POY 5.0

Program Documentation
Version 5.0.5.0

Program and Documentation
Andrés Varón
Nicholas Lucaroni
Lin Hong
Ward C. Wheeler

Documentation
Louise M. Crowley
Megan Cevasco
John S. S. Denton

AMERICAN MUSEUM OF NATURAL HISTORY
www.amnh.org

Previous Version POY 4

Program and Documentation
Andrés Varón
Le Sy Vinh
Illya Bomash
Ward C. Wheeler

Documentation
Ilya Tëmkin
Megan Cevasco
Kurt M. Pickett
Julián Faivovich
Taran Grant
William Leo Smith

Andrés Varón
Jane Street Capitol, 1 New York Plaza, New York, NY, U.S.A.

Louise M. Crowley, Lin Hong, Nicholas Lucaroni, Ward C. Wheeler
Division of Invertebrate Zoology, American Museum of Natural History, New York, NY, U.S.A.

John S. S. Denton
Richard Gilder Graduate School and Department of Ichthyology, American Museum of Natural History, New York, NY, U.S.A.

Megan Cevasco
Coastal Carolina University, Department of Biology, Conway, SC, U.S.A.

Illya Bomash
Department of Physiology and Biophysics, Weill Medical College of Cornell University, New York, NY, U.S.A.

Julián Faivovich
División Herpetología, Museo Argentino de Ciencias Naturales - CONICET, Buenos Aires, Argentina.

Taran Grant
Universidade de São Paulo, Instituto de Biociências, Departamento de Zoologia, Cidade Universitária, São Paulo, Brasil.

Kurt M. Pickett
Department of Biology, University of Vermont, Burlington, VT, U.S.A.

William Leo Smith
Department of Ecology and Evolutionary Biology, The University of Kansas, Lawrence, KS, U.S.A.

Ilya Tëmkin
Northern Virginia Community College, Annandale Campus, VA, U.S.A.

Le Sy Vinh
College of Technology and Information Technology Institute, Vietnam National University, Hanoi, Vietnam.

The American Museum of Natural History

Varón, A., N. Lucaroni, L. Hong, W. C. Wheeler. 2013. POY 5.0. New York, American Museum of Natural History. Documentation by L. M. Crowley, M. Cevasco, J. S. S. Denton. http://www.amnh.org/our-research/computational-sciences/research/projects/systematic-biology/poy

Available online at http://www.amnh.org/our-research/computational-sciences/research/projects/systematic-biology/poy and http://code.google.com/p/poy/

Comments or queries relating to the documentation should be sent to
crowley@amnh.org

Contents

Chapter 1

What is POY5

POY5 is a flexible, multi-platform program for the phylogenetic analysis of a diversity of data types under different optimality criteria — parsimony and likelihood. An essential feature of POY5 is that it implements the concept of dynamic homology [61, 62] allowing optimization of ***UNALIGNED*** sequences. POY5 offers flexibility for designing heuristic search strategies and implements a diversity of algorithms including multiple random addition sequence, swapping, tree fusing, tree drifting, and ratcheting. As output, POY5 generates a comprehensive character diagnosis, graphical representations of cladograms and their user-specified consensus, support values and implied alignments. In addition, POY5 can also output synteny block maps from the analysis of both chromosomal and genomic data. POY5 provides a unified approach to co-optimizing different types of data, such as morphological and molecular sequence data. In addition, POY5 can analyze entire chromosomes and genomes, taking into account large-scale genomic events (translocations, inversions, and duplications).

1.1 The structure of POY5 documentation

Chapter 2, *POY5 Quick Start*, will get you started using POY5. The first few sections are intended to provide detailed instructions on how to obtain and install POY5, introduce the user to two of the program's working environments, the *Graphical User Interface* and the *Interactive Console*. These sections also show how to initiate a POY5 session and point to various resources to obtain further assistance. Subsequent sections build on that knowledge and give step-by-step examples on how to conduct a basic analysis and visualize the results. The following chapter, *POY5 Commands*, describes POY5 commands

and their valid syntax. It also includes examples of simple operations for every command. Chapter 4 discusses the heuristic procedures used in POY5. Their understanding helps in the creation of efficient search strategies. More advanced operations are described in the fifth chapter, *POY5 Tutorials*.

1.2 What's new in POY5

There are myriad new features and options in POY5. These are described and documented in full in the pages that follow.

- New optimality criterion–likelihood:
  – Maximum Average Likelihood (MAL) analysis can now be performed on qualitative data of any alphabet size for aligned sequence data (including gaps as missing, independent, or coupled in 5-state models); and fixed states for unaligned sequences.
  – Most Parsimonious Likelihood (MPL) can also be employed on these data types as well as unaligned sequences under an MPL-DO heuristic.
  – Multiple models are available and different models can be assigned to partitions within a combined analysis.
  – Model selection (AIC, AICc and BIC) improved.

- The MAUVE genome aligner algorithm has been implemented as an annotation option for unannotated chromosomal and genomic (multi-chromosomal) data.

- The transform option `level` has been added to increase control and heuristic effectiveness for amino acid and custom alphabet sequence character types.

- Search-Based sequence optimization has been added through the `transform` command.

- Additional median solvers implemented for rearrangement analysis in `break_inv`, `chromosome`, and `genome` sequence characters.

- XML-based output for easy parsing of diagnostic information.

- A change in the default indel cost from 2 to 1. After over 20 years (MALIGN to POY), time for a change.

- New required packages for compilation to support likelihood and median solvers.

- Small syntax changes in the transform command, as well as new options, e.g. `level`.
- A diversity of bug fixes and smaller enhancements.

Chapter 2

POY5 Quick Start

2.1 Requirements: software and hardware

2.1.1 Software

POY5 is a platform-independent, open-source program that can be compiled for many operating systems and hardware configurations, including Mac OSX, Microsoft Windows and Linux. piece of software necessary to run POY5 *Graphical User Interface* of POY5 provides the functionality for running analyses using pull-down menus and field selections, as well as creating and running POY5 scripts. Some utility programs (such as Notepad and Ghostscript for Windows, TextEdit for Mac, or Nano for Linux), can help preparing POY5 scripts and formatting data files, while others (such as Adobe Acrobat) can facilitate viewing the graphical output in PDF (Portable Document Format).

2.1.2 Hardware

POY5 runs on a variety of computers from laptops and desktops to clusters of various sizes and symmetric multiprocessing hardware. There are no particular requirements for disk space, but XML and diagnosis report files can be large.

2.2 Obtaining and installing POY5

2.2.1 Installing from the binaries

POY5 installers for Windows and Mac OSX, source code, and documentation in PDF format are available from the POY5 download website at the

Computational Sciences site of the American Museum of Natural History:

http://www.amnh.org/our-research/computational-sciences/research/projects/systematic-biology/poy

The latest source code can also be obtained from POY5 Google Group website:

http://code.google.com/p/poy/source

The following detailed step-by-step instructions will guide you through downloading and installing POY5 *binaries* for various platforms.

Windows

- Download the *poy5* folder to the desktop by selecting the *Windows* download link.
- Open the *POY_version.zip*. You will need Administrator privileges to install the application. Extract the zip file to install in the desired location and execute the *poy.exe* file. If you have Windows XP SP2, Windows Vista or Windows 7 and possess more than one core or processor, you can take advantage of this processing power by installing the parallel components MPICH2.

  [Note: The POY5 developers encountered no problems when using MPICH2 3.0.2.]

Mac OSX

- Download *poy-buildXXXX.dmg* disk image to the desktop. The complete installation of the Mac OSX version of POY5 includes MPICH2 1.4.1, which is used to communicate processes during parallel execution.
- Drag the POY5 application from the disk *poy5* and drop it into the *Applications* folder on the hard drive.

  [Note: During the first execution in parallel you may be asked by the Firewall to unblock POY5 and MPICH. This is necessary for successful execution of the program.]

Linux

- No binaries are available for the Linux operating system. The user should compile POY5 directly from the source (see Section 2.2.2).

2.2.2 Compiling from the source

For the majority of users, downloading the binaries from the POY5 download site will suffice. However, in some cases it may be necessary: analysis of chromosomal and genomic data (>16383 large alphabets (>255 elements), orpreference for working in a command-line environment or running POY5 analyses in parallel (in the case of Linux machines or on a cluster computer), or necessary (in the case of Linux users) — to compile POY5 directly from the source code (see Table 2.1 and 2.2). If the user chooses to compile, it is advisable to check out the various configuration options that can be found in `./configure -help` of the `src` directory.

In order to compile POY5 the following tools are required:

1. The GNU Make 3.8 utility.
2. OCaml version 3.11.2. or later.
3. A **C** compiler, for example The GNU Compiler Collection.
4. The zlib compression library.
5. The Linear Algebra PACKage LAPACK must be installed in order to use the likelihood option.
6. The ncurses library is necessary to compile the default interface, i.e. *ncurses* or the *Interactive Console*. If this library is not available, the *flat* interface will be compiled instead.
7. The Message Passing Interface MPICH2, which is used to communicate processes during parallel execution.
8. The Concorde Package, which includes TSP median solvers used in the calculation of chromosome and genome rearrangements.

Download, ungzip and untar the POY5 source code. In a terminal window, change directories to the path of this uncompressed directory. In order to compile under default setting, working in the source directory (`src`), type:

```
./configure
make
make install
```

To `make install`, it may be necessary to do this as the root user using `sudo`. This script will compile the *Interactive Console* or *ncurses* interface that will be found in `/src/_build`. Another configuration option includes a *readline* interface. Similar to the *ncurses* interface, this allows for the use of arrow keys to modify commands and browse command history. is available. A *flat* interface is also available that supports the running of the program in parallel, irrespective of the operating system. This version is run through a terminal window and invoked in a script (see Section 2.3). In order to run `POY5` in parallel environments, Message Passing Interface (invoked by `mpiexec` or `mpirun`, depending on your implementation) must be invoked. More than likely, your system administrator already has one installed and should be able to provide you with the proper paths to set your config file. In order to compile this *flat* version with parallel support type:

```
./configure --enable-interface=flat --enable-mpi CC=mpicc
make
make install
```

[Note: CC=mpicc is not available for the Windows version of mpi, therefore it is not necessary to include this component in the compiling script.]

Table 2.1 should be used as a guide as to the type of interface that should be employed depending on the type of data ('standard' or 'long').

2.3 Executing a Script

A number of startup options are available when executing a script through the command line in a terminal window. The options below, can be viewed by typing `poy -help` in a terminal window.

`-w` Run poy in the specified working directory
`-e` Exit upon error
`-d` Dump filename in case of error
`-q` Don't wait for input other than the program argument script
`-no-output-xml` Do not generate the output.xml file
`-plugin` Load the selected plugins
`-script` Inlined script to be included in the analysis
`-help` Display this list of options

The use of these options are appropriate any time the user chooses to execute `POY5` from the command line, when working with *ncurses* and *flat*

Table 2.1: Interface Guide. 'Standard' data equates to a molecular sequence or single partition that is fewer than 16383 nucleotides in length or contain fewer than 255 elements, while 'long' data partitions accommodate lengths greater than these values. The field 'config+' indicates that the options long-sequences and/or large-alphabets must be enabled during compilation for these datatypes to be analyzed. A distinction is made between the *Interactive Console* or *ncurses* that is downloaded as binaries (`bins`) from the website and that which is compiled from the source (`src`). [Note: It is not possible to analyze either long sequences or large alphabets using the *GUI* or the *Interactive Console* when downloaded from the `POY5` website.]

Data type	GUI	Interactive Console (bins)	Interactive Console (src)	Flat
Standard	+	+	+	+
Long	–	–	config+	config+

interfaces. These options are also useful when operating `POY5` in a cluster environment. For example typing

```
poy -w /Users/username/poyfiles mol.poy
```

in a terminal window will invoke the program POY to run the script `mol.poy` in the directory `/Users/username/poyfiles`. This is the equivalent of including

```
cd ("/Users/username/poyfiles")
```

When attempting to run `POY5` in parallel from the command line, the programs MPI or Mpiexec must be used to initialize the parallel job from within a PBS batch or interactive environment. For example typing

```
mpirun -np 8 ~/POY_bins/poy5a-mpi ~/POY_analyses/mol.poy
```

in the terminal window will invoke a parallel version of `POY5` to run the script `mol.poy`, utilizing 8 processors (Figure 2.1). The file `mol.poy` resides in the directory `POY_analyses`, which is found in the home directory. This parallel

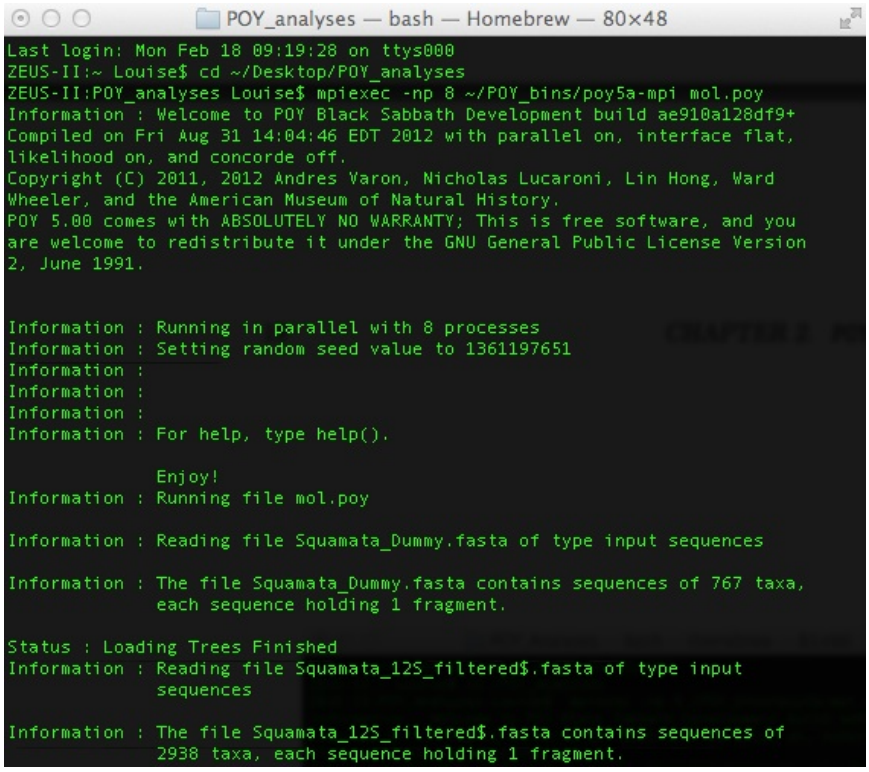

Figure 2.1: POY5 flat interface displayed in a terminal window. The interface indicates that the program has been compiled with 'parallel on'. The program is running the script `mol.poy` in parallel over 8 processors. In this case, MPICH is used to communicate processes during parallel execution.

version of POY5 was compiled in the directory `POY_bins` in the home directory. With the flat interface, it is not possible to run parallel jobs interactively, therefore a script ***must*** be included in the command, so that it can be passed to the application.

2.4 The Graphical User Interface

Two of the working environments that POY5 provides are the *Graphical User Interface* and the *Interactive Console* (also known as the *ncurses* interface). The *Graphical User Interface* has a user-friendly appearance like other native stand-alone applications where different functions are accessible through

menus and windows. Thus, the entire analysis can be carried out by clicking on appropriate selections and, where necessary, typing specifications in designated fields. Currently, the *Graphical User Interface* is designed for the analysis of data with parsimony or likelihood (with minimal model selection) as the optimality criterion. Unlike the *Interactive Console*, it is not possible to specify all options with the *Graphical User Interface*. The minimum screen size for the *Graphical User Interface* is 1024 x 768 pixels.

On the other hand, the *Interactive Console* requires a detailed knowledge of `POY5` commands, their arguments, and the conventions of `POY5` scripting. All these features are described in the *POY Commands* chapter (3.1.1).

Even though the Mac OSX version of the *Graphical User Interface* is used for screen shots throughout this chapter, the Windows version contains the same items and functionality, differing only in the generic window format specific to the platform.

When `POY5` is first opened, two items appear on the screen: the `POY5` menu bar across the top and the *POY Launcher* window (Figure 2.2).

[Note: In Windows the menu bar is within the launcher window.]

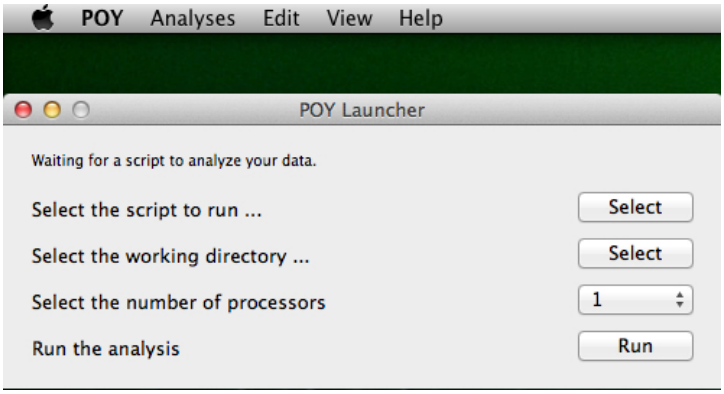

Figure 2.2: The `POY5` menu bar and the *POY Launcher* window. These items appear when `POY5` is opened.

2.4.1 POY menu bar

The menu bar contains the following drop-down menus:

POY (Mac OSX only) Contains generic items as with other Mac OSX applications. This pull down menu allows selection of the *About POY*

window (Figure 2.3) that lists the current version of POY5, a copyright statement, and the address of the POY5 website. In addition, it includes a *Quit POY* tab that closes the program.

Analyses Contains options for different types of tree searches, calculation of support values, tree diagnosis, and their respective outputs. Other items in this menu open the *POY Launcher* (Open Launcher) and the *Interactive Console*.

Edit Contains standard tools for undoing, cutting, copying, pasting, deleting, and selecting.

View opens the *Output* window to display the results (including warning and error messages) and the current state of the analysis. This *Output* window also contains an *update* tab. It also contains the *About POY* menu item in Windows.

Help Opens the POY5 *Manual* in PDF format (requires a PDF viewer).

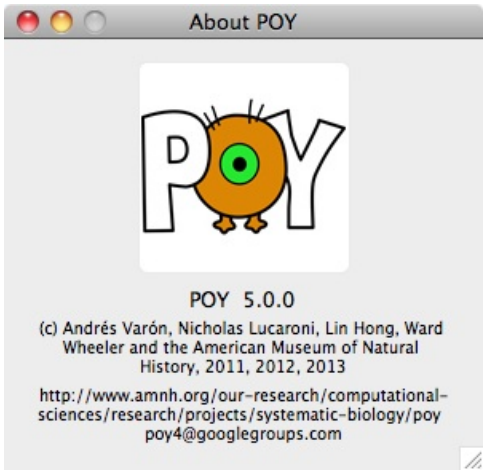

Figure 2.3: The *About POY* window.

2.4.2 POY Launcher

The *POY Launcher* is the only window that automatically opens upon starting POY5. This allows the user to import a previously created script,

Table 2.2: Parallelization Guide. The field 'mpi+' indicates that mpi must be enabled during compiling. A distinction is made between the *Interactive Console* or *ncurses* that is downloaded as binaries (`bins`) from the website and that which is compiled from the source (`src`).

Operating System	GUI	Interactive Console (bins)	Interactive Console (src)	Flat
Mac OSX	+	–	–	mpi+
Windows	+	–	–	mpi+
Linux	N/A	–	–	mpi+

designate a working directory, specify the number of processors, and start the analysis.

Select the script to run Allows the user to specify the location of a `POY5` script.

Select the working directory The working directory is the directory that contains the input data and output files. By default, the working directory is set to be the same as the directory containing the selected `POY5` script.

Select the number of processors If more than one processor or core is available, up to 8 can be designated for running the analysis. It is important to note that once specified, the selection is applied to *all* subsequent analyses in the current `POY5` session. Table 2.2 is a guide to the parallelization ability of `POY5` depending on the operating system and the `POY5` interface being used. Observe that parallelization is ***never*** supported in interactive sessions, see Section 2.5.

Run the analysis Clicking the *Run* button starts the execution of the selected script. Once the script is initiated, the *Run* button becomes the *Cancel* button that can be used to interrupt a `POY5` session.

If the *Run* button is clicked without the selected script and working directory, or the names of the scripts and working directory are entered incorrectly, `POY5` issues an error message in the upper part of the *POY Launcher* window, such as `POY finished with an error`.

2.4.3 The *Analyses* menu

The *Analyses* menu is the main toolbox of the POY5 *GUI* interface (Figure 2.4, left). Selections are subdivided into four functional categories. The first three deal with tree searching, support calculation, and tree diagnosis; the fourth one is used for script management or interactive command execution that bypasses the menu-driven script generation. Each of the menu items is described below in the order it appears on the menu.

Most options are consistently applied through different kinds of analysis. Therefore, all options are described in detail only for the *Simple Search* analysis. The descriptions of other analyses are made with reference to the the *Simple Search* and focus on unique options.

Tree searching options

A number of different tree searching options are available through the *Graphical User Interface*. These include a *Simple Search*, *Timed Search*, *Search with Ratchet* and *Search with Perturb*.

