sequences are maintained and alignment correspondences are not re-examined. Prealigned amino acid sequence data *must* be of the same length.

prefasta:STRING Specifies that the sequences are prealigned in a FASTA format, leaving gap characters ("-") in the sequences and alignment correspondences are not re-examined. This option exists to ensure proper parsing (and in case auto-format detection is incorrect). Prefasta files can include any single character element such as nucleotide sequence data, binary data or IUPAC amino acid sequences. Prealigned FASTA files *must* be of the same length.

prefastc:STRING Specifies that the sequences are prealigned in a FASTC format, leaving gap characters ("-") in the sequences and alignment correspondences are not re-examined. This option exists to ensure proper parsing (and in case auto-format detection is incorrect). Prealigned FASTC files *must* be of the same length.

prenucleotide:STRING Ensures that file contents are parsed as IUPAC coded nucleotide data in FASTA [27] format, leaving gap characters ("-") in the sequences and alignment correspondences are not re-examined. Prealigned nucleotide sequence data *must* be of the same length.

rename:STRING Replaces the name(s) of specified terminals in the file. This command allows for substituting taxon names and can help merge multiple datasets without modifying the original data file. The file contains a series of lines, each of which contains at least two strings—these strings equate to synonyms separated by spaces. The first string (input taxon name) will replace the second and all subsequent strings (taxon names) on that line. The `rename` function can also be specified as a command, see `rename` (Section 2.2.6) for more detail and examples.

STRING Reads the file specified in the path included in the string argument. A path can be absolute or relative to the current working directory. The file type is recognized automatically, but as mentioned previously, this should be confirmed.

tcm:STRING This refers to a file containing a custom-alphabet matrix that specifies varying costs among alphabet elements in a sequence. The elements in the alphabet can be letters, digits, or both. The `tcm` contains two parts: the first line of the file contains the alphabet elements separated by a space and the transformation cost matrix, which follows below. The dash character representing an insertion/deletion or indel character is not specified on the first line of the file, but added to the alphabet automatically. The second part is the `tcm`, which is a square matrix with $n + 1$ elements (n is the size of the alphabet). The positions from left to right and top to bottom in this matrix correspond to the sequence of the elements as they are listed in the alphabet. An extra rightmost column and lowermost row correspond to indel (gap) costs to and from alphabet elements. At present, this matrix must be symmetrical, but not necessarily metric. Non-metric tcms can yield unexpected results. Transformation costs must be integers. If real values are desired, a character can be weighted with a floating point value factor.

For a sequence with four elements alpha, beta, gamma and delta and an indel cost of 5 for all insertion deletion transformations, a valid custom alphabet file is provided below:

alpha	*beta*	*gamma*	*delta*	
0	2	1	2	5
2	0	2	1	5
1	2	0	2	5
2	1	2	0	5
5	5	5	5	0

In this example, the cost of transformation of `alpha` into `beta` is `2`, and cost of a deletion or insertion of any of the four elements is `5`.

tnt:STRING Ensures that file contents are parsed in TNT [14] format. Not all TNT data commands are currently supported. To ensure that the file is correctly parsed, the file must begin with `xread`, followed by an optional comment in single quotes ('this is a comment'), followed by the number of characters and taxa. The data follow on a new line. Taxon names are followed by character state data. Data can be in multiple blocks (interleaved) or in sequential format. These interleaved blocks can consist of a series of single character states without spaces between them, or multiple (or single) character states (e.g. **alpha**) with space between the individual codings. Blocks must be of all one type (i.e. single character codings without spaces, or multi-character separated by spaces). The data block *must* be followed by a single semicolon (';') on its own line.

The character settings (i.e. `ccode` commands) follow the data block, beginning on a new line. These character settings always terminate with a semi-colon ('`;`'). These settings include: activate ('`[`') or deactivate ('`]`'); make additive/ordered ('`+`') or non-additive/unordered ('`-`'); apply step-matrix costs ('`(`') with scopes (e.g. `cc + 10 12;` and `cc (.; costs 0 = 0/1 1 0/2 2 0/3 3 0/4 4 1/2 1 1/3 2 1/4 3 2/3 1 2/4 2 3/4 1;`) including abbreviated scopes ('`cc -.;`'). There may be multiple character setting statements in a single line. Character settings must be followed by `proc/;` on its own line. `PhyG` will not process any file contents that follow

`proc/;`.

Additive/ordered character states must be numbers (integer or floating point). Ranges for continuous characters are specified with a dash within square brackets (e.g. `[1-2.1]`). Character state polymorphism are specified in square brackets without spaces for single character states (e.g. `[03]`), and with spaces for multi-character states.

Dashes in multi-character states (e.g. `Blue-ish`) are treated as part of the character state specification.

Example file:

```
xread
'An example TNT file' 8 5
A 000
B a14
C b22
D ?33
E d[01]4

A Blue-ish -
B Green-ish OneFish
C Rather-Red TwoFish
D Almost-Cyan RedFish
E Orange-definitely BlueFish

A 5.2 - ?
B 5.3 0.3 1.1
C 3.2 0.1 1.1
D 5.2 1.1 0.1
E 5.1 1.1 0.1
;
cc .;
cc + 2;
proc/;
```

Defaults

`read("fileName")` reads data within the data file "fileName" and attempts to recognize the file type and parse accordingly. The assumed file type is printed to `stderr` for verification.

Examples

- `read(nucleotide:"/Users/UserName/Desktop/phyg/metazoa.fas", tcm:"sg1t4.mat")` Reads the file "**metazoa.fas**" located in the path **/Users/UserName/Desktop/phyg/**,

parsing it as nucleotide sequence data. The information in the transformation cost matrix **sg1t4.mat** is applied to this imported sequence data.

- `read(prefasta:"myDnaSequenceFile.fas")`
  Reads sequence data from "**myDnaSequenceFile.fas**" as prealigned data.

- `read(include:"IncludeTaxa.txt")`
  Reads a list of taxa in the file "**IncludeTaxa.txt**" to be included in the analysis. All other taxa not included in this list, but present in the inputted data files, will be excluded from the analysis.

- `read(rename:"RenameFile.txt")`
  Reads the file "**RenameFile.txt**" that contains a list synonyms, where the name of the item listed first will be substituted for all the subsequently listed names.

2.2.4 Reblock

Syntax

`reblock(STRING list)`

Description

Assigns input data to 'blocks' that will follow the same display tree when optimized as 'softwired' networks. By default, each input data file is assigned its own block with the name of the input file. The command `read(block)` (see Section 2.2.3) is used to reassign these data to new, combined blocks. Spaces are not allowed in block names and will produce 'unrecognized block name' errors.

Arguments

STRING list The first argument is the block to be created, the remainder are the input data to be assigned to that block. Blocks are initially named as the input file name with ':0' appended. Blocks are reported using the `report(data)` command.