Simple Search

The *Simple Search* window permits the analysis of a number of different data types, including a range of molecular characters (from DNA sequences to complete genomes), custom alphabet characters, and qualitative characters, under parsimony. It is also possible to carry out a likelihood analysis of DNA sequence, morphological and qualitative data under a number of different models. In the simplest sense, a typical search involves a series of steps. First, initial trees are generated by random addition sequence from the imported character data. These trees are then subjected to branch swapping, after which trees are selected to report. The *Simple Search* window (Figure 2.4, right) provides the most common and basic options for a standard tree search in POY5 that must be selected by either clicking the appropriate buttons or by typing. Note that *all* the empty fields must be filled in (even if that means assigning a cost of 0 to all the *Sequence Parameters*), otherwise the default values will be used. The window is subdivided into five sections:

Input Files Contains the list of files that are to be input into POY5. These include character files in nucleotide, Hennig86, and Nexus formats, as well as tree files. Continuous characters can be input into POY5 in a Hennig86 format matrix (see `read` (Section 3.3.14)). Character data in

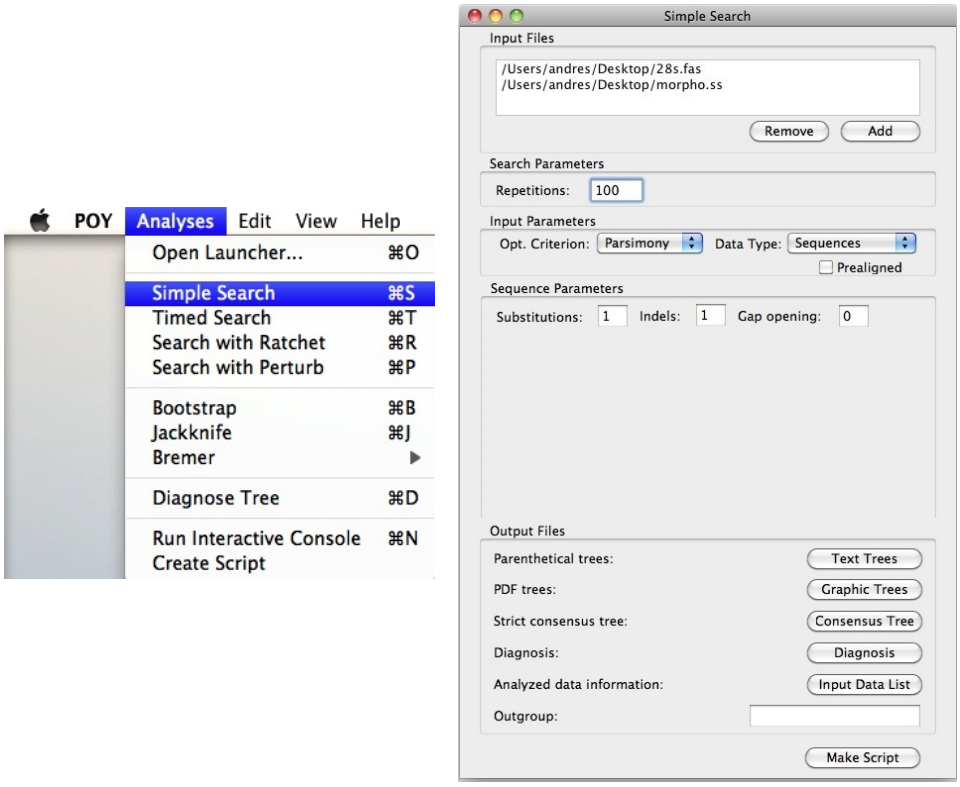

Figure 2.4: The *Simple Search* window. Selecting *Simple Search* from the *Analysis* menu (left) opens the *Simple Search* window options (right).

other formats can be input by specifying additional arguments in the script (see `read` (Section 3.3.14)).

NOTE

Gap-opening cost greater than 0 can not be specified with prealigned data as the columns are treated as independent in this file format.

Search Parameters Holds one field to set the number of independent random addition Wagner replicates to be generated.

Input Parameters Holds fields to specify the optimality criterion (parsimony or likelihood). With *Parsimony* as the optimality criterion, it is possible to select different datatypes (sequence, chromosome, genome, custom alphabet, break inversion or qualitative) and allows the user to select whether these data should be treated as prealigned (if possible). Selection of different datatypes will invoke an additional subsection (see below). Currently, it is only possible to analysis sequence and qualitative data with the maximum likelihood criterion.

Parsimony Optimality Criterion The parameters in this section are dependent on the data types selected in *Input Parameters*. More detailed explanations of the different data types can be found below and in the difference character types sections of both `read` (Section 3.3.14) and `transform` (Section 3.3.26).

Sequence Parameters If *sequence* data types are chosen, the user can specify the substitution, indel, and gap opening costs of sequences. Enter `0` if no gap opening cost is desired. If the value of a parameter is not specified, default values are used. The default value for both substitutions and insertion deletion events is `1` and that of gap-opening is `0`.

Chromosome and Mauve Parameters *Chromosome* characters are multi-locus nucleotide sequences and can include nuclear chromosomes, as well as, mitochondrial and viral genomes. It is possible to submit either `annotated` (by selection of the *Annotated* box) or *Unannotated* (by leaving the *Annotated* box unchecked) chromosomes. Within *Annotated* chromosomes, homologous regions, such as loci, are separated with the pipe symbol ("|"). *Unannotated* chromosomes are entirely without

delimiters. For *Unannotated* chromosomes, the Mauve Parameters, must be set by the user. These parameters (*Match Quality, Match Coverage, Min. Match* and *Max. Match*) are employed by the Mauve aligner to find regions of homology or synteny blocks between chromosomes (see `annotate` (Section 3.3.27) within the command `transform`). Default values for these parameters are provided in the *GUI*.

Within this subsection it is necessary to specify both *Locus Indel* and rearrangement (*Locus Breakpoint* or *Locus Inversion*) costs. The cost of a *Locus Indel* is by default set to 10 plus 0.9 times the length of the locus (see `locus_indel` (Section 3.3.27) within the command `transform`). Rearrangements of homologous regions—as defined by the user in the case of *Annotated* chromosomes or as determined by the Mauve aligner as in *Unannotated* chromosomes)—are then optimized using either *Locus Breakpoint* or *Locus Inversion* costs (see `locus_breakpoint` (Section 3.3.27) and `locus_inversion` (Section 3.3.27) within the command `transform`). The default cost for both is set to 10. The user must also specify the *Median solver* for the optimization of rearrangements of *Annotated* chromosomes. The default median solver is *Caprara*, but the user can alternatively choose *BBTSP, ChainedLK, COALESTSP, MGR, Siepel, SimpleLK* and *Vinh* (see `median_solver` (Section 3.3.27) within the command `transform`).

The user must also specify whether the chromosome is *Circular* (true) or linear (false). It is not possible to submit pre-aligned data files for either *Annotated* or *Unannotated* chromosomes.

NOTE

Locus definition. In the *Sequence Parameters* section for a parsimony analysis, the user may be required to specify the cost associated with a *Locus Breakpoint*, *Locus Inversion* or *Locus Indel*, depending on the data type. In these cases, *Locus* should **not** be taken to be the functional, biological unit in the classical sense, but only as a homologous segment of a sequence.

Genome and Mauve Parameters *Genome* characters are multi-locus, multi-chromosomal nucleotide sequences, wherein transformations (i.e. indels, substitutions, and rearrangements) are optimized at the sequence, locus *and* chromosomal level. Within the genome data file, individual chromosomes are separated by the at symbol ("@") and the individual chromosomes remain *Unannotated*.

As with *Unannotated Chromosome* characters, homologous regions are determined using the Mauve parameters (*Match Quality, Match Coverage, Min. Match* and *Max. Match*). Default values for these parameters are provided in the *GUI*. The *Locus Indel* and rearrangement (*Locus Breakpoint* or *Locus Inversion*) costs are set by the user. By default, the cost of a *Locus Indel* is set to 10 plus 0.9 times the length of the locus (see the argument `locus_indel` (Section 3.3.27) within the *Chromosome and genome transformation methods* of the command `transform`). Rearrangements of homologous regions, as determined by the Mauve aligner, are then optimized using either *Locus Breakpoint* or *Locus Inversion* costs (see the arguments `locus_breakpoint` (Section 3.3.27) and `locus_inversion` (Section 3.3.27) within the *Chromosome and genome transformation methods* of the command `transform`). The default cost for both is set to 10. A *Median solver* must also be specified for the optimization of rearrangements: *BBTSP, ChainedLK, COALESTSP, MGR, Siepel, SimpleLK* and *Vinh* (see the argument `median_solver` (Section 3.3.27) within the *Chromosome and genome transformation methods* of the command `transform`).

Two other costs must be set for the analysis of this data type– *Translocation* events and *Chromosome Indel*. The cost of the *Translocation* of a region of one chromosome to another chromosome is set to 10 by default. The cost of the insertion or deletion of an entire chromosome is by default set to 10 plus 0.9 times the length of the chromosome.

As with *Chromosome* characters, it is not possible to input prealigned data files.

Custom Alphabet Parameters *Custom Alphabet* characters are those that employ a user-specified alphabet. With this data type, only insertion-deletion and substitution events are allowed. *Custom Alphabet* characters can be input as prealigned. Within this subsection, the user must specify the heuristic *Level* of the median sequence calculation. *Direct Optimization* is employed in median sequence calculation. Because calculating the median states between custom alphabet strings becomes more computationally intensive (and time consuming) as the number of elements in the alphabet increases, the user should select a heuristic level of median calculation appropriate for their data. The default level is 2.

In addition to the data file, the user is require to upload a *Cost Matrix* that specifies the substitution and indel transformation costs for alphabet elements. By selecting the *Cost Matrix* button within this subsection, the user can upload a cost matrix that specifies these costs for their custom alphabet data. For details on the format requirements for custom alphabet data files and their associated cost matrices see the argument `custom_alphabet` (Section 3.3.14) within the command `read`.

Break Inversion Parameters *Break Inversion* characters are an enhancement of *Custom Alphabet* characters. In addition to allowing substitution and insertion deletion events, element rearrangements, as well as orientation information can also be optimized. The median solvers provided restrict the analysis of prealigned data. The rearrangement costs for *Break Inversion* characters can be optimized using either *Breakpoint* or *Locus Inversion* approaches (see the arguments `locus_breakpoint` (Section 3.3.27) and `locus_inversion` (Section 3.3.27) within the *Chromosome and genome transformation methods* of the command `transform`). The default cost for both is set to 10. A *Median solver* must also be specified for the optimization of rearrangements. The default median solver is *Caprara* [8], but the user can alternatively choose *BBTSP, ChainedLK, COALESTSP, MGR* [5], *Siepel* [45], *SimpleLK* and *Vinh* (the TSP solvers BBTSP, CoalesTSP, ChainedLK and SimpleLK taken from the Concorde package) (see `median_solver` (Section 3.3.27) within the command `transform`). The calculation of median states between *Break Inversion* strings becomes more computationally intensive (and time consuming) as the number of elements in the alphabet increases, therefore a single heuristic level of median calculation can only be employed for these character types–the default level is 1.

The requirements for *Break Inversion* character types are identical to those for *Custom Alphabet* characters, with respect to substitution and indel transformation costs. By selecting the *Cost Matrix* button within this subsection, the user can upload a cost matrix to specify these costs. The user should see the argument `custom_alphabet` (Section 3.3.14) within the command `read` for details on the format requirements for these cost matrices, which are identical in form to those for *Custom Alphabet* characters.

Qualitative Parameters *Qualitative* data are any non-sequence, pre-

aligned data type (e.g. morphology, behavior). These character types are optimized as additive, non-additive or Sankoff characters and this information must be included in the data file when using the *Graphical User Interface*.

Likelihood Optimality Criterion Currently, it is only possible to perform a likelihood analysis of DNA sequences (prealigned being permitted), types. Using the *GUI*, these data can be analyzed with `likelihood` only (currently, transformation of the data to `elikelihood` cannot be performed using the *GUI*). More detailed explanations of these options can be found below and in the *Likelihood transformation methods* section of `transform` (Section 3.3.26). In this section the user can specify the *likelihood model* of character substitution under which the analysis will be performed. Available substitution models include JC69/Neyman, F81, K2P/K80, F84, HKY85, TN93, GTR, and NCM. Users can also perform phylogenetic model selection using AIC, AICc and BIC. Within this section it is possible to specify the nature of among-site variation under *Rate Distribution*. Rate variation distributions allow multipliers to be applied to separate groups of characters. These distributions can be set to *Constant*, *Gamma* (for non-zero rate variation) or *Theta* (for parameterization of invariant sites). These distribution values can be specified for all of the available models. In addition, *Rate Classes* enables the user to specify the number of rate classes for the discrete *Gamma* rate distribution. The user can choose up to 8 rate classes for either *Gamma* or *Theta* models (the default is 4). *Gap Treatment* specifies the treatment of indels. There are three options *missing*, *character*, and *character plus coupled*. When gaps are treated as *missing*, they play no role in the calculation of tree likelihoods. The *character* option treats the insertion and deletion of A, C, G, and T each as different types of events that are independently estimated (hence additional parameters over *coupled*). When *coupled* is specified, all indel events are treated with the same rate parameter. *Cost* specifies the form of likelihood employed. The options for prealigned data (sequence and qualitative) are *MAL* for *maximum average likelihood* and MPL for *most parsimonious likelihood* [3]. Unaligned sequences can be optimized with both *MAL* (by transforming the sequences to `fixed_states` first) and *MPL*. The prior probabilities of the states are determined using *Priors*. The options are *equal*, where each state prior is set to 1 divided by the number of states, and *estimated*, where the priors are set by their frequency in the data set. Priors for gaps are estimated by the

maximum difference in length between input sequences.

Output Files Designates the names and locations of files containing results of the analysis. By default, all of these output options are generated with default names applied. The names can then be changed in the generated script or the option can be removed entirely. As implied by their respective titles, the *tree* buttons output trees in both parenthetical (best and consensus trees) and postscript form (although this button only outputs a PDF file of the optimal trees found, a useful commands that the user can include in the generated script is to output a PDF file of the consensus tree (see the argument `graphconsensus` (Section 3.3.19) within the command `report`)).

A *diagnosis* file provides information relating to the analysis. Information in this file includes the cost of the tree, the rearrangement costs (in the case of the analysis of chromosome, genome and break inversion data types), as well as information about each resulting node in the tree. At each node, the user is provided with a cost of the tree down to that node, a rearrangement cost (if applicable), the "descendant nodes" coming from this node and information concerning individual characters at these nodes.

NOTE

The `root` in the diagnosis file may not be the same as the root `set` by the user. This is because the tree length heuristics may be based on an alternate rooting scheme than that used for the `newick` or `graphic trees` output.

The *Analyzed data information* outputs a summary of the input data. Specifically, the number of terminals to be analyzed, a list of included terminals with numerical identifications, list of synonyms (if specified), a list of excluded terminals, the number of included characters in each character-type category (additive, non-additive, Sankoff and sequence) with the corresponding cost regimes, a list of excluded characters, and a list of input files.

The *Outgroup* field allows the the user to specify the outgroup taxon. The name of the taxon should reflect the name as interpreted by `POY5`. Therefore, the name should take into account synonymy files, and taxon names that contain commented out information via use of a $ sign (see the argument `rename` (Section 3.3.18)).

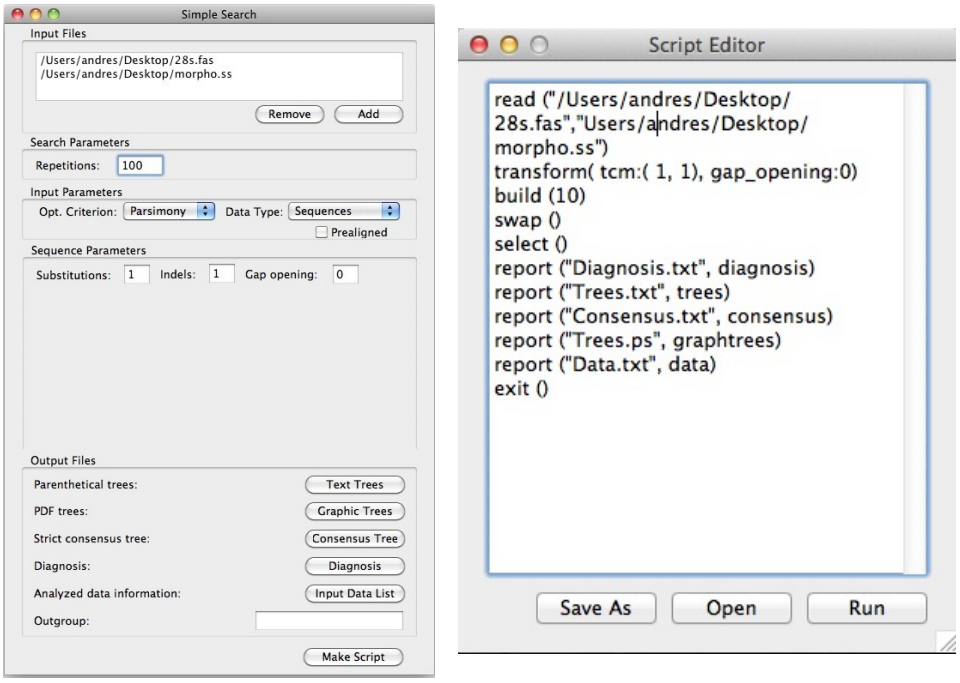

Figure 2.5: The *Simple Search* window with specified search parameters (left) and the corresponding *Script Editor* window. Observe that the names of the output files are left as the default output names.

Once all the parameters are selected, click the *Make Script* button and another window–the *Script Editor*–containing the generated script, appears on screen (Figure 2.5). The script can be edited by typing in the commands directly in the *Script Editor* window, saved (by clicking the *Save As* button), or replaced with another script (using the *Open* button). To start the analysis, click the *Run* button in the *Script Editor* window. When the *Run* button is clicked, `POY5` will issue a request to save the script. Thus, not only does `POY5` execute the script but it also creates the record of the type of analysis (including all user-defined specifications) that was performed. Moreover, these scripts can later be executed manually in the *ncurses* or *flat* interfaces, or selected as a script to run in the *Graphical User Interface*.

Timed Search

Timed Search (Figure 2.6) implements a default search strategy that effectively combines tree building with TBR branch swapping, parsimony ratchet, and tree fusing. The *Timed Search* window has the same four parameter groups described for the *Simple Search*. However, the *Search Parameters* section (called *Search and Perturb Parameters*) contains four fields specifying the search targets instead of the *Repetitions* field. These include the following:

Maximum time The maximum total execution time for the search. The time is specified as days:hours:minutes.

Minimum time The minimum total execution time for the search. The time is specified as days:hours:minutes.

Maximum memory The maximum amount of memory allocated for the search.

Minimum hits The minimum number of times that the minimum cost must be reached before terminating the search.

This heuristic search is a powerful tool for analyzing data. The number of rounds of successive searching is limited only by the previously specified search targets. Therefore, when performing a *Timed Search*, it is **crucial** to set the maximum time such that the program has a reasonable amount of time to perform a search. Thus, it is important to have some approximation as to the length of time it would take to perform a single round of searching (e.g. build (1), followed by TBR, ratchet and fusing in the case of a parsimony analysis of DNA sequence data). Clearly, this is data and optimality criterion dependent. With this information, the user can then estimate the amount of

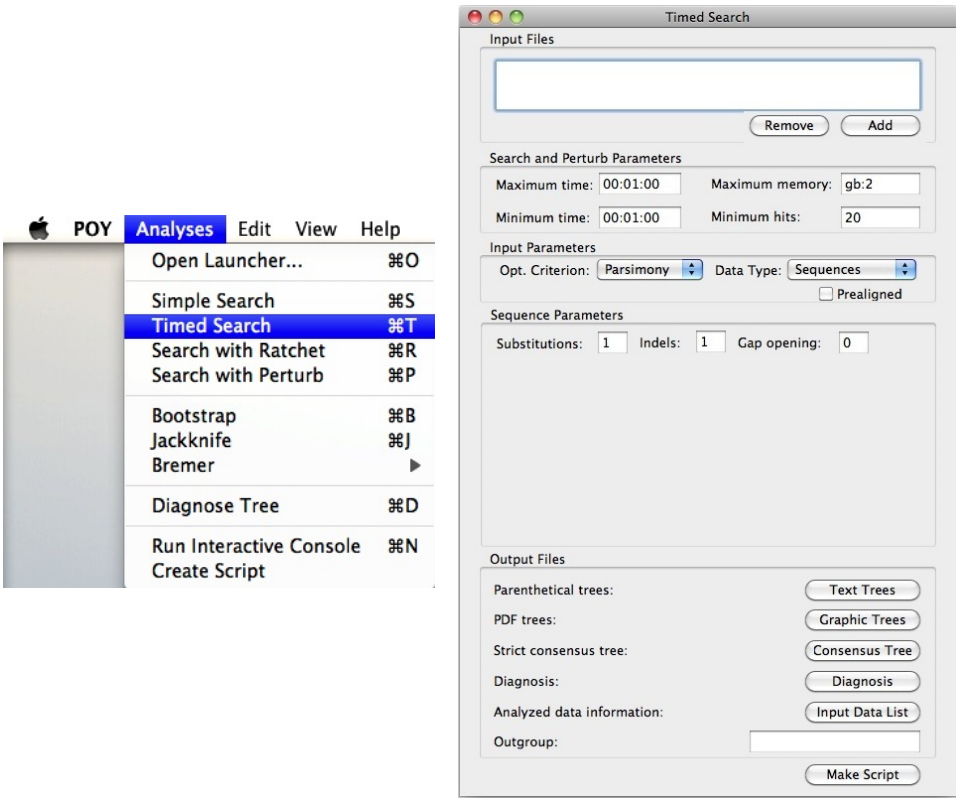

Figure 2.6: The *Timed Search* window. Selecting *Timed Search* from the *Analysis* menu (left) and viewing the *Timed Search* window options (right).

time necessary to perform a thorough search (perhaps 10 times the amount of time it took to perform this single round of build, swap, ratchet and fusing). The user should also allow some time for the program to collate and write the results to files. If the user has opted to run this analysis in parallel, this can take some time.

Search with Ratchet

The parsimony ratchet is a heuristic strategy to escape local optima during tree searching [36]. The ratchet reweights a given percentage of characters for a specified number of iterations of a search. An analysis is then performed and the resulting tree topology is evaluated using the original data matrix with all characters (with original weights) to determine the length of the tree. The *Search with Ratchet* (Figure 2.7) follows the same basic steps of a simple search but includes the ratchet step after the swap. In addition to the same sequence alignment and search parameters as described for the *Simple Search* window, the *Search Parameters* section provides the following ratchet parameters fields:

Ratchet iterations The number of iterations for the parsimony ratchet.

Severity The severity parameter of the ratchet (the weight change factor for the selected characters).

Percentage The percentage of characters to be reweighted during ratcheting.

Search with Perturb

Search with Perturb (Figure 2.8) provides an alternative means to escape local optima by changing the transformation cost matrix of the sequence characters, a procedure similar in spirit to the parsimony ratchet. In addition to the same sequence optimization and search parameters as described for the *Simple Search* window, the *Search with Perturb* window provides three extra fields with the parameters for the transformation cost matrix perturbation as follows:

Perturb iterations Sets the number of perturb iterations to be performed.

Substitutions Specifies the cost of the perturbed substitutions.

Indels Specifies the cost of the perturbed indels.

During this heuristic search, `POY5` performs a parsimony ratchet search during each iteration (default values are used, i.e. 25% probability, 2 severity).

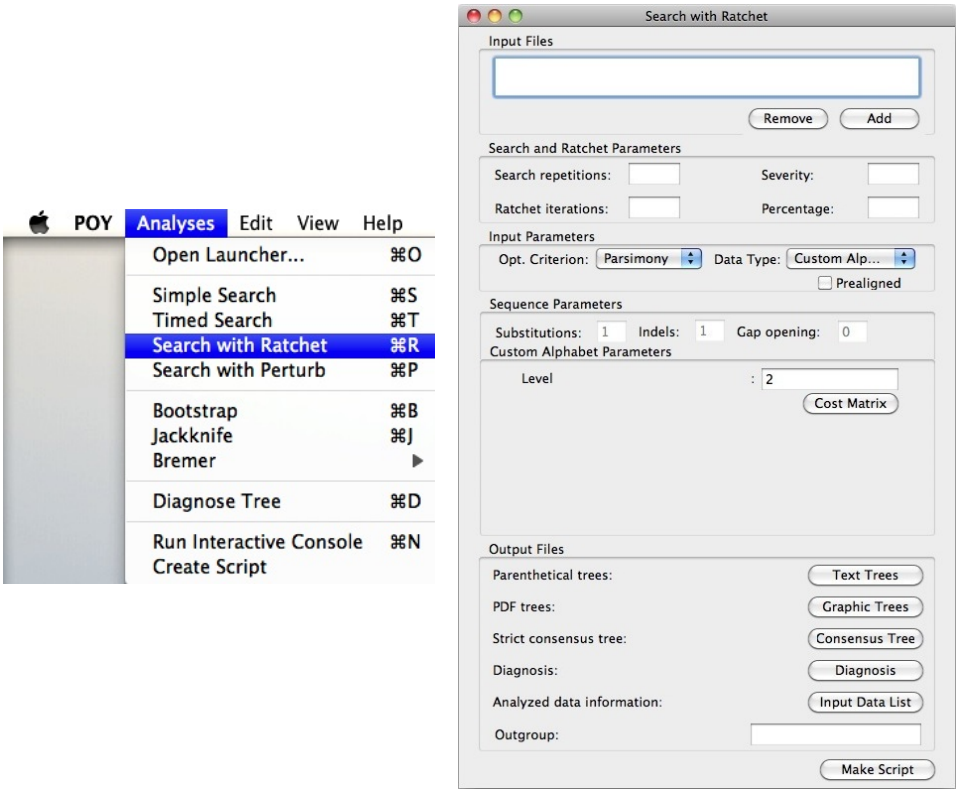

Figure 2.7: The *Search with Ratchet* window. Selecting *Search with Ratchet* from the *Analysis* menu (left) and viewing the *Search with Ratchet* window options (right).

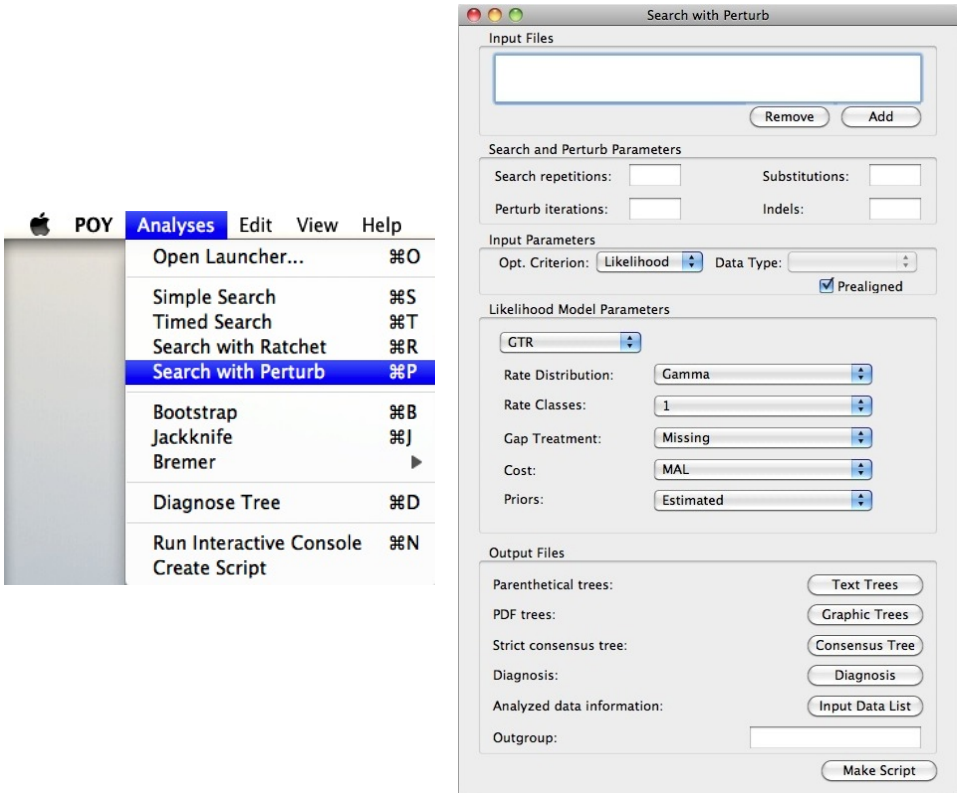

Figure 2.8: The *Search with Perturb* window. Selecting *Search with Perturb* from the *Analysis* menu (left) and viewing the *Search with Perturb* window options (right).

Support calculation options

It is possible to calculate several support values using this interface. Two of these measures, Bootstrap and Jackknife, involve resampling techniques, while the third, Bremer support, is an optimality-based measure based on the cost of the tree.

Although it is possible to calculate Jackknife and Bootstrap support values for trees constructed using dynamic homology characters, it is recommended against doing so as resampling of dynamic characters occurs at the fragment, rather than nucleotide, level. Consequently, the bootstrap and jackknife support values calculated for dynamic characters are not directly comparable to those calculated based on static character matrices. If it is desired to perform character sampling at the level of individual nucleotides, the dynamic characters must be transformed into static characters using `static_approx` argument of the command `transform` (Section 3.3.26) prior to executing `calculate_support`. Of course, if the dataset of dynamic characters contains a large number of fragments, this caveat may not be warranted.

For chromosome and genome character types, only the calculation of Bremer support values is recommended.

None of the support calculation windows include functions for tree building and searching. Therefore, one of the input files must contain trees for which support values are going to be calculated.

Bootstrap

As a resampling technique, the non-parametric *Bootstrap* resamples the original data (with replacement), creating a simulated dataset equal to the size of the original dataset. The *Bootstrap* window (Figure 2.9) specifies parameters for estimating the Bootstrap support values. In addition to the *Simple Search* window fields, it contains a field for the bootstrap parameters, in this case a *Pseudoreplicates* field, to specify the number of bootstrap pseudoreplicates.

Pseudoreplicates Specifies the number of resampling iterations.

Jackknife

An alternative statistical measure of support is the *Jackknife*, wherein the original data matrix is resampled, but in this case without replacement. The *Jackknife* window (Figure 2.10) specifies parameters for estimating the

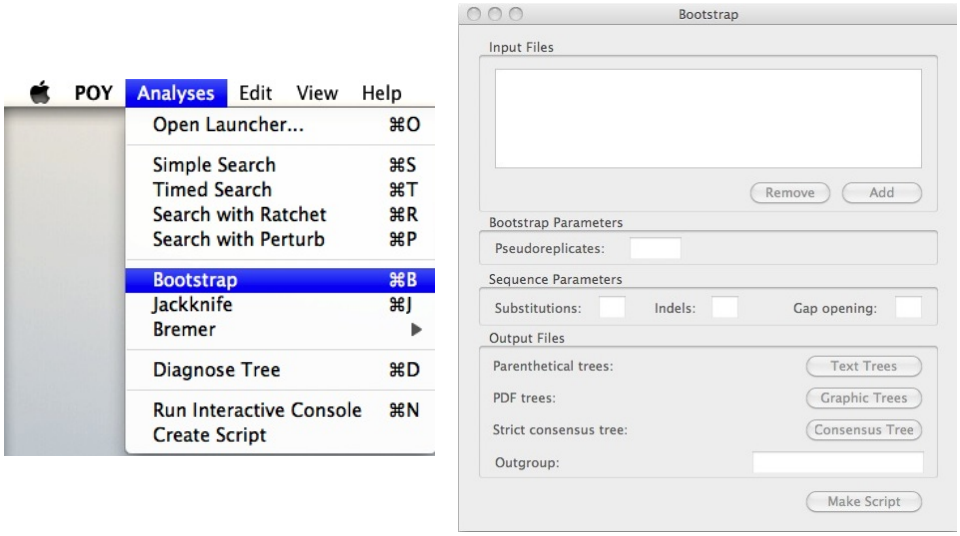

Figure 2.9: The *Bootstrap* window. Selecting *Bootstrap* from the *Analysis* menu (left) and viewing the *Bootstrap* window options (right).

Jackknife support values. In addition to the *Simple Search* window fields, *Jackknife Parameters* contains fields to specify the number of Jackknife pseudoreplicates (*Pseudoreplicates*) and the number of characters to be removed (*Remove*) during each pseudoreplicate.

Pseudoreplicates Specifies the number of resampling iterations.

Remove Specifies the percentage of characters being deleted during a pseudoreplicate.

Bremer

As an optimality-based measure of calculating tree support, Bremer values are the number of extra steps required before a clade is lost in the most parsimonious or strict consensus of the most parsimonious trees. Bremer support under likelihood is equivalent to the log of the likelihood ratios for each branch [67]. There are two ways to determine Bremer support values in `POY5`. The first involves performing a series of searches, where each group supported on the examined cladogram is constrained not to occur in the result. The second does not involve constraining the tree but collects information for all the clades not present in the set visited trees. Currently, calculating support via "visited" trees can only be performed sequentially and not parallel.

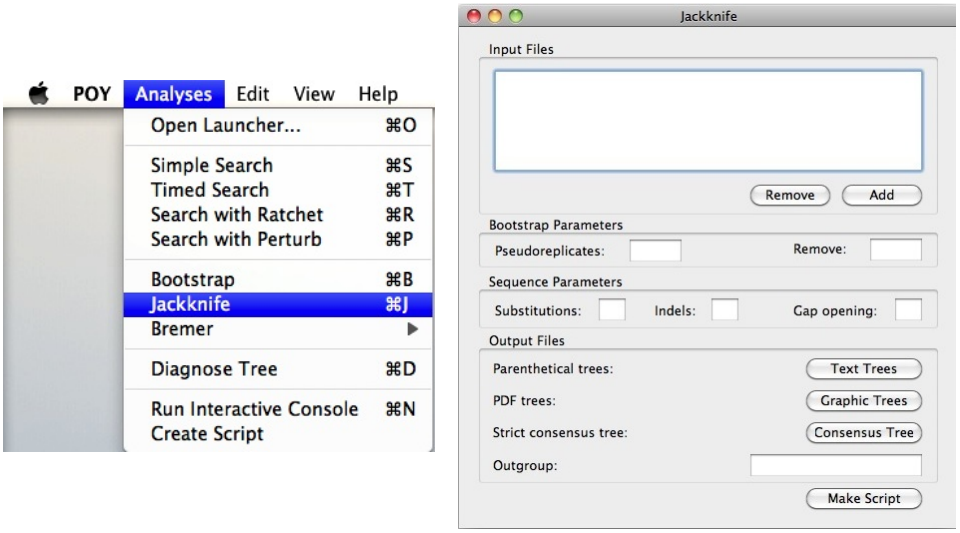

Figure 2.10: The *Jackknife* window. Selecting *Jackknife* from the *Analysis* menu (left) and viewing the *Jackknife* window options (right).

The *Bremer* option (Figure 2.11) is divided into two windows: the *Search for Bremer* window, that specifies the Bremer support [6, 28] calculation parameters, and the *Report Bremer* window to format the output of the results (Figure 2.12).

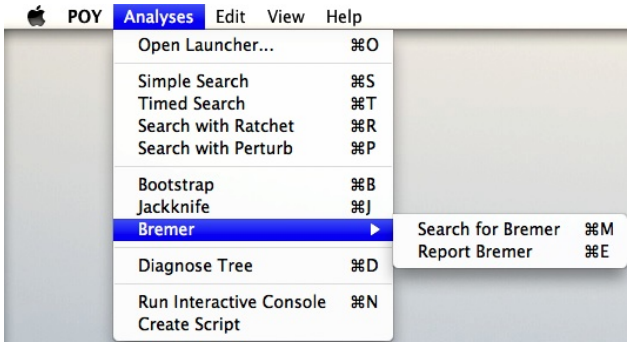

Figure 2.11: Selecting the *Bremer* windows from the *Analysis* menu.

Search for Bremer The script produced in this window collects trees visited during a search for Bremer support calculations. This search can take

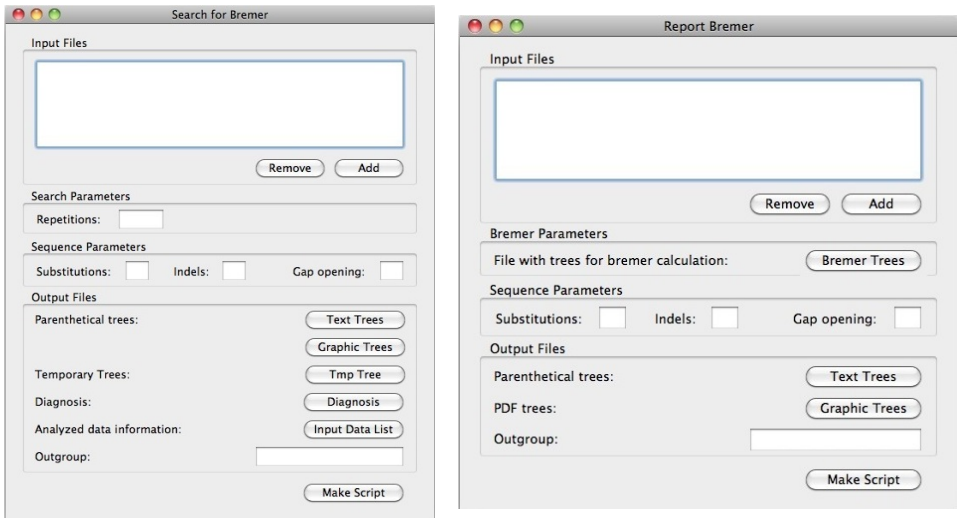

Figure 2.12: Viewing the options of the *Search for Bremer* (left) and the *Report Bremer*(right) windows.

a long time, as the goal of this search strategy is to broadly sample variation among trees, and guarantee that all clades have Bremer support values.

In addition to the standard four sections defined for the *Simple Search* window, that one of the output files is the *Temporary Trees* file, which contains all the information required to produce the Bremer support tree results in the *Report Bremer* window. Make sure to choose a file name that does not overwrite this output.

If the search does not finish within the time frame available to the user the search can be interrupted and the intermediate results remain stored in the *Temporary Trees* file. As Bremer calculations are upper-bound values, terminating the search prior to completion and, thus, storing a smaller pool of visited trees may inflate support values relative to those generated by a more exhaustive search. The trees from the *Temporary Trees* file can then be reported using the *Report Bremer* window.

Report Bremer The script produced in this window takes the *Temporary Trees* file generated in the *Search for Bremer* window in the *File with trees for Bremer calculation* field.

Diagnosis

Diagnose Tree

The *Diagnose Tree* window (Figure 2.13) specifies parameters for reporting a diagnosis of the input tree. This window lacks the *Search Parameters* section because the diagnosis is performed on the trees resulted from prior searches and no new trees are generated during the diagnosis procedure. In addition to the tree (or trees) file, the user must have input the data file associated with this tree file, in order to diagnose the tree(s) in this file.

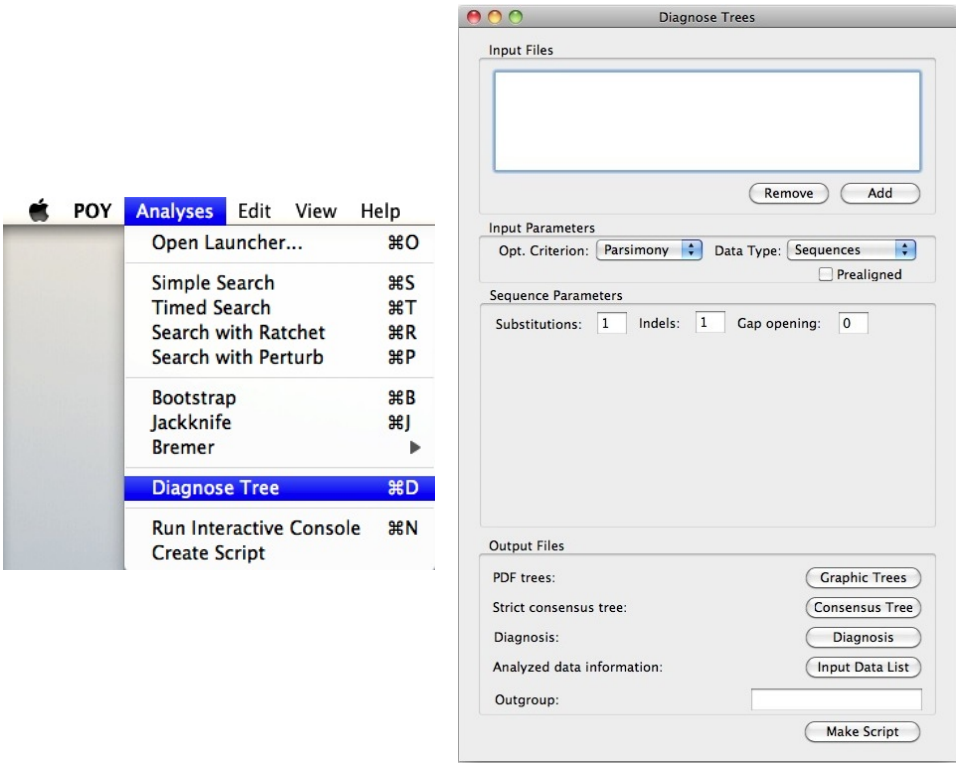

Figure 2.13: The *Diagnose* window. Selecting *Diagnose Tree* from the *Analysis* menu (left) and viewing the *Diagnose* window options (right).

Script editing and the Interactive Console

Open POY Launcher

Selecting *Open POY Script* (Figure 2.14) displays the *POY Launcher* window (Figure 2.2), the function of which is described above.

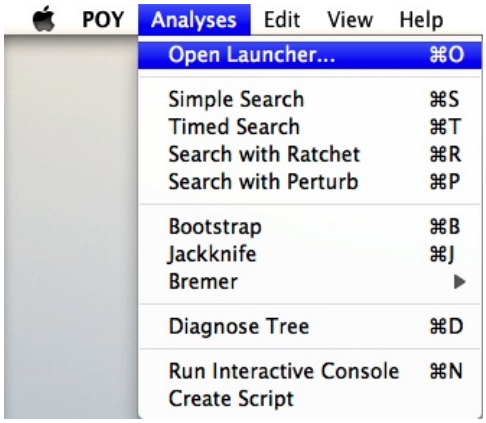

Figure 2.14: The *Open POY Launcher* selection opens the *POY Launcher* window.

Run Interactive Console

Selecting *Run Interactive Console* (Figure 2.15) opens the *ncurses* interface that enables the user to run the analysis interactively by entering `POY5` commands directly via the command-line interface of the *Interactive Console* See *Using the Interactive Console* (Section 2.5).

Create Script

The *Create Script* selection opens a blank *Script Editor* window that allows opening, creating, modifying, saving, and executing a customized script.