Defaults

None.

Examples

- `reblock("a","b#0","c#0")`
  Assigns input data from file "**b**" and "**c** to block "**a**", provided each of these files contain a single block of data.

2.2.5 Refine

Syntax

`refine([argument list])`

Description

Performs Genetic Algorithm for any graph type. In addition, it performs edit operations (addition, deletion, and move) on network edges that only applies to softwired and hardwired graphs.

Arguments

acceptEqual[:FLOAT] Specifies that equal cost graphs are accepted with the probability as set by the FLOAT value. This argument can be applied to `drift` and `annealing`.

acceptWorse[:FLOAT] The acceptance of candidate graphs is determined by the probability $1/(wf + c_c - c_b)$, where c_c is the cost of the candidate graph, c_b is the cost of the current best graph, and wf is the values as specified by the float (default 1.0). This argument can be applied to `drift` and `annealing`.

all Turns off all preference strategies to make network edits, by simply trying all possible edits to the graph. This is a memory intensive refinement. The refinement examines the entire rearrangement neighborhood of the current graph before retaining the best (lowest cost) solutions.

annealing[:INT] Specifies the number of rounds (as specified by the integer value) of simulated annealing optimization [20, 18, 7]. This is performed in concert with `netAdd`, `netDel`, and `netMove`. The acceptance of candidate graphs is determined by the probability $e^{-(c_c - c_b)/(c_b*(k_{max}-k)/k_{max})}$, where c_c is the cost of the candidate graph, c_b is the cost of the current best graph, k is the step number, and k_{max} is the maximum number of steps (set by the `steps:m`, default 10).

steps:INT Specifies the number of temperature steps performed during simulated annealing (as specified by the `annealing`) option. The default is 10.

atRandom Network edit neighborhoods are traversed in a randomized order (compare with `inorder`). This will result in different trajectories of the network edit space being explored.

drift[:INT] Specifies the number of rounds of the 'drifting' form of simulated annealing optimization [15] are performed. This is done in concert with `netAdd`, `netDel`, and `netMove`. The acceptance of candidate graphs is determined by the probability $1/(wf + c_c - c_b)$, where c_c is the cost of the candidate graph, c_b is the cost of the current best graph, and wf is the `acceptWorse` (set by the `acceptWorse:m`, default 1.0) option. Equal cost graphs are accepted with probability set by the `acceptEqual` option. `Drift` differs from `annealing` in that there are no cooling steps to modify acceptance probabilities. The maximum number of graph changes is set by `maxChanges`.

ga Synonym of GeneticAlgorithm.

geneticAlgorithm Performs Genetic Algorithm [17] refinement in concert with the following options.

generations:[INT] Specifies the number of generations (sequential iterations) for `ga`. The default is 10.

popsize:[INT] Specifies the population size for `ga`. The default is 20.

recombinations:[INT] Specifies the number of recombination (fusing) events for `ga`. The default is 100.

stop:INT Causes the `ga` to terminate after the number of generations as specified by the integer value without improvement in graph cost. Default is to only terminate when the number generations has been completed.

inorder Contra `atRandom`, network edit neighborhoods are always traversed in the same undefined, but consistent order.

keep:INT Limits the number of returned graphs to that as specified by the integer value.

maxnetedges:INT Specifies the maximum number of network edges, as indicated by the integer value.

netadd Adds network edges to existing input graphs at all possible positions until no better cost graph is found.

netadddelete Consecutively adds and then deletes network edges from input graphs until certain conditions are met.

rounds:INT Specifies the number of combine network add and delete edit operations.

netdel Deletes network edges from input graphs one at a time until no better cost graph is found.

netmove Moves existing network edges in input graphs one at a time to new positions until no better cost graph is found.

steepest Specifies that refinement follows a greedy path, abandoning the neighborhood of the current graph when a better (lower cost) graph is found.

Defaults

`refine(ga, generations:10, popsize:20, recombinations:100)` By default, `PhyG` will perform Genetic Algorithm refine, with its associated default options, if no other arguments are specified.

Examples

- `refine(netadddel, rounds:3, maxnetedges:5)`
  Consecutively adds and then deletes network edges from input graphs until either no improvement (graph cost) is found in a round or until the number of rounds of addition and deletions (in `rounds:n`) is reached. The maximum number of edges is still specified by `maxnetedges:n` within each round.

- `refine(netmove, atrandom, steepest)`
  Moves existing network edges in input graphs one at a time to new positions until there are no more improvements in the cost of the graph. This edit operation follows a greedy path, abandoning the neighborhood of the current graph when a better graph of lower cost is found. During this operation, the network edit neighborhoods are traversed at random.

2.2.6 Rename

Syntax

```
rename(STRING list)
```

Description

Replaces the name(s) of specified terminals in the file. This command can be useful when combining data from different sources, such as GenBank, or in revising names to reflect taxonomic changes. It also allows for merging multiple datasets without modifying the original data file. The command arguments are (minimally) two strings—these strings equate to synonyms separated by spaces. The first string will replace the second and all subsequent strings (taxon names) on that line. In the example given in Figure 2.1 the taxon Hydrus_granulatus will be renamed as Acrochordus_granulatus, the taxa Gloydius_boehmei and Gloydius_mogoi will be renamed as Gloydius_halys and the taxa Crotalus_mutus, Scytale_catenatus and Coluber_crotalinus will be renamed as Lachesis_muta. Irrespective of where this command appears in the script file, `PhyG` will execute this command prior to importing the data files. Compare with the argument `rename` of the command `read` (Section 2.2.3).

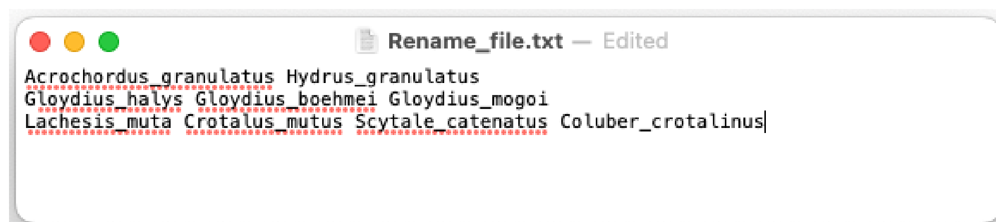

Figure 2.1: Renaming text file containing the lists of terminal taxa to be renamed.

Arguments

STRING list The first argument is the "new name" for the remainder of the following arguments.

Defaults

None.

Examples

- `rename("a","b","c")`
  Renames the terminal names "b" and "c" to the terminal name "a".