2.4.4 The *View* menu

The *View* menu contains the *Output* window which is subdivided into two fields: the upper *Results and Errors* and lower *Status of Search* (Figure 2.16). These fields display, respectively, the results (including warning and error messages) and the current state of the analysis. These fields are not updated

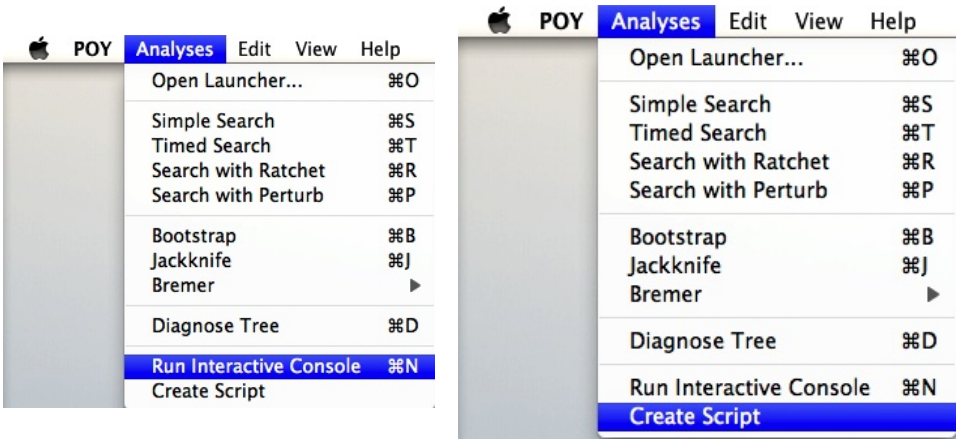

Figure 2.15: The *Run Interactive Console* selection (left) opens POY5 interactive console in a new window. The *Create Script* selection opens the *Script Editor* window (Figure 2.5).

automatically and in order to display the current state of the analysis the user must click the *Update* button. The *View* menu also contains the *About POY* window in Windows.

2.5 Using the Interactive Console

This section will help you get started using the POY5 *Interactive Console* and will prepare you for the more extensive, technical descriptions in the next chapter, *POY5 Commands*. This section will illustrate how to input data files, check the data you are analyzing, generate a set of initial trees, do basic branch swapping to find a local optimum, and, finally, produce and visualize the resultant trees, their consensus, and generate support values in a command-line environment rather than using a *Graphical User Interface*.

For the purpose of this exercise, two data files, which are included in the download package, are available at the POY5 download page.

http://www.amnh.org/our-research/computational-sciences/research/projects/systematic-biology/poy/download

- 28s.fas contains unaligned DNA sequences (partial 28S ribosomal RNA) in FASTA format. [37]

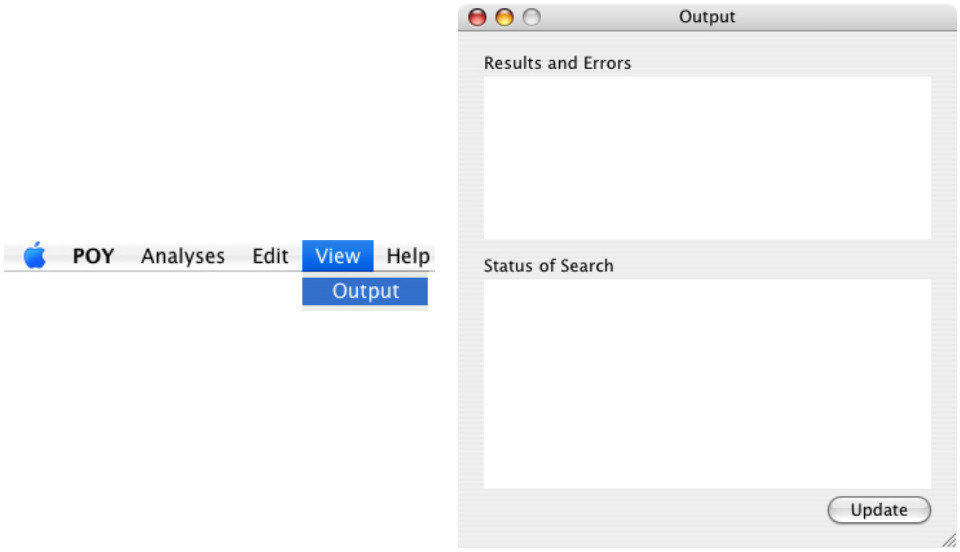

Figure 2.16: Selecting the *Output* window (left) and viewing the *Results and Errors* and *Status of Search* fields.

- `morpho.ss` contains a morphological data matrix in Hennig86 format. [12]

Once `POY5` has been launched and the interface (Figure 2.17) appears on the screen, data can be input and analysis can proceed. As you follow the instructions, you are encouraged to consult the help file by using the command `help` (see Section 2.7 to learn more about `POY5` commands and their arguments).

2.5.1 The interface

The *Interactive Console* provides a terminal environment with enhanced ability to display the results and the state of the analysis. We recommend the use of the console to explore and verify the data in the early steps of the analysis, and to learn the scripting language. Using the console requires familiarity with `POY5` commands, their arguments, and the conventions of `POY5` scripting (which are discussed in the *POY Commands* chapter). It has four windows: *POY Output*, *Interactive Console*, *State of Stored Search*, and *Current Job* (Figure 2.17):

POY Output (Figure 2.17, upper box) displays the status of the imported data, outputs the results of the phylogenetic analyses (such

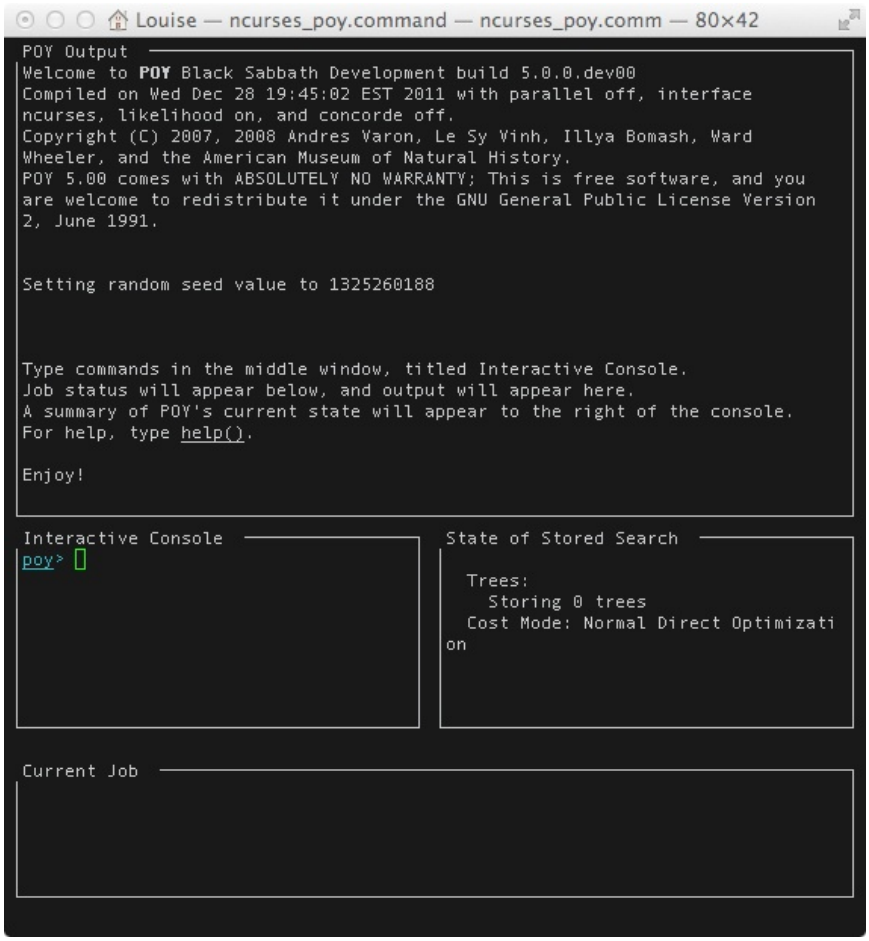

Figure 2.17: POY5 interface displayed in the Terminal window prior to analysis. Observe the cursor at the POY5 prompt in the *Interactive Console* and note that the *State of Stored Search* and *Current Job* windows are empty.

as trees, character diagnoses, and implied alignments), reports errors, and displays descriptions of `POY5` commands.

Interactive Console (Figure 2.17, mid-left box) is used to issue the commands interactively and to execute the commands by clicking the Return key. (See Section 3.1.1 on the description of `POY5` commands.)

State of Stored Search (Figure 2.17, mid-right box) displays the time (in seconds) elapsed since the initiation of the current operation. This window also reports the number of trees currently in m emory and displays the range of their costs.

Current Job (Figure 2.17, lower box) describes the currently running operation. When the operation is completed, the box is blank.

2.5.2 Starting a `POY5` session using the *Interactive Console*

Windows

- Start>All Programs>POY>Interactive Console

Mac OSX

- Double-click `POY5` application icon to start the program.
- Select *Run Interactive Console* from the *Analyses* menu.

Linux

- Add `/opt/poy5/Resources/` (or the location you plan to install) to your `PATH` and run `ncurses_poy` from a terminal.

2.5.3 Entering commands

Once this `POY5` interface is opened, the cursor appears in the *Interactive Console* portion of the window and is ready to accept commands. The *Interactive Console* does not support using the mouse and, as is true for most command-line applications, the cursor can be moved using the left and right arrow keys, and the Backspace (in Windows) or Delete (in Mac) keys are used to erase individual characters to the left of the current cursor position. To eliminate the need for retyping commands anew during a `POY5` session, keyboard shortcuts can be used: control-p ("previous") and control-n ("next")

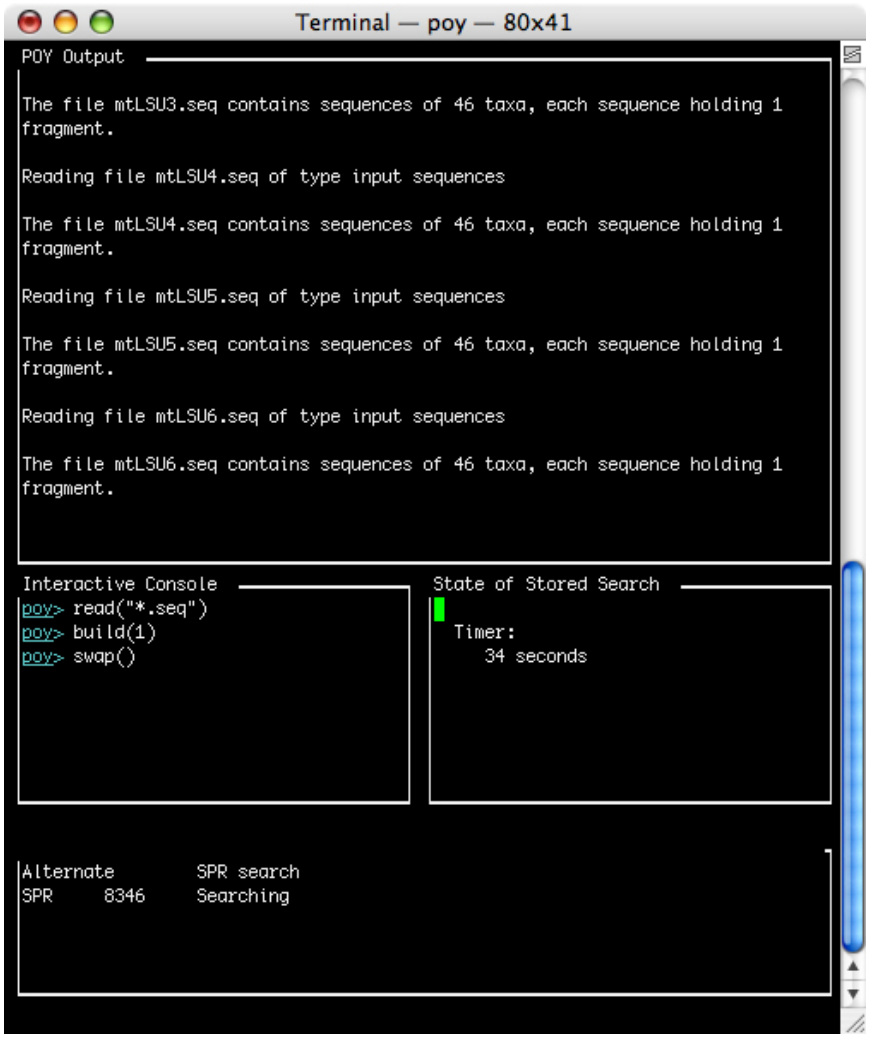

Figure 2.18: POY5 *Interactive Console* during a process. The *POY Output* window displays (by default) the information on the input data files. The *Interactive Console* lists the commands that have been executed. The *Current Job* window shows the state of the current operation and the current tree score. The *State of Stored Search* shows the time elapsed since the last command `swap`, was initiated.

will scroll through the commands previously entered during the session. In addition, the *Interactive Console* is equipped with the autocomplete feature: it involves `POY5` predicting a command, an argument, of file name that the user wants to type from the first letter(s) entered. Upon typing the first letter or part of the phrase, repeatedly pressing the TAB key scrolls through the list of command, argument, and file names that begin with that letter or phrase. Autocomplete simplifies interaction with the program.

2.5.4 Browsing the output

As output is reported in the *POY Output* window, only the most recent reports will be seen in the window. Using the Up and Down keys allows the user to scroll up and down the *POY Output* window to see the welcome line, and previously printed reports and help descriptions. Pressing Up and Down keys automatically places the cursor in the lower left corner of the *POY Output* window indicating that you are interacting with that window. Only 1000 lines are stored in the memory and the output that was reported before that will not be accessible by scrolling. The number of lines, however, can be modified by the user using the command `set()`, see `history` (Section 3.3.24). If the user desires to keep the entire output or specific items in the output, a log can be created using the command `set()`, see `log` (Section 3.3.24)) or specific outputs can be redirected to files (see `report` (Section 3.3.19)). The user should be aware that outputting a log file can slow down the program due to IO (input/output) delay.

2.5.5 Switching between the windows

To return to the *Interactive Console*, start typing and the cursor will automatically be placed back at the `POY5` prompt. When an operation is in progress (shown in the *Current Job* window), the cursor stays in the upper left corner of the *State of Current Search* window, and switching between the *Interactive Console* and the *POY Output* window is disabled. There are no user interactions in the *Current Job* or *State of the State of Current Search*.

2.5.6 Input of data

The basic command to input data in `POY5` is `read()`, which includes the list of files (in quotation marks and separated by commas) enclosed in parentheses. Suppose that we would like to simultaneously analyze morphological and molecular datasets, contained in separate data files, `morpho.ss` and `28s.fas`, respectively. We can issue a pair of `read()` commands (Figure 2.19):

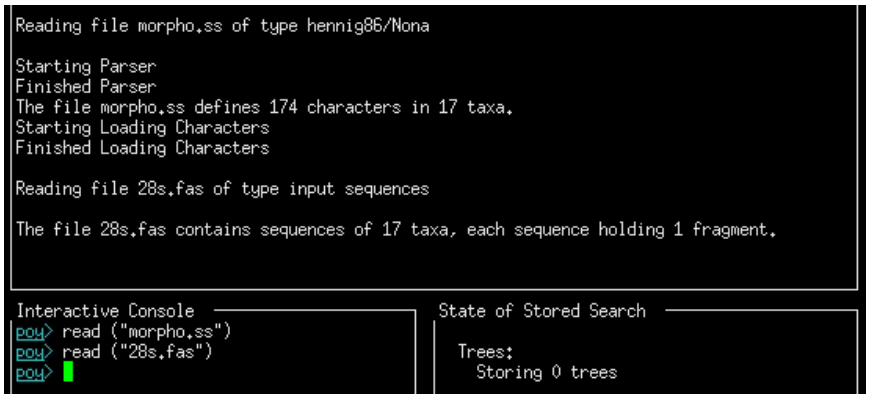

Figure 2.19: Importing data files using the *Interactive Console*. Two consecutive `read` commands specify both the morphological data file in Hennig86 format (`morpho.ss`), and the molecular data file in FASTA format (`28s.fas`). Observe that `POY5` automatically reports in the *POY Output* window the names and types of files that have been imported.

```
read("morpho.ss")
read("28s.fas")
```

The syntax of `read`, like every command in `POY5`, contains two elements: the name of the command, in this case `read`, followed by an optional list of arguments separated by commas and enclosed in parentheses. All filenames read into `POY5` should include the appropriate suffix for the file type (e.g. .fas, .ss, .aln, .tre etc:). Typically, the arguments of the command `read()` are names of data files, each being enclosed in double quotes (as shown in the example above). Even though there might be only one argument or none in some commands, parentheses (e.g. `pwd()`) always follow the command name. An exhaustive discussion of `POY5` command structure and detailed descriptions of all commands with examples of their usage are provided in the *POY Commands* chapter (3.1.1).

In order to import data by entering the names of the files, the directory containing these files must be identified. This can be established in two ways–by using the command `cd` to redirect the path to the directory where the data are found and then reading in the data file:

```
cd("/Users/username/docs/poyfiles")
```

```
read("28s.fas")
```

or by including the full path in the argument of `read`:

```
read("/Users/username/docs/poyfiles/28s.fas")
```

Most of the time users are interested in importing multiple data files to analyze an entire dataset. In this case, multiple data files can be specified as arguments for a single command. For example, importing both files, `morpho.ss` and `28s.fas`, can be written more succinctly:

```
read("morpho.ss","28s.fas")
```

or if the full path is included in the argument of `read` as:

```
read("/Users/username/docs/poyfiles/morpho.ss",
"/Users/username/docs/poyfiles/28s.fas")
```

This is equivalent to sequentially importing each file as shown in (Figures 2.19 and 2.20).

Figures 2.19 and 2.20 also illustrate an important feature that makes `POY5` different from many other phylogenetic analysis programs: every time a file is imported during a `POY5` session, the input data are *added* to the data in memory and *do not replace them*. This allows additional analytical flexibility. For example, if only morphological data are read and trees are built based on these data alone, a subsequently imported molecular character dataset will be used in conjunction with the previously imported morphological data, despite the fact that current trees in memory were generated only from morphological data (Figure 2.20):

```
read("morpho.ss")
build()
read("28s.fas")
rediagnose()
swap()
```

It must be noted that if the numbers of terminals differ among data files, *only* the data that correspond to the terminals used to generate the trees (in

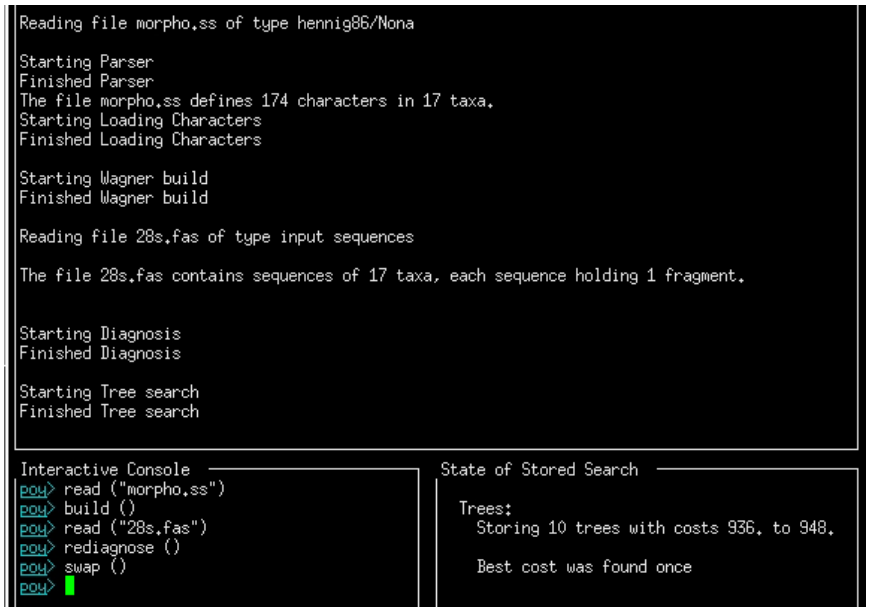

Figure 2.20: Building trees with morphological data only but continuing the analysis using combined morphological and molecular data. This example shows how we can add data to the analysis incrementally by loading files at different points in the search. First, the morphological data are imported from `morpho.ss` file using `read()` and trees are built based on these data. Then molecular data from the `28s.fas` file are loaded into memory. Finally, subsequent analyses, `rediagnose()` and `swap()`, are conducted using all the data in memory, that is the trees based on morphological data, and both morphological and molecular character sets.

this case, the morphological data file) are used. The rest of the character data are ignored, unless the trees are built again with the data files containing the expanded number of terminals. Also, because `POY5` appends trees and data in memory, it is a good practice when starting a new analysis within the same interactive session to clear the data using the command `wipe()`.

Valid input files include nucleotide and amino acid sequence files in many formats, and morphological data in Hennig86 and Nexus formats. (For information on specific formats supported by `POY5` and other types of input files see `read` (Section 3.3.14).)

2.5.7 Inspecting data

Once a dataset (or multiple datasets) is imported, `POY5` automatically reports a brief description of contents for each loaded file in the *POY Output* (Figure 2.19). However, it may be desirable to inspect the imported data in greater detail to ensure that the format and contents of the files have been interpreted correctly. This practice helps avoid common errors, such as inconsistently spelled terminal names, which may result in bogus results, produce error messages, and aborted jobs.

The basic command for outputting information is `report()`. One of its arguments, `data`, outputs a set of tables showing the list of terminals, the number and types of characters, and the lists of terminals and characters excluded from the analysis. To produce a report of the data files that were used in the previous example (`morpho.ss` and `28s.fas`), we import the data and execute `report(data)`:

```
read("morpho.ss","28s.fas")
report(data)
```

This will generate an extensive, detailed output, partial views of which are shown in Figure 2.21. Obviously, the entire report will not be visible in the *POY Output* window. Therefore, the Up and Down arrow keys and Page Up and Page Down keys can be used to scroll. By default, `POY5` reports the results of executed commands to the *POY Output* window. However, the same output can be redirected to a file simply by adding the name of the output file in the list of argument of the command `report()` *before* the argument specifying the type of the requested report (in this case `data`, see the command `report` (Section 3.3.19)). For instance, to output the data into the file `data_analyzed.txt` we would enter:

```
read("morpho.ss","28s.fas")
report("data_analyzed.txt",data)
```

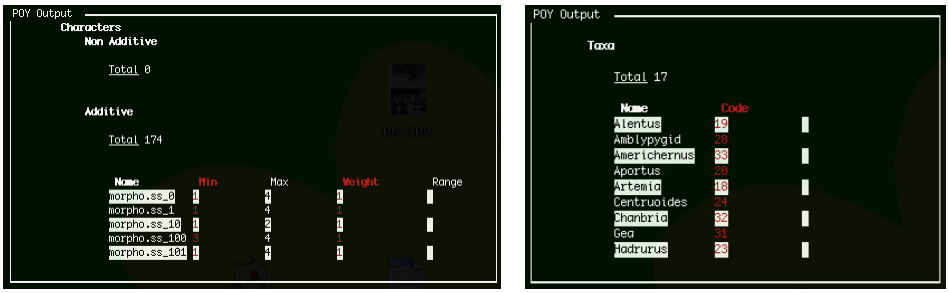

Figure 2.21: Inspecting imported data. The figure shows segments of a data report generated by `report(data)`. The left and right panels demonstrate a typical table output the character and terminal data respectively.

In this example, all the imported data are analyzed and, therefore, the report fields that list excluded data will appear empty. One can, however, exclude specific characters or terminals from the analysis using additional commands (see the command `report` (Section 3.3.19)).

Another useful argument of `report` is `cross_references`. This argument displays whether character data are present or absent for each terminal in each one of the imported data files. This provides a comprehensive visual overview of missing data. Building on the previous example, such output can be generated by the following sequence of commands:

```
read("morpho.ss","28s.fas")
report("cross_refs.txt",cross_references)
```

A typical output of `cross_references` command is shown in Figure 2.22. This argument is a very useful tool for visual representation of missing data. Moreover, reporting all the data to a cross references file can also highlight inconsistencies in the spelling of taxon names in different data files.

2.5.8 Building the initial trees

The command to build trees is `build()` (already mentioned in Section 2.5.6). After importing `morpho.ss` and `28s.fas`, executing the command `build()` without specifying any arguments (default settings) generates 10 Wagner trees by random addition sequence.

Many `POY5` commands operate under default settings when executed without arguments. To learn what the default settings are for a particular command use either the `help()` command with the command name of interest inserted in parentheses or consult the *POY Commands* chapter (3.1.1).

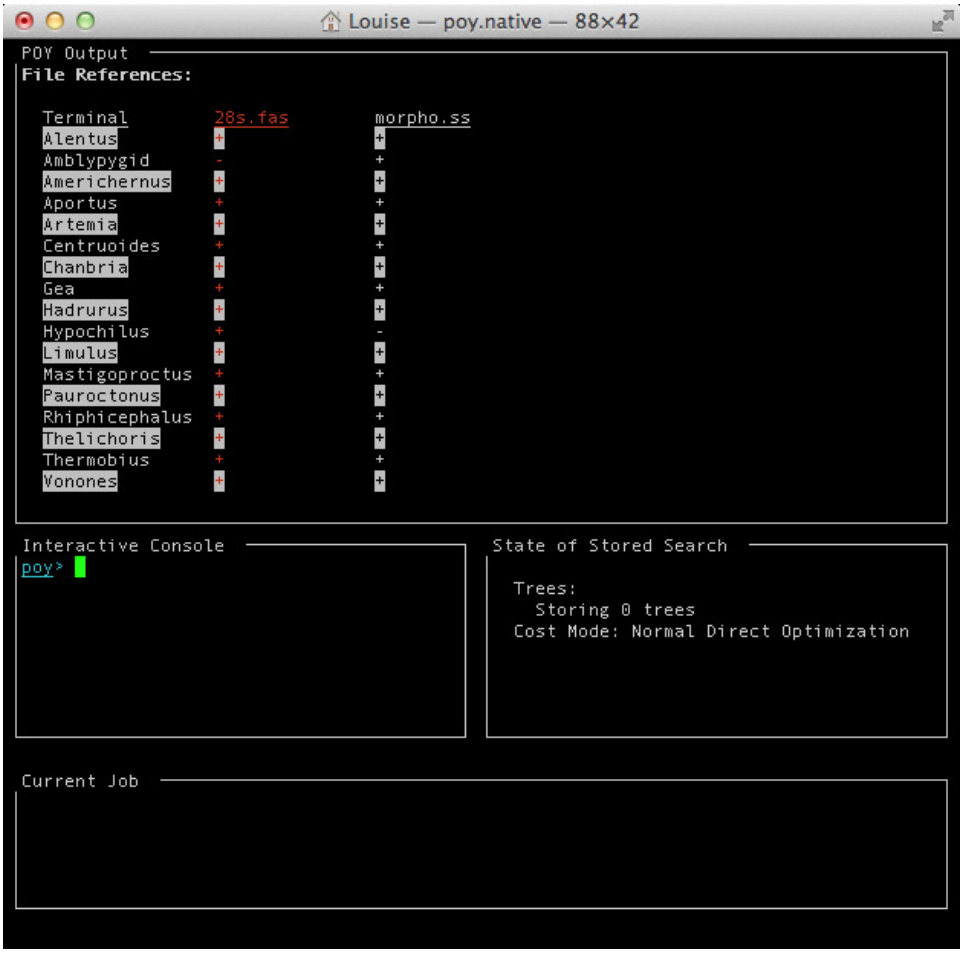

Figure 2.22: Visualizing missing data. The command `cross_references` displays a table showing whether a given terminal (in the left column) is present ("+") or absent ("–") in each data file. In this example, `28s.fas` is missing for Amblypygid and `morpho.ss` for Hypochilus.

Figure 2.23: Generating Wagner trees. During the process of tree building (left panel), the *Current Job* window displays how many builds have been performed so far (`57 of 100`), the number of terminals added in the current build (`13 of 17`), the cost of a current tree recalculated after each terminal addition (`362`), and the estimated time (in seconds) for the completion of the operation (`4 s`). Because the process is not complete, the *State of Stored Search* window contains no trees. Once tree building is complete, the *State of Stored Search* window displays the best (`451`) and worst (`472`) costs, the number of trees stored in memory (`100`), and the number of trees with the best cost (`2`).

If the user would like to specify a number of tree building replicates different from the default value of 10, the argument `trees` followed by a colon (":") and an integer specifying the number of trees must be included in the argument list of the `build` command: `build(trees:100)`. This command has a shortcut that omits the argument `trees`. Thus, `build(trees:100)` is equivalent to `build(100)`. As defaults, the shortcuts are fully described in Section 3.1.1. The entire sequence of commands minimally required to import the data and build 100 trees is the following:

```
read("morpho.ss","28s.fas")
build(100)
```

As the tree building advances, the *Current Job* window displays the current status of the operation (Figure 2.23). This window shows how many Wagner builds have been performed out of the total number requested, the number of terminals added in the current build, the cost of the current tree (recalculated after each terminal addition), and the estimated time for the completion of all the builds. When all the trees are generated, the *State of Stored Search* window displays the range of tree costs (the best and worst costs), the number of trees stored in memory, and the number of trees with the best cost (Figure 2.23).

Figure 2.24: Performing a local search. When searching (left panel), the *Current Job* window reports the number of the tree that is currently being analyzed (`73 of 100`), a method of branch swapping (`Alternate`), a function being currently performed (`SPR search`), and a cost of the current tree (`456`). When the searching is finished (right panel), the *State of Stored Search* window displays the best (`446`) and worst (`463`) costs, the number of trees stored in memory (`100`), and the number of trees of the best cost (`9`) recovered from independent tree builds and swaps. Notice that these trees may not necessarily have unique topologies.

2.5.9 Performing a local search

Now that the trees have been generated and stored in memory, a local search can be performed to refine and improve the initial trees by examining additional topologies of potentially better cost. The command `swap()` implements an efficient strategy by performing SPR and TBR branch swapping alternately. As with other commands, the arguments of `swap()` allow the customization of the swap algorithm. In the following example, branch swapping is performed under the default settings on each of the 100 trees build in the previous step:

```
read("morpho.ss","28s.fas")
build(100)
swap()
```

Branch swapping is performed sequentially on all trees stored in memory. During swapping, the *Current Job* window reports the number of the tree that is currently being analyzed, the method of branch swapping, the specific routine being currently performed, and the cost of the current tree (Figure 2.24). When the process is complete, the *State of Stored Search* window displays the range of tree costs (the best and worst costs), the number of trees stored in memory, and the number of trees with the best cost (Figure 2.24). Notice that the local search had reduced the costs of the initial best (from 451 to 446) and narrowed the range of tree costs.

Using different combinations of `swap()` arguments allow the designation of a large number of search strategies with different levels of complexity. Some simple options allow the choice between SPR and TBR. More complex strategies allow keeping a specific number of best trees per single initial tree (generated during the building step). For example, the command `swap(trees:10)` will keep up to 10 best trees generated during branch swapping on a single initial tree. Consequently, if 100 trees were built initially, this command will produce up to 1,000 trees. The argument `threshold` allows the retention of suboptimal trees within a specified percent of cost difference from the current best tree. For example, `swap(trees:20,threshold:10)` will execute a swap considering trees within a ten percent cost difference of the current best tree and retain the 20 minimal length swapped trees for each initial build. Other options provide the means to sample trees as they are evaluated, timeout after certain number of seconds, transform the cost regime, and other functions in conjunction with many `POY5` commands.

2.5.10 Selecting trees

Having performed the basic steps of importing character data, building initial trees, and conducting a simple local search, we obtained a set of locally optimal trees in memory. Generally, a user would like to select only those trees that are both optimal *and* topologically unique. The default setting of the `select()` does exactly that. Adding `select()` to our example of command sequence for the basic analysis

```
read("morpho.ss","28s.fas")
build(100)
swap()
select()
```

selects only unique trees of best cost. The remaining trees are deleted from memory. The *State of Stored Search* window reports the number and the cost of the best tree(s) (Figure 2.25).

As an alternative, the user may choose to select topologically unique trees, regardless of the cost, using `select(unique)`. This may ensure that a larger tree space is explored. If this is used as an option during the search, the user should remember to `select()` at the end of the run, prior to reporting the results.

The command `select()`, is another multifunctional command the arguments of which are also used to select (include or exclude) specific terminals,

Figure 2.25: Selecting unique best trees. Executing `select()` keeps only unique trees of best cost. The *State of Stored Search* window reports that there is only one unique tree of best cost (`446`).

characters, and trees.) Comparing the output reported in the *State of Stored Search* before (Figure 2.24) and after (Figure 2.25) executing `select()` shows that swapping on 9 of 100 initial trees produced the trees of best cost (`446`), but these trees are identical, because only one was retained when filtered using `select()`.

2.5.11 Visualizing the results

There are several options for visualizing results in `POY5` (see `report` (Section 3.3.19)). The command `report("my_first_tree",graphtrees)` outputs a cladogram in PDF format (Figure 2.26), which can be displayed, edited, and printed using graphics software (such as Adobe Illustrator or Preview). `POY5` also appends the "pdf" extension when generating graphic output to a file. A quick way to see the tree(s) on screen is to use the command `report(asciitrees)` that draws a cladogram in the *POY Output* window (Figure 2.26). The ascii tree(s) can also be reported to a file, if an output file name is specified within the command (`report("my_first_trees", asciitrees)`). These trees will be saved to a text file.

The command `report("my_first_trees.txt",trees)` reports the trees in memory in parenthetical notation to the file `my_first_trees` that can be imported in other programs. Supported tree output formats include Newick and Hennig86. `report()` can also generate consensus trees in the graphical and parenthetical formats when appropriate arguments are specified (for example, `report("strict_consensus",graphconsensus)`).

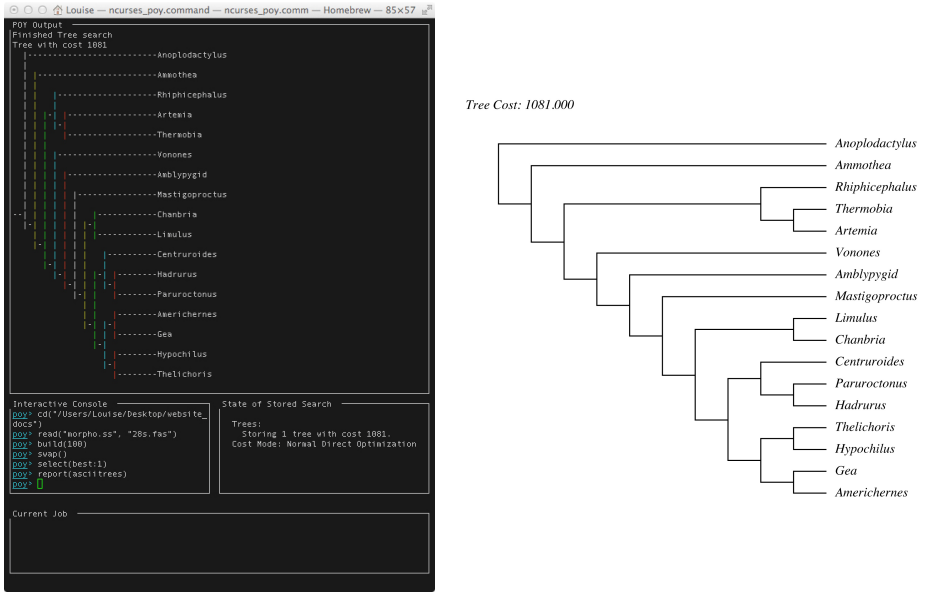

Figure 2.26: Visualizing trees. An ascii tree (left) is generated using the command `report(asciitrees)`. The same tree is reported to a file in a PDF format (right) using `report("my_first_tree",graphtrees)`. Observe that both representations of trees are preceded by their costs.

2.5.12 Interrupting a process

To interrupt a process, press control-c. By default, an error, `Error: Interrupted`, is reported in the *POY Output* window. The program does not close, however, and a new command can be entered. Interrupting the analysis cancels the execution of the last command requested by the user and restores the data and trees in memory before that last command. For example, the following two session are equivalent:

```
read("morpho.ss") <ENTER>
```

and

```
read("morpho.ss") <ENTER>
read("28s.fas") <CONTROL-C>
```

In both of these sessions, only the morphological dataset ‘‘`morpho.ss`’’ is read into `POY5`.

2.5.13 Reporting errors

If there is an error pertaining to incorrect syntax (such as a typo in a command name), `POY5` will indicate the location of the error by underlining the problematic part of the input with a hat symbol (“^”) in the *Interactive Console* (Figure 2.27). The description of the corresponding command, its syntax, and examples of its usage from the help file are automatically printed in the *POY Output* window. As noted above, the Up and Down keys can be used to scroll through the output and determine the source of the error. Certain types of errors are reported explicitly (Figure 2.27).

2.5.14 Exiting

To finish a `POY5` session, enter the command `exit()` (Figure 2.28) or `quit()`. This will close the `POY5` interface and resume the Terminal window (Mac OSX) or the Command Prompt window (Windows).

2.6 Creating and running `POY5` scripts

So far, we have communicated with `POY5` interactively through the *Graphical User Interface* or by executing commands from the *Interactive Console*.

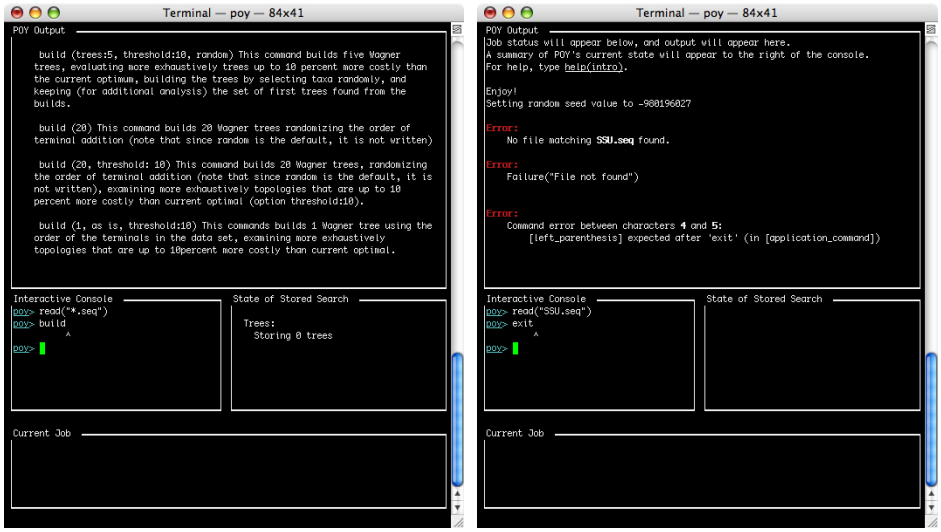

Figure 2.27: Displaying errors. POY5 displays error messages in several ways. In the example in the left panel, the command `build` was entered without parentheses, which is required for a valid POY5 command syntax; the exact place of the error is marked by "^", in this case following the `build` commands. Examples of the proper usage of the command are automatically displayed in the *POY Output*. In other cases (right panel), error messages are explicitly reported in the *POY Output* window. The first and second error messages indicate that the data file `SSU.seq` is not present, which could have been caused either by a mistake in the name of the file, or missing file, or the location of the file in a directory, other than the one specified prior to starting the POY5 session. The third error message indicates that the valid syntax of `exit` requires the parentheses following the command name (also shown by "^" in the *Interactive Console*).

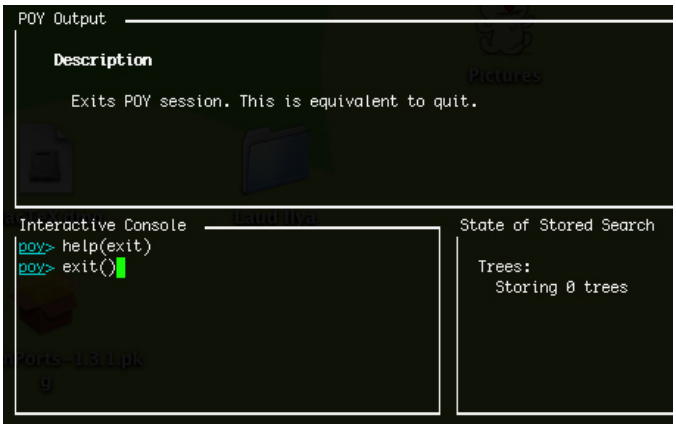

Figure 2.28: Exiting POY5

Another way of conducting an analysis is to run a *script*, a simple text file containing a list of commands to be performed (Figure 2.29).

Running analyses using scripts has many advantages: not only does it allow for the entire analysis to proceed from the beginning to the end at one click of a button, but it also provides means to examine the logical dependency of the commands and optimize memory consumption (see the description of `script_analysis` argument of the command `report` in the *POY Commands* chapter). Submitting jobs using scripts may produce results faster because POY5 automatically optimizes the workflow of the entire analysis by taking into account the functional relationships among various tasks and efficiently distributing the jobs and resources (such as memory and multiple processors).

Another advantage of using scripts is that they may contain comments that are ignored by POY5 but can be helpful to describe the contents of the files and provide other annotations. The comments are enclosed in parenthesis *and* asterisks, e.g. `(* this is a comment *)`. Comments can be of any length and span multiple lines.

Obviously, using scripts requires the user to design the workflow of the process prior to conducting the analysis. POY5 scripts can be created and saved using the *Script Editor* window of the POY5 *Graphical User Interface* or any conventional text editor (such as TextPad, TextWrangler, BBEdit, Emacs, or NotePad).

POY5 scripts are extremely useful in cases when operations may take a long time to complete, eliminating the need to wait for a part of the analysis to finish in order to proceed to the next step. Moreover, scripts can contain a

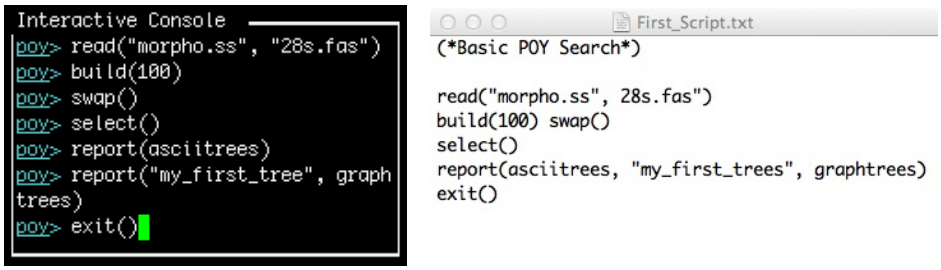

Figure 2.29: Using POY5 scripts. The list of commands executed interactively using the *Interactive Console* (left) and a script containing the same list of commands (right). Observe that the header of the script is a comment, enclosed in "(* *)", that is ignored by POY5. Also note that commands can either be listed in a row or in a column (compare `build()` and `swap()` in the console and in the script) and different arguments of the same command can either be specified separately or combined in a single argument list (compare `report()` in the console and in the script). (Both conventions are valid for interactive command submission and for scripts.)

series of individual scripts that are run sequentially (see for example tutorial 5.4.