2.2.7 Report

Syntax

```
report([STRING, argument list])
```

Description

Outputs the results of the current analysis to a file or to `stderr`. To redirect the output to a file, the file name (in quotes), followed by a comma, must be included in the argument list of report. All arguments for `report` are optional. This command allows the user to output information concerning the characters and terminals, diagnosis, export static homology data, implied alignments, trees, graphs, dot files, as well as other miscellaneous arguments. By default, new information printed to a file is appended to that file. The option `overwrite` overrides the default and rewrites the file. Many of the report options are output in csv format, which can subsequently be imported into spreadsheet programs like *Excel* or *Numbers* for easy viewing.

Arguments

append When reporting data or graphs to a file, this information is appended to the end of the file. Compare with the `overwrite` argument. By default, files are appended to the report, rather than overwritten.

ascii Reports ASCII character representations of the reported graphs. This file can be viewed in any text editor.

branchlengths When reporting graphs, `PhyG` will report the branch lengths of these graphs. In newick and enewick files, branch lengths follow the terminal names, separated by a colon. In eps and pdf files, branch lengths appear above the branches. In ASCII and dot files, branch lengths follow the terminal labels. Compare with `nobranchlengths`. This is the default. is this okay?

collapse Specifies that zero length branches are collapsed. Compare with the `nocollapse` argument. If a dotpdf graph is specified, the branches are collapsed by default. If ASCII, newick, enewick and dot are specified, the zero length branches are not collapsed by default.

crossrefs Reports whether data are present or absent for each terminal in each of the imported data files. The argument will report a table with terminals represented in rows, and the data files in columns. A plus sign ('+') indicates that data for a given terminal is present in the corresponding file; a minus sign ('–') indicates that it is absent. This provides a comprehensive visual overview of the completeness of the data. It will highlight missing data, as well as inconsistencies in the spelling of taxon names in different data files (see Figure 2.2). The reported file is in csv format.

data Outputs a summary of the input data and terminals. This file summarizes information relating to the input data (number of terminals, number of input files, number of character blocks and the total number of characters). It also provides information relating to the terminal taxa included in the analysis, including the names of the taxa, a list of the excluded taxa (if any), and whether any terminals were renamed. In this file you will also see information relating to "Index", "Block", "Name", "Type", "Activity", "Weight", "Prealigned", "Alphabet", and "TCM". "Index" reports the character number in the overall dataset; "Name" reports the name of the character (by default based on its source data file); "Type" is the type of character (e.g. Non-Additive, Matrix, Nucleotide Sequence), "Activity" reports whether the character is active (included in the analysis) or not (excluded), "Weight" is the weight of the

	A	B	C	D	E
1		Mollusca_cox1.fas#0	Mollusca_16S.fas#0		
2	Antarctic_neomeniomorph	+	-		
3	Aplysia_californica	+	+		
4	Cerebratulus_lacteus	+	-		
5	Chaetopleura_apiculata	+	+		
6	Crassostrea_virginica	-	+		
7	Euprymna_scolopes	+	+		
8	Greenland_neomeniomorph	+	+		
9	Helobdella_robusta	-	+		
10	Ilyanassa_obsoleta	-	+		
11	Lingula_anatina	+	+		
12	Littorina_littorea	+	+		
13	Lottia_gigantea	+	+		
14	Lymnaea_stagnalis	-	+		
15	Mytilus_californianus	+	+		
16	Perotrochus_lucaya	+	+		
17	Scaphopods	+	+		
18	Siphonaria_pectinata	+	+		
19	Venerupis_decussatus	+	+		
20	Venerupis_philippinarum	+	+		
21	Yoldia_limatula	+	-		
22					

Figure 2.2: The figure shows a crossrefs files, which has been imported into *Excel*.

character, "Prealigned" denotes whether a sequence character (e.g. amino acids) is to treated as prealigned or not, "Alphabet" the elements of a sequence character, "TCM" is the transition cost matrix specifying costs among sequence elements and "gap" or insertion-deletion. The reported file is in csv format.

diagnosis Outputs graph diagnosis information such as vertex, states and edge statistics. The reported file is in csv format.

displaytrees Reports graph information for softwired networks. The 'display' trees are output for each data block.

dot Outputs a graph in dot format. The dot file can be viewed (and modified) in *Graphviz* (see also `dotpdf`). In order to output pdf files the application *dot* must be installed from the Graphviz website. Graphviz is open-source graph visualization software. `dot` is the default graph representation—a dot file will only be reported if no other graph type is specified.

dotpdf Outputs two files—a graph file in dot format, along with either an eps (on OSX) or pdf (on Linux) (see also `dot`). The eps and pdf files can be read in *Adobe Illustrator*, *Apple Preview* or any vectorial image edition program. By default, when `dotpdf` is specified, edges are collapsed (contracted) if they have a minimum weight (length) of 0.

graphs Outputs a graph in the format specified by the other arguments in the command. These are either newick, enewick, ascii, dot, and depending on the operating system, eps (on OSX) or pdf (on Linux) formats.

htulabels Labels the HTUs in the output files (compare with `nohtulabels`). This is the default.

ia Synonym of `impliedalignment`.

impliedalignment Outputs the implied alignments of the specified set of characters in FASTA or FASTC (depending on sequence type) format. This argument is synonymous with the argument `ia`. By default, an implied alignment is reported for each block of input data.

concatenate This optional argument can be used with `ia` | `impliedalignment`. Instead of outputting an implied alignment for each block of input data, it will report a file with all the implied alignments concatenated into a single file.

includemissing Outputs a fasta or (fastc depending on sequence type), including taxa in the implied alignment, which are missing data for that particular block of data. In this case, `PhyG` will print '?' characters for the missing taxon in this block of the implied alignment. This option interacts nicely with `concatenate` in creating prealigned fasta (implied alignments) with all the terminals included for all the data. By default, taxa with missing data are not included.

newick Outputs graphs in the E/Newick format, with the terminals separated with commas, and graphs separated with semicolons. Branch lengths follow terminals, separated by a colon.

nobranchlengths When reporting graphs, `PhyG` will by default report the branch lengths of these graphs. In newick and enewick files, branch lengths follow the terminal names, separated by a colon. In eps and pdf files, branch lengths appear above the branches. In ASCII and dot files, branch lengths follow the terminal labels. The argument `nobranchlengths` will override this default and branch lengths of graphs are not reported.

nocollapse By default, if a dotpdf graph is specified, the zero-length branches are collapsed. Compare with the `collapse` argument. If ASCII, newick, enewick and dot are specified, the zero length branches are not collapsed by default. If `nocollapse` is specified, zero length branches are not collapsed for output graphs.

nohtulabels Labels of the HTUs are not included in the output files. This option can not be applied to enewick graph. Compare with the argument `htulabels`.

overwrite By default, when reporting data or graphs to a file, this information is appended to the end of the file. The option `overwrite` overrides this default and rewrites the file rather than appending to the existing information.

pairdist Outputs a taxon pairwise distance matrix in csv format.

reconcile Outputs a single 'reconciled' graph from all graphs in memory. The methods include consensus, supertree, and other supergraph methods as described in [42, 45]. When `reconcile` is specified as a command option a series of other options may be specified to tailor the desired outputs:

Compare: Specifies how group comparisons are to be made.

combinable Group comparison are made by identical match [(A, (B,C))≠(A,B,C)]. This is the default.

identity Group comparison are made by combinable sensu [25] [(A, (B,C)) consistent with (A,B,C)]. This option can be used to specify 'semi-strict' consensus [3].