There are two ways to import and run a script:

- using the *POY Launcher* in the *Graphical User Interface*;
- using the command `run()` of the *Interactive Console*; for example, `run("script.txt")`, where `script.txt` is the name of the file containing the script. Within this script it is also possible to specify additional scripts to run.

It is critical to include the command `exit()` at the end of the script. Otherwise POY5 will be waiting for further instructions to be entered after executing the script's contents.

2.7 Obtaining help

Instructions to run POY5, command descriptions, and the theory behind POY5 can be obtained from a variety of sources.

POY5.0 Program Documentation (this manual) is a comprehensive and detailed manual on all the aspects of using POY5, from installation to

output and visualization of results. Included are *Quick Start*, `POY5` command reference, practical guides and tutorials that make the program immediately accessible for beginners and provide in-depth information for experienced users. As with any PDF, this document is easily searched using key words or phrases. The documentation in PDF format can be accessed from the *Help* menu of the graphical user interface or downloaded separately from `POY5` web site at

http://www.amnh.org/our-research/computational-sciences/research/projects/systematic-biology/poy/download

POY interactive help can be obtained by entering `help()` at the `POY5` *Interactive Console*. To obtain help on a particular command, the name of the command must be specified in the parentheses following `help()`. For example, to learn about the command `exit`, type `help(exit)`. (Figure 2.28.)

POY5 Mail Group is an Internet-based forum for discussing all issues related to `POY5` and provides the best way to communicate with `POY5` developers on specific issues (see *WWW resources* below). The website is located at https://groups.google.com/forum/#!forum/poy4. Questions relating to both `POY4` and `POY5` can be posed to this group.

POY Book (Wheeler et al., 2006 *Dynamic Homology and Phylogenetic Systematics: A Unified Approach Using POY* [69]) provides a review of the theory behind `POY4` and by extension `POY5`, and contains formal descriptions of many algorithms implemented in the program and the descriptions of commands of the earlier version, `POY3`. A PDF version of the book is available at http://research.amnh.org/scicomp/pdfs/wheeler/Wheeler_etal2006b.pdf

POY Paper (Varón et al., 2010. *POY version 4: Phylogenetic analysis using dynamic homologies* [54] provides a description of the overall goals, implementation and philosophy of POY.

2.8 WWW resources

`POY5` is an ongoing project and new versions are being continuously developed to include new procedures, improve performance, and eliminate reported bugs. Therefore, it is imperative to keep up with the program's development

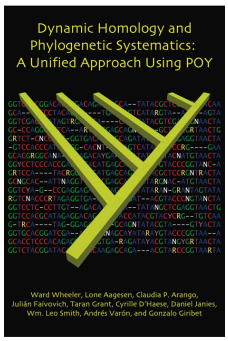

Figure 2.30: The POY Book.

and check regularly for updates. There are several Internet-based resources that offer this information.

POY5 Web Site Has downloadable compressed files of POY5 binaries, source code, and documentation in PDF format. It also provides a links to the *POY Mail Group*. The website is hosted by AMNH Computational Sciences at

http://www.amnh.org/our-research/computational-sciences/research/projects/systematic-biology/poy

POY5 source code repository Contains has downloadable POY5 source code. The site is powered by Google at

http://code.google.com/p/poy/source

POY5 Mail Group Informs registered users via email of new developments, such as new versions and updates. It also provides additional resources for obtaining help and a way for reporting bugs and other problems with POY5 and its documentation. In addition, it allows users to receive and respond to each other's questions thus providing an open forum to discuss the methods and applications of POY5. The users who choose not to register, have access to the archives of the postings but will not be able to either submit or receive emails from other users and POY5 developers. The *POY5 Mail Group* is hosted by Google at

https://groups.google.com/forum/#!forum/poy4

Chapter 3

POY5 Commands

3.1 POY5 command structure

3.1.1 Brief description

POY5 interprets and executes *scripts* issued by the end user. These can come from the *Graphical User Interface* and the command line in the *Interactive Console* of the program, or 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 in lower case (LIDENT), followed by a list of arguments separated by commas and enclosed in parentheses. Most of the arguments are optional, in which case POY5 has default values.

In POY5, we recognize four types of command arguments: *primitive values*, *labeled values*, *lists of arguments*, and *commands*.

Primitive values can be either an integer (INTEGER), a real number (FLOAT), a string (STRING), or a boolean (BOOL).

Labeled arguments are lowercase identifiers (which are referred to as *label*), and an argument, separated by the colon character (":"). Examples of identifiers include items such as characters, terminals, all, characters, names of files, missing, codes etc: In contrast to lists of arguments, which are enclosed in parentheses (see below), when only one argument can follow a command, parentheses are not required.

List of arguments are several arguments enclosed in parenthesis and separated by commas (",").

Commands are standard commands that can affect the behavior of another command when included in its list of arguments.

Thus, certain commands can function as arguments of other commands. Moreover, some commands share arguments. Although such compositional use of commands might seem complex, this structure provides much more intuitive control and greater flexibility. The fact that the same logical operation that functions in different contexts maintains the same name (typically suggestive of its function), substantially reduces the number of commands without limiting the number of operations. Using a linguistic analogy, `POY5` specifies a large number of procedures by a more complex grammar (specific combinations of commands and arguments), rather than by increasing the vocabulary (the number of specific commands and arguments). For example, the command `swap` specifies the method of branch swapping. This command is used to conduct a local search on a set of trees. In addition, `swap` functions as an argument for `calculate_support` to specify the branch swapping method used in each pseudoreplicate during `jackknife` or `bootstrap` resampling. `swap` can also be used to set the parameters for local tree search based on perturbed (resampled or partly reweighted) data as an argument of the command `perturb`. Therefore, to take the maximum advantage of `POY5` functionality, it is essential to get acquainted with the grammar of `POY5`.

3.1.2 Grammar specification

The following is the formal specification of the valid grammar of a script in `POY5`:

```
script: = | command
          | command script

command: = LIDENT "(" argument list ")"

argument list: = |
                 | arguments

arguments: = |
             | argument
             | argument "," arguments

argument: = | primitive
```

```
            | LIDENT
            | LIDENT ":" argument
            | command
            | "(" argument list ")"

primitive: = | INTEGER
            | FLOAT
            | BOOLEAN
            | STRING

LIDENT: = [a-z_][a-zA-Z0-9_]*

INTEGER: = [0-9]+

FLOAT: = | INTEGER
         | [0-9]+ "." [0-9]*

STRING: = """ [^"]* """
```

The following examples graphically show a typical structure of valid `POY5` commands formally defined above. The Figure 3.1 illustrates the syntax of the command `swap`. The name of the command, `swap`, is followed by a list of two arguments, `tbr` and `trees:2`, enclosed in parentheses and separated by a comma. Note that `trees:2` is a labeled-value argument that contains a label (`trees`) and a value (`2`) separated by a colon.

Figure 3.2 shows a more complex command structure, using the command `perturb` as an example. This is a compound command because the list of its arguments contains another command, `swap`. This means that executing `perturb` performs a set of specified operations that contains a nested set of operations specified by `swap`. Note also, that in contrast to the first labeled-values argument `iterations`, the second labeled-values argument `ratchet` has multiple values (a float and an integer). When multiple values are specified, they must be enclosed in parentheses and separated by a comma. The third argument is a command (`swap`), therefore it is syntactically distinguished from other arguments, labeled and unlabeled alike, by having parentheses following the command name. It must be emphasized that the parentheses always follow the command name even if no arguments are specified. If no arguments are specified, a command is executed under its default settings

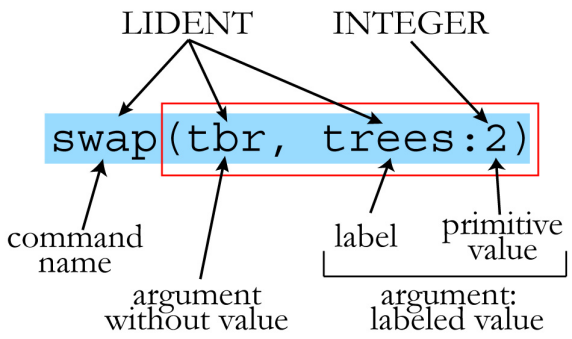

Figure 3.1: The structure of a simple POY5 command. The entire command (highlighted in blue), consists of a command name followed by a list of arguments (enclosed in red box). See text for details.

provided it has default settings. If a command has no default settings *e.g.* `transform`, then typing `transform ()` does nothing.

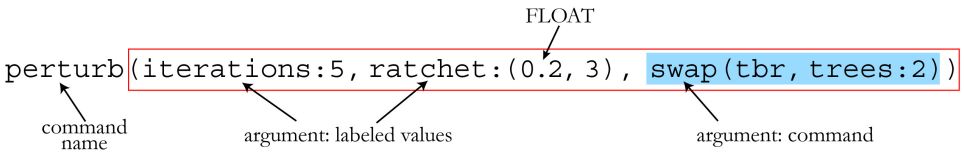

Figure 3.2: A structure of a compound POY5 command. Note that the list of arguments (enclosed in red box) includes a command (highlighted in blue). Also, note that `ratchet` accepts multiple values, a float and an integer, that are inclosed int parentheses and separated by a comma. See text for details.

3.2 Notation

Some arguments are obligatory, whereas others are not; some commands accept an empty list of arguments, but others do not; some argument labels have obligatory values, some have optional values. The POY5 commands and arguments are listed alphabetically in the next section. In the descriptions of POY5 commands below, the elements of POY5 grammar are defined in the text using the following conventions:

- A command that could be included in a POY5 script (that is can be entered in the interactive console or included in an input file) is shown in `terminal` typeface.

- Optional items are inclosed in `[square brackets]`.
- Primitive values are shown in `UPPERCASE`.

Each command description entry contains the following sections:

- The name of the command.
- The valid syntax for the command.
- A brief description of the command's function.
- A list of descriptions of valid arguments.
- Description of default settings.
- Examples of the command's usage.
- Cross references to related commands.

NOTE

Default syntax. The default syntax for all commands is the same: it includes the command name followed by empty parentheses, e.g. `swap()`. However, within the descriptions of each command the default settings include the entire argument list for illustrative purposes only (i.e. in the case of `swap()` the entire argument list appears as `swap(trees:1,alternate,threshold:0,bfs)`).

NOTE

Command order. The effect of the order of arguments in a command depends on the context. If arguments are not logically interconnected, their order is not important. For example, the commands `build(10,randomized)` and `build(randomized,10)` are equivalent. However, executing the commands `transform(tcm:(1,1),gap_opening:4)` and `transform(gap_opening:4,tcm:(1,1))` will produce different results because `gap_opening` *modifies* the values set by `tcm`, while `tcm` *overrides* the values set by `gap_opening`.

NOTE

Output files. When an output file is specified, the file name (in double quotes and followed by a comma) must precede the argument, e.g. `report("first_trees", trees)`.

3.3 Command reference

3.3.1 build

Syntax

build([argument list])

Description

Builds Wagner trees [11]. Building multiple trees with a randomized addition of terminals allows for the evaluation of many possible tree topologies and generates a diversity of trees for subsequent analysis. The arguments of the command `build` specify the number of trees to be generated and the order in which terminals are added during a single tree building procedure. During tree building, `POY5` reports in the *Current Job* window of the *ncurses* interface which of the terminal addition strategies (e.g. `as_is` or `randomized`) is currently used.

By default `POY5` replaces the trees stored in memory with those generated in a subsequent build. For example, executing `build(10)` followed by `build(20)` will replace the 10 trees generated during the first build with 20 new trees. However, it might be desirable to generate a large number of trees by appending trees from multiple separate builds. To keep trees from consecutive builds, a tree output file must be specified using the command `report` (Section 3.3.19) that must precede the `build` command. This will produce a file containing the trees appended from all builds. If the same file name is used for reporting trees for other analysis, the new trees are appended. Alternatively, trees from different builds can be redirected to separate files if different file names are specified.

The command `build` is also used as an argument for the command `calculate_support`.

Arguments

`all[:INTEGER ]` Turns off all preference strategies for adding branches and simply tries all possible addition positions for all terminals. By default ten trees are built but the number of trees can be specified by the integer or by the argument `trees`.

`as_is` Indicates that in one of the trees to be built, the terminals are added in the order in which they appear in the imported data files, and all others are built using a random addition sequence.

`branch_and_bound[:FLOAT ]` Calculates the exact solution using the Branch and Bound algorithm [25]. By default only one optimal tree is kept but the number of optimal trees to be retained can be specified by the argument `trees`. The optional float value specifies the initial bound (either tree cost or likelihood score).

`constraint[:STRING ]` Builds trees using the set of constraints specified by a consensus tree input file. If no input file is provided, the constraint is calculated as the strict consensus of the trees in memory. Every tree built using this method is subjected to the same randomization as Wagner builds within each constraint. Constraining a tree is useful in hypothesis testing.

`INTEGER` The integer argument specifies the number of independent, individual Wagner tree builds. This is a shortcut of the argument `trees`.

`lookahead:INTEGER` The number of trees that can be kept at each build step. If the `lookahead` argument specifies a number n, and the best tree found has cost c, then the best n trees with cost at most $c + threshold$ as specified by the `threshold` (Section 3.3.1) command are held for the next build step. If no `threshold` command is specified, then it is set to 0.

`of_file:STRING` Imports a tree file included in the file path of the argument. This command is useful for importing starting trees for calculating `bremer` (Section 3.3.2) support. In other contexts the command `read` (Section 3.3.14) can be used with the same effect.

`optimize(model[:LIDENT ],branch[:LIDENT ])` Specifies when the likelihood model and how the branches are optimized during the build routine. These options are also available in the `fuse` and `swap` commands. In all cases a complete round of optimization will occur after the completion of a build.

`model:always` Optimize the model after every additional taxon is added.

`model:max_count:INTEGER` Optimize the model after every defined number (`INTEGER`) of taxa are added to the tree.

`model:never` Do not optimize the model during the build (the default).

`branch:all_branches` Optimize all branch lengths after each taxon is added.

`branch:join_region` Optimize a maximum of five branches; the edge connecting the new taxa to the tree, and the two sides of that joined edge (the default).

`branch:never` Do not optimize the branches during the build process. We use an estimate based on the proportion of sites that transform.

`nj` Creates a tree using the Neighbor Joining algorithm [41]. If more than one tree is requested, all the trees will be the same (the algorithm implementation is deterministic).

`random` Generates a tree at random. All possible trees have equal probability.

`randomized` Indicates that terminals are added in random order on every Wagner tree built. This is a default tree-building strategy.

`STRING` This is a shortcut of the argument `of_file`.

`threshold:FLOAT` The numerical value specifies the extra cost over the current best tree that makes another tree acceptable for the lookahead list. This parameter is only useful if `lookahead` (Section 3.3.1) is used.

`trees:INTEGER` The integer value specifies the number of independent, individual Wagner tree builds. The label `trees` is optional: it is sufficient to specify only the integer value. Therefore, `build(5)` is equivalent to `build(trees:5)`. Note that `trees` is also used as an argument of the command `swap` (Section 3.3.26) but with different meaning.

The value `0` generates no trees but it *retains* all trees in memory. This is useful, for example, in the `bremer` (Section 3.3.2) support calculation, where instead of generating new trees per each node, the searches are performed on the trees in the neighborhood of the current trees in memory.

Defaults

`build(trees:10,randomized,lookahead:1,threshold:0)` By default, `POY5` will build 10 trees using a random addition sequence for each of them.

Examples

- `build(20)`
  Builds 20 Wagner trees randomizing the order of terminal addition [Note: because the argument `randomized` is specified by default, it can be omitted].

- `build(trees:20,randomized)`
  A more verbose version of the previous example. By default a build is randomized, but in this case the addition sequence is explicitly set. For the total number of trees, rather than simply specifying `20`, the label `trees` is used. The verbose version might be desirable to improve the readability of the script.

- `build(all:30)`
  Builds 30 Wagner trees, trying all possible addition positions for all terminals.

- `build(15,as_is)`
  Builds the first Wagner tree using the order of terminals in the first imported data file and generates the remaining 14 trees using random addition sequences.

- `build(branch_and_bound,trees:5)`
  Builds trees using branch and bound method and keeps up to 5 optimal trees in memory.

- `build(constraint:"cstree.tre")`
  Builds trees using using the set of constraints specified by the consensus tree file "cstree.tre".

- `build(trees:100,optimize(model:max_count:5,branch:all_branches))`
  Builds 100 trees and optimizes the likelihood model after every 5 taxa are added to the tree. All branch lengths are optimized after the addition of each taxon to the tree.

3.3.2 calculate_support

Syntax

calculate_support`([argument list])`

Description

Calculates the requested support values. POY5 implements support estimation based on resampling methods (Jackknife [14] and Bootstrap [17]) and Bremer support [6, 28]. The Jackknife and Bootstrap support values are computed as frequencies of clades recovered in trees specified (see below). All the arguments of `calculate_support` command are optional and their order is arbitrary. For examples of scripts implementing support measures see tutorials `5.4`, `5.5` and `5.6`.

The `calculate_support` command does not output support values by default. The output of support values must be requested using the command `report` (Section 3.3.19). This is particularly important for Jackknife and Bootstrap support values, as these sampling techniques do not require the presence of trees in memory. Therefore, it is possible to perform the sampling for support values *before* the tree of interest has been found.

NOTE

In the context of dynamic homology, the characters being sampled during pseudoreplicates are entire sequence fragments, not individual nucleotides. Consequently, the bootstrap and jackknife support values calculated for dynamic characters are not directly comparable to those calculated based on static character matrices. If it is desired to perform character sampling at the level of individual nucleotides, the dynamic characters must be transformed into static characters using `static_approx` argument of the command `transform` (Section 3.3.26) prior to executing `calculate_support`. Alternatively, an output file in Hennig86 format can be generated based on an implied alignment using `phastwinclad` (Section 3.3.19) that can subsequently be analyzed using other programs.

Of course, if the dataset of dynamic characters contains a large number of fragments, this caveat may not be warranted.

It is important to remember that the local optimum for the dynamic homology characters can differ from that for the static homology characters based on the same sequence data. Therefore, it is recommended to perform an extra round of swapping on the transformed data to reach the local maximum for the static homology characters prior to calculating support values.

NOTE

The placement of the root affects calculation of Bremer support values. Therefore, it is critical to specify the root prior to executing `calculate_support`. See the description of the command `set` (Section 3.3.24) on how to specify the root.

Arguments

Support calculation methods The following commands allow selection among several methods for calculating support.

`bootstrap[:INTEGER | LIDENT ]` Calculates Bootstrap support [17]. The `INTEGER` and `LIDENT` are both optional. The integer value specifies the number of resampling iterations (pseudoreplicates)—if the value is omitted, 5 pseudoreplicates are performed by default. The `lident` specifies the tree for which bootstrap support values will be calculated: if `individual` (the default), bootstrap values will be calculated and reported for each of the optimal trees stored in memory (these will be reported sequentially); if `consensus`, a consensus tree based on the best trees recovered in each replicate with zero-length branches collapsed will be calculated and chosen; if `STRING`, a tree file containing a single tree, in parenthetical notation, is chosen.

`bremer` Calculates Bremer support values [6, 28] for each tree in memory by performing independent constrained searches for each node. The parameters for the searches can be modified using arguments described under *Search strategy*. [Note: an alternative, more timely way of calculating Bremer support valves is to use the "visited" option (see tutorial `5.4`).]

`jackknife[:([argument list])]` Calculates Jackknife support [14] using the sampling parameters specified by the arguments. The arguments of `jackknife` are optional and their order is arbitrary. If these are not specified, default values for each of these arguments will be used. As in the case of `bootstrap` support calculation, the tree for which jackknife supports will be calculated is determined by the lident value specified (`individual`, `consensus` or `STRING`). Additional arguments include:

`remove:FLOAT` The value of the argument `remove` specifies the percentage of characters being deleted during a pseudoreplicate. The default of `remove` is `36` percent.

`resample:INTEGER` The value of the argument `resample` specifies the number of resampling pseudoreplicates. The default of `resample` is `5`.

Search strategy The calculation of the support values requires a local search, that is performed under the default settings unless the values of the following arguments are specified.

`build` For calculating Bremer support, the integer value of `build` specifies the number of independent Wagner tree builds per node. The integer value `0` (`build:0`) specifies that Bremer support values are calculated on the starting trees currently in memory, rather than on newly generated trees. The initial trees for calculating Bremer support can also be imported using the argument `of_file` of the command `build` (Section 3.3.1).

For calculating Jackknife and Bootstrap supports, `build` specifies the number of Wagner tree builds per pseudoreplicate. Single best trees from all pseudoreplicates are used to calculate the support values. If multiple best trees are recovered in a pseudoreplicate, one is selected. If `build` is omitted from the argument list of `calculate_support`, a single random addition Wagner tree per pseudoreplicate is built by default. This is equivalent to `build(trees:1,randomized)`. See `build` (Section 3.3.1) for a detailed discussion of arguments of the command `build`.

`swap` Specifies the method and parameters for local tree search. If search parameters are not specified, the search is performed under the default settings of `swap` (Section 3.3.26).

Defaults

`calculate_support(bremer,build(trees:1,randomized),swap(trees:1))`
By default `POY5` will calculate the Bremer support for each tree in memory node by node. However, if no trees are stored in memory, executing the command `calculate_support()` does not have any effect.

Examples

- `calculate_support(bremer)`
  Calculates Bremer support values by performing independent searches for every node for every tree in memory. This is equivalent to executing `calculate_support()`, the default setting.

- `calculate_support(bremer,build(trees:0),swap(trees:2))`
  Calculates Bremer support values by performing swapping on each tree in memory for every node and keeping up to two best trees per search round.

- `calculate_support(bremer,build(of_file:"new_trees"), swap(tbr,trees:2))`
  Calculates Bremer support values by performing TBR swapping on each tree in the file `new_trees` located in the current working directory for every node and keeping up to two best trees per search round.

- `calculate_support(bootstrap)`
  Calculates Bootstrap support values under default settings. This command is equivalent to `calculate_support(bootstrap:5,build(trees:1, randomized),swap(trees:1))`.

- `calculate_support(bootstrap:100,build(trees:5),swap(trees:1))`
  Calculates Bootstrap support values performing one random resampling with replacement, followed by 5 Wagner tree builds (by random addition sequence) and swapping these trees under the default settings of the command `swap`, and keeping one minimum-cost tree. The procedure is repeated 100 times.

- `calculate_support(jackknife)`
  Calculates Jackknife support values under default settings. This command is equivalent to `calculate_support(jackknife:(resample:5, remove:36),build(trees:1,randomized),swap(trees:1))`.

- `calculate_support(jackknife:(resample:1000,remove:25), build(100),swap(tbr,trees:5))`
  Calculates Jackknife support values randomly removing 25 percent of the characters, building 100 Wagner trees by random addition sequence, swapping these trees using `tbr`, and keeping up to 5 minimum-cost tree in the final swap per swap (totaling up to 500 stored trees per replicate). The procedure is repeated 1000 times.

See also

- `report` (Section 3.3.19)
- `supports` (Section 3.3.19)
- `graphsupports` (Section 3.3.19)

3.3.3 clear_memory

Syntax

clear_memory([argument list])

Description

Frees unused memory. Rarely needed, this is a useful command when the resources of the computer are limited. The arguments are optional and their order is arbitrary.

Arguments

`m` Includes the alignment matrices in the freed memory.

`s` Includes the unused pool of sequences in the freed memory.

Defaults

`clear_memory()` By default `POY5` clears all memory *except* for the pool of unused sequences and the matrices used for the alignments.

Examples

- `clear_memory(s)`
  This command frees memory including all alignment matrices but keeping unused pool of sequences.

See also

- `wipe` (Section 3.3.30)

3.3.4 cd

Syntax

cd(STRING)

Description

Changes the working directory of the program. This command is useful when data files are contained in different directories. It also eliminates the need to navigate into the working directory before beginning a `POY5` session.

With the *Interactive Console*, the path of the directory can be completed by dragging and dropping the icon of the directory into the terminal window of this interface.

To display the path of the current directory, use the command `pwd` (Section 3.3.12).

Arguments

`STRING` The value specifies a path to a directory.

Examples

- `cd("/Users/username/docs/poyfiles")`
  Changes the current directory to the directory *poyfiles* in a Mac OSX environment. Filenames with spaces between words need to be escaped, e.g. "poy files" should be typed as "poy\ files". When using a PC, the forward slashes should be replaced with backslashes.

See also

- `pwd` (Section 3.3.12)

3.3.5 echo

Syntax

echo`(STRING, output class)`

Description

Prints the content of the string argument into a specified type of output. Several types of output are generated by `POY5` which are specified by the "output class" of arguments (see below). If no output-class arguments are specified, the command does not generate any output.

Arguments

Output class

`error` Outputs the specified string as an error message (`stderr` in the *flat* interface).

`info` Outputs the specified string as an information message (`stderr` in the *flat* interface).

`output[:STRING ]` Reports a specified string (`stdout` in the *flat* interface) to screen or file, if the filename string (enclosed in parentheses) is specified following `output` and separated from it by a colon, ":".

Examples

- `echo("Building_with_indel_cost_1",info)`
  Prints to the output window in the *ncurses* interface and to the standard error in the *flat* interface the message `Building_with_indel_cost_1`.

- `echo("Final_trees",output:"trees.txt")`
  Prints the string `Final_trees` to the file `trees.txt`.

- `echo("Initial_trees",output)`
  Prints the string `Initial_trees` to the output window in the *ncurses* interface, and to the standard output (`stdout` in the *flat* interface).

See also

- `report` (Section 3.3.19)

3.3.6 exit

Syntax

exit`()`

Description

Exits a `POY5` session. This command does not accept any argument. The `exit` command is equivalent to `quit`.

NOTE

To interrupt a process without quitting a POY5 session, use **control-c**. It aborts a currently running operation but keeps all the previously accumulated data in memory. It does not abort the current session permitting the entry of new commands and continuing the session.

Examples

- `exit()`
  Quits the program.

See also

- `quit` (Section 3.3.13)

3.3.7 fuse

Syntax

fuse`([argument list])`

Description

Performs Tree Fusing [20] on the trees in memory. Tree Fusing can be used to escape local optima by exchanging clades with identical composition of terminals, differing in arrangement between pairs of trees. Only *one* exchange between pairs of trees is evaluated during a single iteration.

Arguments

`iterations:INTEGER` Specifies the number of iterations of tree fusing to be performed. The number of iterations is effectively the number of pairwise clade exchanges. The default number of iterations is four times the number of retained trees (as specified by `keep`).

`keep:INTEGER` Specifies the maximum number of trees to keep between iterations. By default, the number of trees retained is the same as the number of starting trees.

`optimize(model[:LIDENT ],branch[:LIDENT ])` Specifies when the likelihood model and how the branches are optimized during the fuse routine. These options are also available in the `build()` and `swap()` commands.

In all cases a complete round of optimization will occur after the completion of a build.

`model:always` Optimize the model after every join.

`model:never` Do not optimize the model during the fuse (the default).

`branch:all_branches` Optimize all branch lengths on each join.

`branch:join_region` Optimize a maximum of five branches; the new edge, and the two edges on either side (the default).

`branch:never` Do not optimize the branches during the fuse process. Estimates are made based on the proportion of sites that would undergo a transformation.

`replace:LIDENT` Specifies the method for tree selection. Acceptable arguments are:

`best` Keeps a set of trees of the best cost regardless of their origin.

`better` Replaces parent trees with trees of better cost produced during a fusing iteration.

The default is `best`.

`swap` Specifies tree swapping strategy to follow each iteration of tree fusing. No swapping is performed under default settings. See the description of the command `swap` (Section 3.3.26).

Defaults

`fuse(replace:best)` By default `POY5` performs fusing, keeping the same number of trees per iterations as the number of the starting trees. The number of iterations is four times the number of starting trees. During this procedure, only the best trees are retained. No swapping is performed subsequent to tree fusing.

Examples

- `fuse(iterations:10,replace:best,keep:100,swap())`
  This command executes the following sequence of operations. In the first iteration, clades of the same composition of terminals are exchanged between two trees from the pool of the trees in memory. The cost of

the resulting trees is compared to that of the trees in memory and a subset of the trees containing up to 100 trees of best cost is retained in memory. During each iteration of fuse, the trees are subjected to swapping under the default settings of `swap`. The entire procedure is repeated nine more times.

- `fuse(optimize:(model:never,branch:join_region))`
  This command performs tree fusing, specifying that the likelihood model is never optimized after each round of fusing , but that a maximum of five branches are optimized each round.

- `fuse(swap(constraint))`
  This command performs tree fusing with modified settings for swapping that follows each iteration. Once a given iteration is completed, a consensus tree of the files in memory is computed and used as constraint file for subsequent rounds of swapping (see the argument `constraint` (Section 3.3.26) of the command `swap`).

See also

- `swap` (Section 3.3.7)

3.3.8 help

Syntax

help`([argument])`

Description

Reports the requested contents of the help file on screen.

Arguments

`LIDENT` Reports the description of the command, the name of which is specified by the LIDENT value.

`STRING` Reports every occurrence in the help file of the expression specified by the string value.

Defaults

`help()` By default `POY5` displays the entire content of the help file on screen.

Examples

- `help(swap)`
  Prints the description of the command `swap` in the *POY Output* window of the *ncurses* interface or to the standard error in the *flat* interface.

- `help("log")`
  Finds every command with text containing the substring `log` and prints them in the *POY Output* window of the *ncurses* interface or to the standard error in the *flat* interface.

3.3.9 inspect

Syntax

inspect`(STRING)`

Description

Retrieves the description of a `POY5` file produced by the command `save` (Section 3.3.21). If the description were not specified by the user, `inspect` reports that the description is not available. If the file is not a proper `POY5` file format, a message is printed in the *POY Output* window of the *ncurses* interface or to the standard error of the *flat* interface.

These `POY5` files are not intended for permanent storage. They are recommended for temporary storage of a `POY5` session, checkpointing the current state of the search (to avoid losing data in case the computer or the program fails), or reporting bugs. `POY5` also automatically generates `POY5` files in cases of terminating errors (an important exception is an out-of-memory error).

Examples

- `inspect("initial_search.poy")`
  Prints the description of the `POY5` file `initial_search.poy` located in the current working directory in the *POY Output* window of the *ncurses* interface or to the standard error in the *flat* interface. If, for example, the file was saved using the command `save ("initial_search.poy","Results_of_Total_Analysis")`, then the output message is: `Results_of_Total_Analysis`.

See also

- `save` (Section 3.3.21)
- `load` (Section 3.3.10)
- `cd` (Section 3.3.4)
- `pwd` (Section 3.3.12)

3.3.10 load

Syntax

load`(STRING)`

Description

Imports and inputs `POY5` files created by the command `save`. The name of the file to be loaded is included in the string argument. All the information of the current `POY5` session will be replaced with the contents of the `POY5` file. If the file is not in proper `POY5` file format, an error message is printed in the *POY Output* window of the *ncurses* interface, or the standard error in the *flat* interface. See the description of the command `save` (Section 3.3.21) on the `POY5` file and its usage.

Thses `POY5` files are not intended for permanent storage: they are recommended for temporary storage of a `POY5` session, checkpointing the current state of the search (to avoid losing data in case the computer or the program fails), or reporting bugs. `POY5` also automatically generates `POY5` files in cases of terminating errors (an important exception is out-of-memory error).

Examples

- `load("initial_search.poy")`
  Reads and imports the contents of the `POY5` file `initial_search.poy`, located in the current working directory.

See also

- `save` (Section 3.3.21)
- `inspect` (Section 3.3.9)
- `cd` (Section 3.3.4)

- `pwd` (Section 3.3.12)

3.3.11 perturb

Syntax

perturb`([argument list])`

Description

Performs branch swapping on the trees currently in memory using temporarily modified ("perturbed") characters. Once a local optimum is found for the perturbed characters, a new round of swapping using the original (non-modified) characters is performed. Subsequently, the costs of the initial and final trees are compared and the best trees are selected. If there are n trees in memory prior to searching using `perturb`, then the n best trees are selected at the end. For example, if there are 20 trees currently in memory, 20 individual `perturb` procedures will be performed (each procedure starting with one of the 20 initial trees), and 20 final trees are produced.

This command allows for movement from a local search optimum in the tree space by *perturbing* the character space (hence the name). The arguments specify the type of perturbation (`ratchet`, `resample`, and `transform`), the parameters of the subsequent search (`swap`), and the number of iterations of the `perturb` operation (`iterations`).

No new Wagner trees are generated following the perturbation of the data; the search is performed by local branch swapping (specified by `swap`). If `perturb` is executed with no trees in memory, an error message is generated. The arguments of `perturb` are optional and their order is arbitrary.

Arguments

`iterations:INTEGER` Repeats (iterates) the `perturb` procedure the number of times specified by the integer value. The number of iterations is reported in the *Current Job* window of the *ncurses* interface and to the standard error in the *flat* interface.

`ratchet[:(FLOAT,INTEGER)]` Perturbs the data by implementing a variant of the parsimony ratchet [36] by reweighting characters listed in `report(data)`. For unaligned data, the `ratchet` randomly selects and reweights a fraction of sequence fragments (*not* individual nucleotides) specified by the float (decimal) value, upweighted by a factor specified

by the integer value (severity). Thus, the number of sequence fragments into which the data is partitioned will impact the effectiveness of using the ratchet on dynamic character matrices. For static matrices, such as those obtained using the command `transform` (Section 3.3.27), `ratchet` randomly selects and reweights individual nucleotide positions (column vectors), as in Nixon's original implementation [36]. Under default settings, `ratchet` selects 25 percent of characters and upweights them by a factor of 2. Unless `ratchet` is performed under default settings (that does not require the specification of the fraction of data to be reweighted and the severity value), both values must be specified in the proper order and separated by a comma. This argument is only used as an argument for `perturb`.

`resample:INTEGER` Resamples the characters in random order with replacement. The `INTEGER` specifies the number of characters to be resampled. No default settings are available for `resample`. This command is only used as an argument of `perturb`.

`swap` Specifies the method of branch swapping for a local tree search based on perturbed data. If the argument `swap` is omitted, the search is performed under default settings of the command `swap` (Section 3.3.26).

`transform` Specifies a type of character transformation to be performed *before* executing a `perturb` procedure. See the command `transform` (Section 3.3.27) for the description of the methods of character type transformations and character selection.

Defaults

`perturb(ratchet:(0.25,2),iterations:1,swap(trees:1))` When no arguments specified, `POY5` performs the ratchet procedure under default settings.

Examples

- `perturb(resample:50,iterations:10)`
  Performs 10 successive repetitions of random resampling of 50 characters with replacement. Branch swapping is performed using alternating SPR and TBR, and and keeping one minimum-cost tree (the default of `swap`).

- `perturb(iterations:20,ratchet:(0.18,3))`
  Performs 20 successive repetitions of a variant of the ratchet (see above)

by randomly selecting 18 percent of the characters (sequence fragments) and upweighting them by a factor of 3. Branch swapping is performed using alternating SPR and TBR, and keeping one optimal tree (the default of `swap`).

- `perturb(iterations:1,transform(tcm:(4,3)))`
  Transforms the cost regime of all applicable characters to the new cost regime specified by `transform` (cost of substitution 4 and cost of indel 3). Subsequently a single round of branch swapping is performed using alternating SPR and TBR, and and keeping one optimal tree (the default of `swap`).

- `perturb(ratchet:(0.2,5),iterations:25,swap(tbr,trees:5))`
  Performs 25 successive repetitions of a variant of the ratchet (see above) by randomly selecting 20 percent of the characters (sequence fragments) and upweighting them by a factor of 5. Branch swapping is performed using TBR and keeping up to 5 optimal trees in each iteration.

- `perturb(transform(static_approx),ratchet:(0.2,5),iterations:25,swap(tbr,trees:5))`
  Transforms all applicable (i.e. dynamic homology sequence characters) using `transform` into static characters. Therefore, the subsequent ratchet is performed at the level of individual nucleotides (as in the original implementation), *not* sequence fragments. Thus, ratchet is performed by selecting 20 percent of the characters (individual nucleotides) and upweighting them by a factor of 5. Branch swapping is performed using TBR and keeping up to 5 optimal trees in each iteration as in the example above.

See also

- `swap` (Section 3.3.26)
- `transform` (Section 3.3.27)

3.3.12 pwd

Syntax

pwd()

Description

Prints the current working directory in the *POY Output* window of the *ncurses* interface and the standard error (stderr) of the *flat* interface. The command `pwd` does not have arguments. The default working directory is the shell's directory when `POY5` started.

Examples

- `pwd()`
  This command generates the following message: "`The current working directory is /Users/username/datafiles/`". The actual reported directory will vary depending on the directory of the shell when `POY5` started, or if it has been changed using the command `cd()`.

See also

- `cd` (Section 3.3.4)

3.3.13 quit

Syntax

quit`()`

Description

Exits `POY5` session. This command does not have any arguments `quit` is equivalent to the command `exit`.

NOTE

To interrupt a process without quitting a `POY5` session, use control-c. It aborts a currently running operation but keeps all the previously accumulated data in memory. It does not abort the current session, thereby permitting the entry of new commands and continuing the session.

Examples

- `quit()`
  Quits the program.

See also

- exit (Section 3.3.6)

3.3.14 read

Syntax

read([argument list])

Description

Imports data files and tree files. Supported formats include ASN1, Clustal, FASTA, GBSeq, Genbank, Hennig86, Newick, NewSeq, Nexus, PHYLIP, POY3, TinySeq, and XML. Filenames must be enclosed in quotes and, if multiple filenames are specified, they must be separated by commas. All filenames read into POY5 must include the appropriate suffix (e.g. .aln, .fas, .fasta, .ss, .tre). The exclusion of these suffixes will result in an error such as "Sys_error ("No such file or directory")". The filename must match exactly.

NOTE

POY5 provides an option that allows the commenting out of 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 Ablepharus_budaki$_AY561421_16S will comment out this information and the taxon name will be read as Ablepharus_budaki by the program.

read automatically detects the type of the input file. This command can also use wildcard expressions (such as *) to refer to multiple files in a single step. For example, read("*.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. Moreover, importing all files that begin with the filename BAP is achieved by typing read("BAP.*"). Specifying a filename(s) is obligatory: an empty argument string, read(), results in no data being read by POY5. The list of imported files and their content can be reported on screen or to a file using report(data).

If a file is loaded twice, POY5 will issue an error message, but this will not interfere with subsequent file loading and execution of commands.

NOTE

When running a script that includes reading in trees from a previous analysis, these trees **must** be read in **after** the build stage. If the trees are read in before the build they will be replaced by the trees generated during the build.

`POY5` automatically reports in the *POY Output* window of the *ncurses* interface or to the standard error in the *flat* interface the names of the imported files, their file type, and a brief description of their contents. A more comprehensive report on the contents of the imported files can be requested (either on screen or to a file) using the argument `data` of the command `report` (Section 3.3.19).

Arguments

Data file types To import data files, individual data file names must be included in the list of `read` arguments, enclosed in quotes, and separated by commas. If no data file types are specified, the types of the imported files are recognized automatically. To specify the data type, an additional argument explicitly denoting the data type, is included; it is followed by a colon (":") and the list of data file names (enclosed in parentheses), separated by commas and enclosed in quotes. This format prevents any ambiguity in importing multiple data file types simultaneously (i.e. included in an argument list of a single `read`) command.

NOTE

Although `POY5` recognizes multiple data file formats, it does not interpret all of their contents. Instead, it will recognize and import only character data and ignore other content (such as blocks of commands, *etc.*). For certain data file formats, `POY5` will interpret additional information as detailed for each file type below. It is important, however, to verify that the data was interpreted properly (using the command `report`).

NOTE

Unlike many phylogenetic programs, POY5 does not clear the memory upon reading a second file. Instead, any subsequently read files will be added to the total data being analyzed. If a *new* taxon appears in a file, then it is be assigned missing data for all previously loaded characters. If a taxon does *not* appear in a file, missing data are assigned for the characters of these taxa.

To eliminate the imported data and then to input a new data the `wipe()` command must be issued first.

NOTE

If one of the terminal names in an imported data file contains a space, " ", POY5 issues a warning. It is therefore advisable to format taxon names in the data files, such that any space is r eplaced by an underscore, e.g. `Rhacodactylus_ciliatus`. A warning is also issued if a taxon name appears to match a nucleotide sequence. If one of the terminal names in an imported molecular file contains a percentage ("%") or an at ("@") symbol, the file will not be loaded because it may cause failure when reporting results.

Basic data types This set of arguments covers the importing of all data files (except `chromosome`, `genome`, `custom_alphabet` and `breakinv`), as well as tree files in parenthetical notation.

`aminoacids:(STRING list)` Specifies that the data listed in the string argument are amino acid sequences in FASTA format.

NOTE

Currently, IUPAC ambiguity codes for amino acids are *not* supported other than for `X` and inputing files that contain amino acid data with ambiguities results in an error message.

`nucleotides:(STRING list)` Specifies that the data in the list of files hold nucleotide sequences in FASTA format. The sequences can be divided into smaller fragments using a pound sign ("#"), and each fragment is treated as an individual character.

NOTE

POY5 recognizes the characters x and n as representing any nucleotide base (a, c, g, or t). The ? symbol inserted in sequence data signifies missing data, a gap, or any nucleotide base may occur in that matrix position. For prealigned data sequence gaps are specified by dashes.

NOTE

Continuous characters can be treated as such by assigning the lower and upper bounds of the range as polymorphic additive character states [21]. Although they will be optimized simultaneously with all other characters, continuous characters must be scored in a separate Hennig86 format matrix with the heading "nstates cont"—an example of this file format (ccm.ss) is available at the POY5 website and is included into the POY5 installation package. http://www.amnh.org/our-research/computational-sciences/research/projects/systematic-biology/poy. Consider a continuous character winglength, the states of which are ranges of measurements in hundredth of a millimeter, for example 2.53-3.68 mm for a given terminal. A corresponding character state in the additive character matrix (in Hennig86 format) is [253,368]. Because additive characters are integers, such characters need to be re-scaled using the weightfactor argument of transform. To scale the values, a transformation is applied to the character winglength as follows: transform(names:("winglength"),(weightfactor:0.01)).