Connect:BOOL Specifies that the output graph be connected (single component), potentially creating a root node and new edges labeled with "0.0". The default value is TRUE. This option will connect any forest elements into a single graph.

Method: Specifies the type of output graph.

cun Graphs are reconciled using the Cluster Union Network [2] method. This argument can used in conjunction with `threshold`.

eun Graphs are reconciled using the Edge-Union-Network method of [21]. This argument can used in conjunction with `threshold`. This is the default.

majority Specifies that values between 0 and 100 of either vertices or edges will be retained. If all inputs are graphs with the same leaf set this will be the Majority-Rule Consensus [19]. This argument is used in conjunction with `threshold`.

strict Strict requires all vertices be present to be included in the final graph. If all inputs are graphs with the same leaf set this will be the Strict Consensus [30].

Adams Adams denotes the Adams II consensus [1].

EdgeLabel:BOOL Specifies the output graph have edges labeled with their frequency in input graphs. The default value is TRUE.

VertexLabel:BOOL Specifies the output graph have vertices labeled with their subtree leaf set. The default value is FALSE.

Threshold:INT Specifies the threshold (between 0 and 100) frequency of vertex or edge occurrence in input graphs to be included in the output graph. Affects the behavior of 'eun,' 'cun,' and 'majority'. The default value is 0.

search Outputs search statistics in csv format. See Section 2.2.9 for details about the randomized series of graph optimization methods included in each iteration of a timed search.

STRING Specifies the name of the file to which all types of report outputs, designated by additional arguments, are printed. If no additional arguments are specified, a graph in dot format, along with an eps or pdf file (depending on the operating system) will be reported to a file named '**defaultGraph**'. By default files are appended to the report, rather than overwritten. See `overwrite`.

support Outputs support graphs (see Section 2.2.12) that have been previously calculated by the command `support`. Resampling graphs [12] are independent of the input graphs while Goodman-Bremer graphs [16, 4] are based on current graphs. Graphs can be output in multiple formats with the use of the options `ascii`, `newick`, `enewick`, `dot`, and depending on your operating system, `dotpdf` for pdf (Linux) or eps (OSX). See Section 1.6 for information relating to viewing and installation requirements. Should you fail to choose one of these reporting formats, `PhyG` will issue a warning in the output display of the *Terminal* window, and output a 'dot' file that can be processed later.

tnt Outputs a file in TNT [14] format (see 1.4.3) using `impliedAlignment` for unaligned sequences.

Defaults

`report(dot, append, branchlengths, htulabels, collapse)` The default graph representation is `dot`. A dot file will only be reported if no other graph type is indicated. When writing to a file, the information is appended to the end of the file (see `overwrite` for details). Branch lengths and HTU labels are printed in the output files (see `nobranchlengths` and `nohtulables` respectively for details). Zero length branches are collapsed (see `nocollapse` for details).

Examples

- `report("outFile.tre", newick, overwrite)`
  Outputs graphs in newick format to the file "**outFile.tre**", overwriting any existing information.

- `report("outFile_cr.csv", crossrefs)`
  Outputs a cross reference file, with data pertaining to the presence and absence for taxa in input files. This information is appended to the end of the file, if existing information exists.

- `report("outFile", dot, reconcile, method:eun, threshold:51)`
  Outputs a graph which has been reconciled using the Edge-Union-Network method with a minimum edge frequency of 51%. The graph is outputted in dot format to the file "**outFile**', appending to any existing information in the file.

2.2.8 Run

Syntax

`run(STRING)`

Description

Executes `PhyG` script file(s) containing commands. The filename must be included in quotes. Executing scripts using `run` can be useful to specify common actions such as inputting file(s) and and graph construction.

Arguments

STRING The only argument is the name of the script-containing file containing the commands to be executed.

Defaults

There are no default settings of `run`.

Examples

- `run("readFiles.pg")`
  Executes "**readFiles.pg**", which may contain multiple input files to be `read`.

- `run("searchCommands.pg")`
  Executes "**searchCommands.pg**", which may contain commands defining a common search strategy (e.g. `build`).

2.2.9 Search

Syntax

`search([argument list])`

Description

This command implements a randomized search strategy, performing a timed series of graph optimization methods including building, swapping, recombination (fusing), simulated annealing and drifting, network edge edits (addition, deletion and moving), and Genetic Algorithm. The parameters and their order of implementation are randomized. The arguments associated with this command specify the duration and number of independent instances of the `search`. Successive rounds of `search` gather any solutions from previous sequential or parallel rounds, as well as any input graphs. Since search methods may vary in how long they take, individual iterations may take longer that the specified duration. By default, search strategies are chosen uniformly at random, and if `Thompson` is specified, Thompson sampling [33, 46] is used to modify the probabilities of search strategies over the search duration.

When performing a `search` it is important to set the amount of time, such that the program has a reasonable amount of time to perform a search. Therefore, it is important to have some idea as to the length of time it would take to do a single round of searching. Performing a simple search that included build, swap, and network edits (if the graph search is for a network) calculating the amount of time for a single graph within this search provides some approximation as to the amount of time necessary to perform a thorough search. Obviously, this estimate is data and optimality criterion dependent. With this information, the user can then estimate the amount of time necessary to perform a thorough search (perhaps 10 times the amount of time it took to perform this simple search).

Arguments

days:INT Adds the number of days (as specified by the integer) to the maximum total execution time for the search.

hours:INT Adds the number of hours (as specified by the integer) to the maximum total execution time for the search.

instances:INT Specifies the number of (potentially parallel) search instances, as indicated by the integer value. The overall search will terminate only when all instances have completed.

maxNetEdges:INT Limits the maximum number of network edges to that specified by the integer value. This should only to be used if `graphtype` has been `set` to `hardwired` or `softwired`.

minutes:INT Adds the number of minutes (as specified by the integer) to the maximum total execution time for the search.

seconds:INT Adds the number of seconds (as specified by the integer) to the maximum total execution time for the search.

stop:INT Causes a search instance to terminate after the number of iterations without graph optimality improvement. This argument is an adjunct to the specified time. The search iteration will continue until the maximum execution time for the search has ended or until the cost of the best graph fails to improve after the specified integer (whichever comes first). The overall search will terminate only when all instances have completed.

Thompson Specifies that randomized choice of search option (e.g. Wagner Build, SPR, Genetic Algorithm) employs Thompson sampling [33]. This sampling method is a heuristic for timed search decision making.

exponential Specifies that the Thompson memory is an exponential function of `mFactor`, m. Updating of search type, θ^k, probability for iteration n is $\left(1 - \frac{1}{2^m}\theta^k_{n-1}\right) + \left(\frac{1}{2^m}\theta^k_n\right)$. Thompson success is a function of whether a search was successful in reducing the graph cost and how long that search took to complete.

linear Specifies that the Thompson memory is a linear function of `mFactor`, m. Updating of search type, θ^k, probability for iteration n is $\frac{m}{m+1}\theta^k_{n-1} + \frac{1}{m+1}\theta^k_n$. Thompson success is a function of whether a search was successful in reducing the graph cost and how long that search took to complete.

mFactor:INT Specifies the memory factor for Thompson sampling, as indicated by the integer value. The lower the value (down to 0), the shorter the memory allowing for more rapid adjustment of search strategy probabilities. This parameter is applied to both the `linear` and `exponential` arguments of `Thompson`.

Defaults

`search(instances:1, seconds:30, keep:10)` Under the default parameters, `PhyG` will perform 1 instance of a search for at most 30 seconds, keeping up to 10 graphs.

Examples

- `search(hours:10, instances:2)`
  Performs 2 search instance (in parallel if the program is executed in parallel) for 10 hours each.

- `search(hours:10, minutes:30)`
  Performs a single search instance for 10 hours and 30 minutes.

- `search(days:3, thompson, linear, mfactor:5)`
  This command will attempt as many iterations of the randomized search strategy, employing Thompson sampling, with linear memory function and the mFactor value set to 5.

2.2.10 Select

Syntax

`select([argument list])`

Description

Specifies the method and number of graphs to be saved at any point during the analysis. When multiple graphs are present, the `select` command will specify which of the graphs to keep for subsequent analysis or reporting.

Arguments

all Specifies all the graphs are kept.

atRandom:[INT] Randomly selects the graphs irrespective of cost. The number of chosen graphs can be specified with an integer value. This can be a useful tool for randomized searches and subsequent analyses where randomized sampling is desired.

best:[INT] Selects the number of best or lowest cost graphs. The number of returned graphs can be specified by the integer value. If the number of optimal graphs exceeds the value of best, the subset of optimal graphs are chosen in an undefined but consistent order.

threshold:FLOAT Keeps unique graphs up to fraction longer than shortest graph. The `FLOAT` is a floating point number (e.g. 0.1) and should be greater than 0.

unique:[INT] Selects only topologically unique graphs, irrespective of their cost. The program will keep as many graphs as specified by the integer value.

Defaults

`select(best)` Keeps all unique graphs of best optimality value.

Examples

- `select(atrandom:10)`
  Keeps up to 10 graphs, selecting the graphs at random.
- `select(best:10)`
  Keeps up to 10 graphs of best optimality value.

2.2.11 Set

Syntax

`set(argument string)`

Description

Changes the settings of `PhyG`. This command performs an array of functions from specifying the seed of the random number generator, to selecting a terminal for rooting output graphs, to specifying graph type, final assignment, and optimality criterion. All `set` commands are executed at the start of a run, irrespective of where they appear in the script. The command `transform` is used to modify global settings during a run (see `transform` Section 2.2.14). `set` arguments have to be indicated on separate lines of the script, otherwise `PhyG` will return an error.

Arguments

criterion: Sets the optimality criterion for graph search.

parsimony Graph costs are determined by the parsimony criterion, that is the minimum sum of transformation events multiplied by their weights. This is the default.

ncm Graph costs are determined by the No-Common-Mechanism likelihood model of [?]. This is exact for trees, but for networks should be treated as a form of character weighting.

finalAssignment: Sets the method of determining the 'final' sequence states.

DirectOptimization DirectOptimization uses the DO method to assign the final states. This is more time consuming than `ImpliedAlignment`. DO has an additional factor of potentially $O(n^2)$ in sequence length compared to the constant factor for IA due to additional graph traversals. This is the default.

DO Synonym of `DirectOptimization`.

IA Synonym of `ImpliedAlignment`.

ImpliedAlignment Uses implied alignments to assign the final states. This final assignment procedure has a lower time complexity than DO.

graphFactor: Sets the network penalty for a softwired network. When conducting a network analysis, a penalty can be specified, so that 'softwired' phylogenetic networks can compete equally with phylogenetic trees on a parsimony optimality basis. A network penalty takes into account the change in cost as edges are added to the graph. Hardwired graphs are automatically set to `NoPenalty`.

nopenalty No penalty is added.

w15 Employs the parsimony network penalty of [43].

w23 Sets the parsimony network penalty of [52], which is more severe but has a lower time complexity than W15.

graphsSteepest:[INT] Sets the maximum number of graphs to be evaluated simultaneously during 'steepest' descent in swapping and network addition and deletion operation. The number is the lower of the number of parallel threads and n (default 10). Setting this number to higher values can improve the use of parallel resources, but at the cost of additional memory footprint. Accordingly, reduction in this value can reduce memory consumption.

graphType: Sets the phylogenetic graph type. The program allows for the input, analysis of and output of a broad class of phylogenetic graphs. These include trees, and both 'softwired' and 'hardwired' networks.

tree Sets the graph type to tree. This is the default.

hardwired Sets the graph type to hardwired. A 'hardwired' network is one where characters can have multiple parents.

softwired Sets the graph type to softwired. A 'softwired' network is a summary of a set of individual 'display' trees that have been generated by removing edges from the network. Individual characters have single parents.

multiTraverse:[BOOL] Controls the multi-traverse character optimization option. If `True`, character trees are traversed from each edge as in [36, 37, 34, 48], but individually for each dynamic character. This is the default and yields better (lower) optimality scores at the cost of added execution time. This is the default. If `False`, only outgroup-based traversals are performed as in [39, 13, 47].

outgroup:STRING Specifies the terminal to root the output trees. This name must appear in double quotes. If the outgroup is not set, the default outgroup is the taxon whose name is lexically first after any renaming of taxa, and/or if taxa were specified by using the arguments `include` or `exclude`. Only a single taxon can be set as the outgroup of the analysis.

partitioncharacter:STRING Sets the character that is used to partition the data. Sequences can be divided into smaller fragments using an assigned character. This single character is chosen by the user (unlike in POY, where the pound sign ('#') was the only option to partition datasets). Each fragment is treated as an individual character. When partitioning the data in this way, the number of partitions must be the same across homologous sequences. The default is '#'.

rootCost: Sets the root cost for a graph.

noRootCost No cost is ascribed to the root. This is the default for parsimony searches.

w15 Sets a cost at $\frac{1}{2}$ the cost of 'inserting' the root character assignments. The W15 root cost is based on the same rationale as the parsimony network penalty of [43].

seed:INT Sets the seed for the random number generator using the integer value. If unspecified, `PhyG` uses the system time as the seed. By setting this value, we are guaranteed to reproduce a given search trajectory each time the script is run. This is the case even when the operations are randomized, as is the case when using the argument `atrandom`.

softwiredMethod: Sets the algorithm for softwired graphs to the 'Naive' method of diagnosing all display trees, as in [43] or the 'Resolution Cache' method of [52].

Naive Determines the score of a softwired network by evaluating all display trees (up to 2^m for m network nodes) and summing the costs of the minimum cost display tree for each block of data. This can be extremely time consuming for graphs with large number of network nodes.

ResolutionCache Uses the method of [52] to determine softwired network costs in greatly reduced time. This is the default.

Defaults

```
set(criterion:parsimony)
set(FinalAssignment:DirectOptimization)
set(graphFactor:w15)
```

`set(graphtype:tree)`
`set(outgroup:STRING)` The default outgroup is the taxon whose name is lexically first after renaming of taxa and/or if taxon were specified for inclusion or exclusion.

Examples

- `set(criterion:parsimony)`
  Sets the graph search optimality criterion to parsimony.

- `set(outgroup:"Lingula_anatina")`
  This command selects the terminal "Lingula_anatina" and sets it as the outgroup for the analysis.

- `set(partitioncharacter:@)`
  Sets the partition character in the input dataset to the at sign '@'.

2.2.12 Support

Syntax

`support([argument list])`

Description

Generates graph supports via resampling [12] and Goodman-Bremer [16, 4]. Currently, Jackknifing, Bootstrapping and Goodman-Bremer supports are the resampling methods that are implemented. If the criterion is set to `ncm` (see Section 2.2.11), the support values for Goodman-Bremer are log likelihood ratios.

Arguments

buildonly Performs very rapid, but not extensive graph searches for each resampling replicate.

bootstrap Calculates Bootstrap support. The user can specify the number of iterations, using `replicates`. The edges are labeled with the bootstrap frequencies when output via the command `report(support)`. Default replicate search is 100 random addition sequence distance-based Wagner builds and TBR branch swapping on the best distance tree. If the graph type is softwired or hardwired, `netadd`, `maxnetedges:5`, `atrandom` are added.

gb Synonym of GoodmanBremer.

goodmanBremer Specifies that Goodman-Bremer support is calculated for input graphs.

> **gbsample:[INT]** Specifies the number of alternate graphs to be examined (i.e. limited). The graphs are chosen uniformly at random. This is used to reduce execution time of the Goodman-Bremer support at a cost of potentially increased overestimates (higher upper bound values).
>
> **spr** Traverses the SPR neighborhood to determine an upper-bound on the NP-hard values.

tbr Traverses the TBR neighborhood as optionally specified (TBR as default) to determine an upper bound on the NP-hard values (this is the method used in POY v2; 13 *et seq.*).

jackknife:[FLOAT] Specifies that Jackknife resampling is performed with n acceptance probability (default 0.6231 or $1-e^{-1}$). When reported (via `report(support)`), edges are labeled with the jackknife frequencies. Default replicate search is 100 random addition sequence distance-based Wagner builds and TBR branch swapping on the best distance tree. If the graph type is softwired or hardwired, `netadd`, `maxnetedges:5`, `atrandom` are added.

replicates:[INT] Specifies the number of resampling replicates, as indicated by the integer value, that are performed in resampling support for Jackknife and bootstrap methods. The default is 100.

Defaults

`support(goodmanBremer:TBR)` Calculates Goodman Bremer support values, via traversing the TBR neighborhood.

Examples

- `support(jackknife:0.50, replicates:1000)`
  Performs 1000 replicates of delete 50% jackknife resampling.
- `support(gb, SPR, gbSample:10000)`
  Produces Goodman-Bremer support based on $10,000$ samples of the SPR neighborhood.

2.2.13 Swap

Syntax

`swap([argument list])`

Description

Performs branch-swapping rearrangement on graphs. This command implements a group of algorithms referred to as branch swapping, that proceed by clipping parts of the given tree and attaching them in different positions. These algorithms include 'NNI' [5, 28], 'SPR' [8], and 'TBR' [11, 32] refinement. Default swapping trajectories ar randomized.

Arguments

all Turns off all preference strategies to make a join, simply trying all possible join positions for each pair of clades generated after a break. The refinement examines the entire rearrangement neighborhood of the current graph before retaining the best (lowest cost) solutions. This can be very memory intensive.

alternate Specifies that alternating rounds of `spr` [8] and `tbr` [11, 32] refinement are performed.

annealing[:INT] Specifies the number of rounds of simulated annealing [20, 18, 7] optimization to be performed, as indicated by the integer value (the default is 1). The acceptance of candidate graphs is determined by the probability $e^{-(c_c-c_b)/(c_b*(k_{max}-k)/k_{max})}$, where c_c is the cost of the candidate graph, c_b is the cost of the current best graph, k is the step number, and k_{max} is the maximum number of steps (set by the argument `steps`, with a default of 10).

steps:[INT] Specifies the number of temperature steps to be performed during simulated annealing (as specified by the `annealing`) option. The default is 10.

atRandom Swap neighborhoods are traversed in a randomized order (compare with `inorder`). This will result in different trajectories of the swap space being explored. This is the default.

drift[:INT] Specifies the number of rounds of the 'drifting' form of simulated annealing [15] optimization that are performed, as indicated by the integer value (default 1). The acceptance of candidate graphs is determined by the options `acceptEqual` and `acceptWorse`. `drift` differs from `annealing` in that there are no cooling steps to modify acceptance probabilities. The maximum number of graph changes is set by `maxChanges`.

acceptEqual[:FLOAT] Specifies that equal cost graphs are accepted with the probability as set by the FLOAT value.

acceptWorse[:FLOAT] The acceptance of candidate graphs is determined by the probability $1/(wf + c_c - c_b)$, where c_c is the cost of the candidate graph, c_b is the cost of the current best graph, and wf is the values as specified by the float (default 1.0).

maxChanges:INT Specifies that drifting graph changes are limited to the value as specified by the integer (default 15).

ia Specifies that Implied Alignment [41] assignment are used for branch swapping as opposed to full Direct Optimization for dynamic characters when the graph type is `tree`.

inorder Contra `atRandom`, swap neighborhoods are always traversed in the same undefined, but consistent order.

keep:INT Specifies the number of equally costly graphs to be retained, as determined by the integer value.

nni Specifies that NNI refinement [5, 28] is performed.

replicates:[INT] Sets the number of randomized swap replicate trajectories. Default 1.

spr:[INT] Specifies that SPR refinement [8] is performed. If an integer value is specified, the re-addition of pruned graphs will be within INT edges of its original placement.

steepest Specifies that refinement follows a greedy path, abandoning the neighborhood of the current graph when a better (lower cost) graph is found.

tbr:[INT] Specifies that TBR refinement [11, 32] is performed. If an integer value is specified, the re-addition of pruned graphs will be within $2 * INT$ edges of its original placement.

Defaults

`swap(alternate, keep:10, steepest)`

Examples

- `swap(tbr, all, keep:10)`
  Performs tbr branch swapping on each current graph returning up to 10 best graphs after examining all graphs in the rearrangement neighborhood.

- `swap(annealing:10, steps:10)`
  Performs alternating rounds of `spr` and `tbr` branch swapping (the default) on each current graph, specifying 10 rounds of simulated annealing, with 10 temperature steps being performed for each simulated annealing round (the default number of steps).

2.2.14 Transform

Syntax

`transform(argument list)`

Description

Transforms the properties of the imported characters, graphs, or general values from one value or type to another. It modifies the global settings during program execution (as opposed to `set`, which operates at the start of the analysis). This includes changes of graph type (e.g. tree to softwired Network) and data types (e.g. dynamic to static approximation), among other operations.

Arguments

displayTrees:[INT] Specifies the number output of display trees for each graph. When `toTree` is specified. The number of returned display trees is limited by this integer value, or by the default value (10). Used in concert with `toTree`.

> **atRandom** Returns the number of display trees as specified by the integer in `displayTrees[:INT]`, where trees are produced by resolving network nodes uniformly at random. Compare with the `first`. option, which takes the 'first' number of display trees resolved in arbitrary, but consistent, order.
>
> **first:INT** Specifies that the first number of displays tree resolutions, as specified by the integer value, are chosen for each input graph.

dynamic Reverts data type to the default 'dynamic' for all dynamic homology [40] character types (e.g. DNA sequences). After this command, graph optimization proceeds in the default manner with sequence characters treated in their non-aligned ('dynamic') condition.

dynamicEpsilon Sets the level at which heuristic graph costs are verified by full (and time consuming) traversal. Candidate graphs with costs within `dynamicEpsilon` of the current best graph are verified.

graphFactor Sets the network penalty for a softwired network. When conducting a network analysis, a penalty can be ascribed, so that 'softwired' phylogenetic networks can compete equally with phylogenetic trees on a parsimony optimality basis. A network penalty takes into account the change in cost as edges are added to the graph. Hardwired graphs are automatically set to `NoPenalty`.

nopenalty No penalty is ascribed.

w15 Employs the parsimony network penalty of [43].

w23 Sets the parsimony network penalty of [52], which is more severe but has a lower time complexity than W15.

graphSteepest[:INT] Changes the number or graphs simultaneously evaluated in a variety of procedures (e.g. swap, fuse, netmove) from the default (the larger of the number of parallel threads and INT). This option can be used to exploit or limit use of parallel thread numbers. Higher numbers will consume more memory and lower, less. The default value is 10.

multiTraverse:BOOL Controls the multi-traverse character optimization option. If `True`, character trees are traversed from each edge as in [36, 37, 34, 48], but individually for each dynamic character. This is the default and yields better (lower) optimality scores at the cost of added execution time. This is the default. If `False`, only outgroup-based traversals are performed as in [39, 13, 47].

outgroup:STRING Specifies the terminal to root the output trees. This name must appear in double quotes. If the outgroup is not set, the default outgroup is the taxon whose name is lexically first after any renaming of taxa, and/or if taxa were specified by using the arguments `include` or `exclude`. Only a single taxon can be set as the outgroup of the analysis.

softwiredMethod: Sets the algorithm for softwired graphs to the 'Naive' method of diagnosing all display trees, as in [43] or the 'Resolution Cache' method of [52].

Naive Determines the score of a softwired network by evaluating all display trees (up to 2^m for m network nodes) and summing the costs of the minimum cost display tree for each block of data. This can be extremely time consuming for graphs with large number of network nodes.

ResolutionCache Uses the method of [**?**] to determine softwired network costs in greatly reduced time. This is the default.

staticApprox Converts non-aligned ('dynamic') sequence characters to their implied alignment [41, 38] condition.

toHardwired Converts exiting graphs to hardwired network graphs and graphfactor to 'NoNetPenalty'.

toSoftwired Converts exiting graphs to softwired network graphs.

toTree Converts exiting graphs to trees. For both Softwired and Hardwired graphs this proceeds via graph resolution of network nodes into "display" trees. Since there are up to 2^n display

trees for a graph with n network nodes, this number can be quite large. The number of display trees produced for each graph is controlled via the options `displayTrees:n`, `atRandom`, and `first`.

usenetaddheuristic:Bool Employ a heuristic cost procedure for network addition that has lower time complexity, but is approximate as opposed to evaluating each network solution via full graph traversal.

Defaults

None.

Examples

- `transform(toSoftwired)`
  Converts each current graph to a softwired network graph.

- `transform(staticApprox)`
  Changes data to all static characters via Implied Alignment for further analysis.

Acknowledgments

The authors would like to thank DARPA SIMPLEX N66001-15-C-4039, the Robert J. Kleberg Jr. and Helen C. Kleberg foundation grant "Mechanistic Analyses of Pancreatic Cancer Evolution", and the American Museum of Natural History for financial support.

Bibliography

[1] E. N. Adams. Consensus techniques and the comparison of taxonomic trees. *Syst. Zool.*, 21:390–397, 1972.

[2] M. Baroni, C. Semple, and M. Steel. A framework for representing reticulate evolution. *Annals of Combinatorics*, 8(4):391–408, 2005.

[3] K. Bremer. Combinable component consensus. *Cladistics*, 6:369–372, 1990.

[4] K. Bremer. Branch support and tree stability. *Cladistics*, 10(3):295–304, 1994.

[5] J. H. Camin and R. R. Sokal. A method for deducing branching sequences in phylogeny. *Evolution*, 19:311–326, 1965.

[6] G. Cardona, F. Russelló, and G. Valiente. Extended newick: it is time for a standard representation of phylogenetic networks. *BMC Bioinformatics*, 9(532), 2008. doi: 10.1186/1471-2105-9-532.

[7] V. Cerny. A thermodynamical approach to the traveling salesman problem: an efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45:41–51, 1985.

[8] M. O. Dayhoff. Computer analysis of protein evolution. *Scientific American*, 221(1):86–95, 1969.

[9] J. S. Farris. A method for computing Wagner trees. *Systematic Zoology*, 19:83–92, 1970.

[10] J. S. Farris. Estimating phylogenetic trees from distance matrices. *American Naturalist*, 106:645–668, 1972.

[11] J. S. Farris. Hennig86, Program and Documentation, 1988.

[12] J. S. Farris, V. A. Albert, M. Källersjö, Lipscomb, and A. G. Kluge. Parsimony jackknifing outperforms neighbor-joining. *Cladistics*, 12(2):99–124, 1996.

[13] D. S. Gladstein and W. C. Wheeler. POY version 2.0. program and documentation available at http://research.amnh.org/scicomp/projects/poy.php. American Museum of Natural History, New York, 1997.

[14] P. Goloboff, J. S. Farris, and K. Nixon. TNT, a free program for phylogenetic analysis. *Cladistics*, 24:774–786, 2008.

[15] P. A. Goloboff. Analyzing large data sets in reasonable times: solutions for composite optima. *Cladistics*, 15(4):415–428, 1999.

[16] M. Goodman, C. B. Olson, J. E. Beeber, and J. Czelusniak. New perspectives in the molecular biological analysis of mammalian phylogeny. *Acta Zoologica Fennica*, 169:19–35, 1982.

[17] J. H. Holland, editor. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor Michigan, 1975.

[18] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[19] T. Margush and F. R. McMorris. Consensus n-trees. *Bull. Math. Biol.*, 43:239–244, 1981.

[20] N. A. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of state calculations by fast computing machine. *J. Chem. Phys*, 21(6):1087–1092, 1953.

[21] M. Miyagi and W. C. Wheeler. Reconciling trees using phylogenetic edge union networks. *Cladistics*, 35:688–694, 2019.

[22] A. Moilanen. Searching for most parsimonious trees with simulated evolutionary optimization. *Cladistics*, 15(1):39–50, 1999.

[23] A. Moilanen. Simulated evolutionary optimization and local search: Introduction and application to tree search. *Cladistics*, 17:S12–S25, 2001.

[24] B. M. E. Moret, J. Tang, and T. Warnow. Reconstructing phylogenties from gene-order and gene-content data. In O. Gascuel, editor, *Mathematics of Evolution and Phylogeny*, pages 321–352. Oxford, 2005.

[25] G. Nelson. Cladistic analysis and synthesis: Principles and definitions, with a historical note on Adanson's Familles des plantes (1763-1764). *Syst. Zool.*, 28:1–21, 1979.

[26] R. D. M. Page. TREEVIEW: An application to display phylogenetic trees on personal computers. *Computer Applications in the Biosciences*, 12:357–358, 1996.

[27] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *PNAS*, 85:2444–2448, 1988.

[28] D. F. Robinson. Comparison of labelled trees with valency three. *J. Combiatorial Theory*, 11:105–119, 1971.

[29] N. Saitou and M. Nei. The neighbor-joining method: A new method for reconstructing phylogenetic trees. *Mol. Biol. Evol.*, 4:406–425, 1987.

[30] R. T. Schuh and J. T. Polhemus. Analysis of taxonomic congruence among morphological, ecological, and biogeographic data sets for the Leptopodomorpha (Hemiptera). *Syst. Zool.*, 29:1–26, 1980.

[31] R. R. Sokal and C. D. Michener. A statistical method for evaluating systematic relationships. *University of Kansas Science Bulletin*, 38:1409–1438, 1958.

[32] D. L. Swofford. PAUP: Phylogenetic Analysis Using Parsimony, 1990. Note–Although the citation is 1990, I personally used version 2.4 as early as 1985.

[33] W. R. Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933.

[34] A. Varón, L. S. Vinh, I. Bomash, and W. C. Wheeler. Poy 4.0. American Museum of Natural History. http://research.amnh.org/scicomp/projects/poy.php, 2008.

[35] A. Varón, L. S. Vinh, and W. C. Wheeler. POY version 4: Phylogenetic analysis using dynamic homologies. *Cladistics*, 26:72–85, 2010.

[36] A. Varón and W. C. Wheeler. The tree-alignment problem. *BMC Bioinformatics*, 13:293, 2012.

[37] A. Varón and W. C. Wheeler. Heuristics for the general tree alignment problem. *BMC Bioinformatics*, 14:66, 2013.

[38] A. J. Washburn and W. C. Wheeler. Efficient implied alignment. *BMCBioinformatics*, 21:296, 2020.

[39] W. C. Wheeler. Optimization alignment: The end of multiple sequence alignment in phylogenetics? *Cladistics*, 12(1):1–9, 1996.

[40] W. C. Wheeler. Homology and the optimization of DNA sequence data. *Cladistics*, 17:S3–S11, 2001.

[41] W. C. Wheeler. Implied alignment. *Cladistics*, 19:261–268, 2003.

[42] W. C. Wheeler. *Systematics: A course of lectures*. Wiley-Blackwell, 2012.

[43] W. C. Wheeler. Phylogenetic network analysis as a parsimony optimization problem. *BMC Bioinformatics*, 16:296, 2015.

[44] W. C. Wheeler. Distance wagner tree refinement as a heuristic approach to character-based initial tree construction. *Cladistics*, n/a(n/a):1–9, 2021.

[45] W. C. Wheeler. Phylogenetic supergraphs. *Cladistics*, 38(1):147–158, 2022.

[46] W. C. Wheeler. The use of Thompson sampling for the optimization of phylogentic search strategies. *Unclear*, 0:0–0, 2023.

[47] W. C. Wheeler, D. S. Gladstein, and J. De Laet. POY version 3.0. program and documentation available at http://research.amnh.org/scicomp/projects/poy.php (current version 3.0.11). documentation by D. Janies and W. C. Wheeler. commandline documentation by J. De Laet and W. C. Wheeler. American Museum of Natural History, New York, 1996-2005.

[48] W. C. Wheeler, N. Lucaroni, L. Hong, L. M. Crowley, and A. Varón. POY version 5.0. American Museum of Natural History. http://research.amnh.org/scicomp/projects/poy.php, 2013.

[49] W. C. Wheeler, N. Lucaroni, L. Hong, L. M. Crowley, and A. Varón. POY version 5: Phylogenetic analysis using dynamic homologies under multiple optimality criteria. *Cladistics*, 31, 2015. 189-196.

[50] W. C. Wheeler and A. Varón. Phylogenetic minimum descriptive length: An optimality criterion for phylogenetic analysis. *Cladistics*, 2022. in prep.

[51] W. C. Wheeler and A. J. Washburn. FASTC: a file format for multi-character sequence data. *Cladistics*, 35(5):573–575, 2019.

[52] W. C. Wheeler and A. J. Washburn. Parsimony optimization of soft-wired phylogenetic networks. *In preparation*, 0:0–0, 2023.