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 (as printed by pwd()). The file type is recognized automatically. Molecular files are assumed to contain nucleotide sequences. Valid files to read using this command are: tree files using parenthetical notation (Newick, POY5 trees), Hennig86 files, Nona files, Sankoff character files as used in POY 3, FASTA files (and virtually any file generated by Genbank), and Nexus files. Only taxon names, trees, characters, and cost regimes will be imported from each one of this files, no other commands are currently recognized.

Chromosome and genome type characters This set of arguments governs characters that are either multi-locus nucleotides sequences (`chromosome`) or multi-locus, multi-chromosomal nucleotide sequences (`genome`). Chromosome sequences can be `annotated` or unannotated.

`annotated:(STRING list)` Specifies that the data listed in the string argument are chromosomal sequences with pipes (" | ") separating individual loci. This data type allows for locus-level rearrangements specified by the command `transform` (Section 3.3.27). Locus homologies are determined dynamically, but based on annotated regions [57]. (For a sample script using this data type see tutorials `5.9` and `5.10`.

`chromosome:(STRING list)` Specifies that the data in the files listed in the string argument are chromosomal sequences without predefined locus boundaries, i.e. unannotated chromosomes. Specifying that imported sequences are chromosome type data enables the application of parameter options that optimize chromosome-level events such as rearrangements, inversions, and large-scale insertions and deletions (including duplications). These parameter options (e.g. inversion cost) are specified using the command `transform` (Section 3.3.27). Unlike when using `annotated` data type, both locus-level and nucleotide-level homologies are determined dynamically [10, 56] (see tutorial `5.8`). If chromosome sequences are imported as nucleotide type data, they can be converted to chromosome type data using the `seq_to_chrom` argument of `transform` (Section 3.3.27).

`genome:(STRING list)` Specifies that the data listed in the string argument are multi-chromosomal nucleotide sequences with an at sign ("@") separating individual chromosomes. This data type allows for chromosome-level rearrangements which are specified by the command `transform` (Section 3.3.27). Chromosome homologies are determined using the Mauve aligner [10] within the command `transform` (Section 3.3.27). [Note: for genome character types, it is only possible to separate the individual chromosomes and not the loci within these chromosomes. A sample script using this data type can be found in tutorial `5.11`.]

Custom alphabet type characters This set of arguments are for characters are those that employ a user-specified alphabet. These include characters of the custom alphabet, as well as break inversion type.

`breakinv:(STRING list,tcm:(STRING),[LIDENT list])` An enhancement of the data file type `custom_alphabet` (see below), allowing rearrangement events. Syntactically, `breakinv` data type is identical to the `custom_alphabet` data type. Three optional arguments are possible (`LIDENT` list): `level`; `init3d`; and `tiebreaker`. These three arguments can be used in conjunction with both `breakinv` and `custom_alphabet` character types (see the argument `custom_alphabet` (Section 3.3.14) below for a description of these arguments). Specifying that imported sequences are `breakinv` type data enables the application to calculate either locus breakpoint or locus inversion costs to these data. These parameter options are specified using the command `transform` (Section 3.3.27).

NOTE

Break Inversion characters differ from custom alphabet characters, in that orientation of the alphabet characters can be specified with Break Inversion characters. A tilde ("∼") symbol preceding an alphabet character indicates the negative orientation.

`custom_alphabet(:STRING list,tcm:(STRING),[LIDENT list])` Reads the data in the user-defined alphabet format. The first string argument is the name of a data file(s) that contains custom-alphabet sequences in FASTA format. The characters can be (but are not required to be) separated by spaces. An example of a corresponding input file follows:

```
>Taxon1
alphabetagammadelta
>Taxon2
alphabetabetagammadelta
>Taxon3
alphabetabetadelta
```

The `tcm` refers to the custom-alphabet matrix that contains two parts: an alphabet itself, where the alphabet elements are separated by spaces, and a transformation cost matrix. The elements in an alphabet can be letters, digits, or both, as long as one element is not a prefix of another ("prefix-free"). For example, the following pairs of custom-alphabet

elements are *not* valid because the first is a prefix of the second (which would prevent the proper parsing of an input file): `AB` and `ABBA` or `122` and `122X`. The transformation cost matrix contains the rows and columns in which the positions from left to right and top to bottom correspond to the sequence of the elements as they are listed in the alphabet. An extra rightmost column and lowermost row correspond to a gap. It is important that the cost matrix be symmetrical. An example of a valid custom alphabet input 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 costs `5`.

Three optional arguments of `custom_alphabet` are possible ([`LIDENT` list]): `init3d`; `level`; and `tiebreaker`.

`init3d:BOOL` This argument requires an obligatory boolean value, namely `true` or `false`. `init3d` initiates a 3D matrix, but the user should be aware that this option can consume a great deal of memory.

`level:(INTEGER,LIDENT)` This argument determines the heuristic `level` of the median sequence calculation. The user must define both the level itself, as specified by the `INTEGER`, and the keep method, as specified by the `LIDENT` (`first`, `last`, or `at_random`. If the `LIDENT` is `first`, ties are broken (if the number of equally costly states is greater than the level number) by choosing the first median state examined; if `last`, the last state, and if `at_random` then uniformly at random. The default `level` is `2` and keep method is `first`. The maximum `level` of any dataset is equal to the alphabet size + 1.

`tiebreaker:LIDENT` This argument determines how ties among median states are chosen: `first`, `last`, and `at_random`. If `first` is chosen, then the ties are broken (if the number of equally costly states is greater than the level number) by choosing the first median state examined; if `last`, the last state, and if `at_random`

then uniformly at random. The default choice method is `first`.

NOTE

As a rule, transformation cost matrices are employed at the `transform` stage of the analysis. With `prealigned` and `custom_alphabet` characters however, the cost matrix (`tcm`) needs to be read in along with the data files.

Prealigned data This set of arguments specifies how certain characters, namely sequences, amino acids and custom alphabet characters, are read as prealigned. Prealigned data files must be of the same length. Because these data are prealigned, affine gap costs can not be applied.

`prealigned:(LIDENT:(STRING list)[,tcm:STRING ])` Specifies that the data indicated in the `STRING`, of the type identified by the `LIDENT` (i.e. `aminoacid(s)`, `custom_alphabet` or `nucleotides`) are prealigned. A transformation cost matrix, as defined in the tcm `STRING` argument can be specified. If these are not specified, the default cost of tcm:(1,1) will be assigned. (See the argument `tcm` (Section 3.3.27) of the command `transform`.)

NOTE

By default, upon importing prealigned sequence data, all the gaps are removed and the sequences are treated as dynamic homology characters. To preserve the alignment the data must be imported using the `prealigned` argument of the command `read`.

`prealigned:(LIDENT:(STRING list)[,tcm:(INTEGER,INTEGER)])` Specifies that the input sequences are prealigned and should be assigned substitution and indel costs as defined by the `tcm` integers `(INTEGER,INTEGER))`. (See the argument `tcm` (Section 3.3.27) of the command `transform`.)

Defaults

`read()` If no data files are specified, `POY5` does nothing. If however, data files are listed but character type is not indicated, `POY5` automatically detects data file types and interprets sequence files as nucleotides-type data.

Examples

- `read("/Users/andres/data/test.txt")`
  Reads the file `test.txt` located in the path `/Users/andres/data/`.

- `read("28s.fas","initial_trees.txt")`
  Reads the file `28s.fas` and loads the trees in parenthetical notation of the file `initial_trees.txt`.

- `read("SSU*","*.txt")`
  Reads all the files with names starting with `SSU`, and all the files with the extension `.txt`. The types of the data files are determined automatically.

- `read(nucleotides:("chel.FASTA","chel2.FASTA"))`
  Reads the files `chel.FASTA` and `chel2.FASTA`, containing nucleotide sequences.

- `read(aminoacids:("a.FASTA","b.FASTA","c.FASTA"))`
  Reads the amino acid sequence files `a.FASTA`, `b.FASTA`, and `c.FASTA`.

- `read("hennig1.ss","chel2.FASTA",aminoacids:("a.FASTA"))`
  Reads the Hennig86 file `hennig1.ss`, the FASTA file `chel2.FASTA` containing nucleotide sequences, and the amino acid sequence file `a.FASTA`. script

- `read(annotated:("ch1.txt","ch2.txt"),chromosome:("ch3.txt"))`
  Reads three files containing chromosome-type sequence data. The sequences in two files, `ch1.txt` and `ch2.txt`, contain pipes (" | ") separating individual loci, whereas the sequences in the third (`ch3.txt`), are without predefined boundaries. [Note: see tutorial `5.10`, which illustrates the transformation of these two data types in the same analysis.]

- `read(genome:("mt_genomes","nu_genomes"))`
  Reads two files containing genomic (multi-chromosomal) sequence data.

- `read(breakinv:("BI_run1",tcm:"alphabet",level:1,tiebreaker:first)`
  Reads the file, `BI_run1`, which contains data in custom alphabet format. This data is defined in the `tcm` file, `alphabet`. The heuristic level of median sequence calculation is set to 1. If ties among median states are encountered, the `first` will be chosen.

- `read(custom_alphabet:("CA_run1",tcm:"alphabet",level:2, tiebreaker:last))`
  Reads the file, `CA_run1`, which contains data in custom alphabet format. This data is defined in the `tcm` file, `alphabet`. The heuristic level of median sequence calculation is set to 2. If ties among median states are encountered, the `last` will be chosen.

- `read(prealigned:("18s.aln",tcm:(1,2)))`
  Reads the prealigned data file `18s.aln` which was generated from the nucleotide file `18s.FASTA` using the the transformation costs `1` for substitutions and `2` for indels.

- `read(prealigned:(nucleotides:("*.nex"),tcm:"matrix1"))`
  Reads character data from all the Nexus files as prealigned data using the the transformation cost matrix from the file `matrix1`.

See also

- `report` (Section 3.3.19)

3.3.15 recover

Syntax

recover`()`

Description

Recovers the best trees found during swapping, even if the swap were cancelled. This command functions only if the argument `recover` (Section 3.3.26) were included in a previously executed (in the current `POY5` session) command `swap`. Otherwise, it has no effect.

The trees imported by `recover` are appended to those currently stored in memory.

Note that using recovered trees is not intended for temporary storage of trees. It is useful only as an intermediary operation in a given part of a `POY5` session. When other commands that require clearing memory are executed (such as `build`, `calculate_support`, or another `swap`), the trees stored by `recover` can no longer be retrieved.

Examples

- `recover()`
  If the command `swap` (executed earlier in the current `POY5` session) contained the argument `recover`, for example, `swap(tbr,recover)`, this command will restore the best trees recovered during swapping.

See also

- `swap` (Section 3.3.26)

3.3.16 rediagnose

Syntax

rediagnose`()`

Description

Performs a re-optimization of the trees currently in memory. This function is useful for sanity checks of the consistency of the data. Its main usage is for the `POY5` developers. This command does not have arguments.

Arguments

`clear` Specific for likelihood characters, this rediagnoses the tree, clearing the optimized model parameters and branch lengths. Additional optimizations are performed after diagnosis.

`preserve` Performed during a likelihood search, this rediagnoses the tree, keeping the current model parameters and branch lengths. Additional optimizations are performed after diagnosis.

Examples

- `rediagnose()`
  See the description of the command.

3.3.17 redraw

Syntax

redraw`()`

Description

Redraws the screen of the terminal. This command is only used in the *ncurses* interface, other interfaces ignore it. `redraw` clears the contents of the *Interactive Console* window but retains the contents of the other windows. It does not affect the state of the search and the data currently in memory.

Examples

- `redraw()`
  See the description of the command.

3.3.18 rename

Syntax

rename`([argument list])`

Description

Replaces the name(s) of specified item(s) (characters or terminals). This command allows for substituting taxon names and helps merging multiple datasets without modifying the original data files. More specifically, it can be used, for example, (1) for housekeeping purposes, when it is desirable to maintain long verbose taxon names (such as catalog or GenBank accession numbers) associated with the original data files but avoid reporting these names on the trees (although see the note on the usage of a "$" in the taxon name below); (2) to provide a single name for a terminal in cases where the corresponding data are stored in different files under different terminal names; and (3) to change an outdated or invalid terminal name.

The command consists of a terminal or character identifier followed by a comma and then by either a string containing a synonymy file or a pair (or pairs) of strings containing the names of items being renamed.

In order to change these names, the command `rename` must be executed ***before*** importing the data files (see `read` (Section 3.3.14)) that contain the taxa that are to be renamed.

NOTE

Once the command `rename` is applied, subsequent commands must refer to the terminals using the new, substitute names. This is critical, for example, when importing a terminals file using the command `select` (Section 3.3.23) or specifying a root using the command `set` (Section 3.3.24).

NOTE

`POY5` provides an option that allows the commenting out of 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 `Ablepharus_budaki$_AY561421_16S` will comment out this information and the taxon name will be read as `Ablepharus_budaki` by the program.

Arguments

Identifiers The identifiers specify whether terminals or characters are being renamed. An identifier must precede the subsequent arguments.

`characters` Specifies that the subsequently items to be renamed are characters.

`terminals` Specifies that the subsequently items to be renamed are terminals.

Specifying items to be renamed These arguments allow the user to specify the items to be renamed either in a group (by importing a *synonymy* file) or individually (by using a pair of string arguments). The former is useful when there are multiple items to be renamed and/or when it is desirable to substitute a single name for multiple ones.

`STRING` Specifies the name of the file (a *synonymy* file) that contains the list of terminals or characters to be renamed. The synonymy file has the following structure: each line contains a list of synonyms (two or more) separated by spaces. The name of the item listed first will be substituted for all the subsequently listed names. Consider, for example, the synonymy file below:

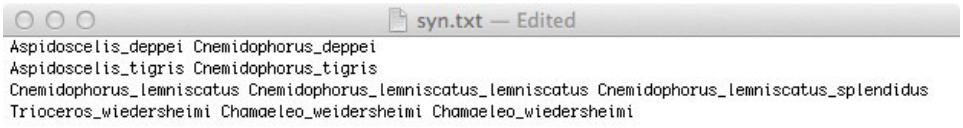

Figure 3.3: A synonymy file containing lists of terminal taxa to be remained.

When this file is imported using `rename`, the taxon Cnemidophorus_deppei will be remained as Aspidoscelis_deppei and both Chamaeleo_weidersheimi and Chamaeleo_wiedersheimi will be renamed as Trioceros_wiedersheimi etc:

`(STRING, STRING)` Specifies the names of individual items to be renamed. The first item (character or taxon) is renamed as the second item. Examine the renaming script below:

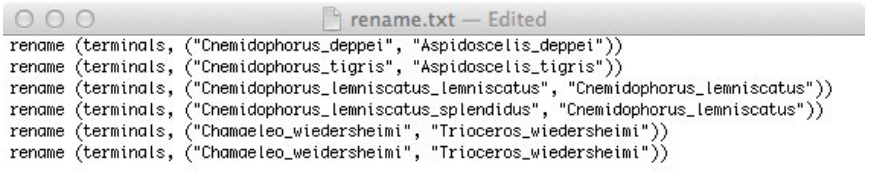

Figure 3.4: A example of a “renaming" script that is run prior to importing the data files.

The above script will perform the exact same renaming function as that of the previous example (Figure 3.3). Generating scripts such as this are recommended when more than a single taxon needs to be renamed. This script is employed using the command `run`.

NOTE

Note that when `rename` is applied by specifying pairs of synonyms in the command's argument (`STRING`, `STRING`), the substitute name is listed *second*. This is in contradistinction to a synonymy file, where the substitute name appears *first* and is followed by one or more synonyms.

Examples

- `rename(terminals,"synfile")`
  This command renames terminal names contained in the synonymy file `synfile` in all subsequently imported data files.

- `rename(terminals,("Mytilus_sp","Mytilus_edulis"))`
  This command renames the terminal taxon `Mytilus_sp` as `Mytilus_edulis` in all subsequently imported data files.

- `rename(terminals,("Chamaeleo_weidersheimi","Trioceros_wiedersheimi"),("Chamaeleo_wiedersheimi","Trioceros_wiedersheimi"))`
  This command renames the terminal taxa `Chamaeleo_wiedersheimi` and `Chamaeleo_weidersheimi` (a misspelling of the previous name) as `Trioceros_wiedersheimi` in all subsequently imported data files.

3.3.19 report

Syntax

report`([argument list])`

Description

Outputs the results of current analysis or loaded data in the *POY Output* window of the *ncurses* interface, the standard output of the *flat* interface, or to a file. To redirect the output to a file, the file name in quotes and 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, as well as other miscellaneous arguments.

Arguments

Reporting to files

`new: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, the data, trees, and diagnosis are reported to that file by default. In this case, a new file is created or the previously existing file of the same name is overwritten.

`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, the data, trees, and diagnosis are reported to that file by default. By default files are appended to the report, rather than overwritten.

A string (text in quotes) argument is interpreted as a filename. Therefore, `"/Users/andres/results1.tre"` represents the file `results1.tre` in the directory `/Users/andres`. If no path is given, the path is relative to the current working directory as printed by `pwd()`.

Characters and terminals This set of arguments reports the current status of terminals and characters from the imported data files.

`cross_references[:identifiers[:STRING]]` Reports 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 not. It is highly recommended that the user report a `cross_references` file having imported the data into `POY5`. Not only is this argument is a **very** useful tool for visual representation of missing data, reporting all the data to a cross references file can also highlight inconsistencies in the spelling of taxon names in different data files.

Under default settings, cross-references are reported for all imported data files. To report cross-references for some of the fragments within a given file, a single character, or a subset of characters, optional arguments (`identifiers`) must be specified. A combination of a character identifier (see command `select` (Section 3.3.23)) and the file names (specified in the the string value) is used to select specific data files to be cross-referenced. For example, to only report information for `file1` type `cross_references:names:("file1")`.

The argument `cross_references:all` generates a table that shows presence and absence of fragments contained within each file. If each data file contains a single fragment, executing `cross_references:all` is equivalent to executing `cross_references`.

By default, the cross-reference table is printed on screen or to an output file, if specified. The reported cross references file is output as a plaintext document, which can then be imported into a spreadsheet application such as Microsoft Excel or Apple Numbers for easier viewing.

`data` Outputs a summary of the input data. More specifically, `POY5` will report the number of terminals to be analyzed, a list of included terminals with numerical identification, a list of synonyms (if specified), a list of excluded terminals, the number of included characters in each character-type category (i.e. additive, non-additive, Sankoff, and molecular) with the corresponding cost regimes, a list of excluded characters, and a list of input files. If the report is directed to a file with extension "nex" or "nexus" then the output is suitable for a nexus file (including the NEXUS header). Hennig format is produced if the report is directed to a file with extension "ss" or "hen" or "hennig".

`lkmodel` Reports the likelihood model, costs, and tree length for the characters in memory in a style similar to that of `PHYLIP`.

`searchstats` Outputs a summary of the results of the last search command, including the number of builds, fuses, ratchets, and the costs of the trees found.

`seq_stats:identifiers` Outputs a summary of the sequences specified in the argument value, for all taxa. The summary includes the maximum, minimum, and average length and distance for all terminals. In this case, identifiers include file names, characters, codes etc:

`terminals` Reports a list and number of terminals included and excluded per input file. Use the command `select` (Section 3.3.23) for including and excluding terminals.

`treestats` Reports the number of trees in memory for each cost.

`treecosts` Reports the cost of each tree separated by colons.

Diagnosis This argument will output the diagnosis.

`diagnosis` Outputs the diagnosis of each tree on screen or redirects it to a file, if specified. If the extension *.xml* is appended to the name of the output file, the diagnosis is reported in XML format, rather than in simple text format.

Exporting static homology data The following commands export the static homology characters currently in memory.

`nexus` Produces a file in the Nexus format that contains all the characters currently in memory. In order to export an implied alignment as a Nexus file, the characters must first be transformed into static characters using the `transform` command (see the Hennig86 example in tutorial `5.2`):

```
transform(all,(static_approx))
report("report.nexus",nexus,trees:(nexus))
```

NOTE

To generate a file that contains implied alignments only for a subset of fragments, an identifier must be included in the argument list of `transform`. For example,

```
transform(names:("fragment_1","fragment_2"),
(static_approx))
report("myfile.ss",phastwinclad)
```

will produce Hennig86 files only for `fragment_1` and `fragment_2`. The resulting file can be imported into other programs, such as WinClada. This is equivalent to the `phastwincladfile` command in `POY3`.

`phastwinclad` Produces a file in Hennig86 format that contains the additive and nonadditive characters currently in memory. In order to export an implied alignment as a Hennig86 file, the characters must first be transformed into static characters using the `transform` command (see example in tutorial `5.2`):

```
transform(all,(static_approx))
report("report.ss",phastwinclad)
```

Implied alignments This set of arguments outputs implied alignments [64].

`fasta:identifiers` The same as `implied_alignments` (Section 3.3.19) but no additional headers are added, producing a valid FASTA file. Intended for easy automation, by producing a file that other programs can read immediately.

`ia[:identifiers]` Synonym of `implied_alignments`.

`implied_alignments[:identifiers]` Outputs the implied alignments of the specified set of characters in FASTA format. The optional value of the argument specifies the characters included in the output, using the same identifiers described for the character specification in the entry for the command `select` (Section 3.3.23). If no characters are specified, then the implied alignment of all the sequence characters is generated. The output is reported on screen unless an output file (in parentheses) is specified, preceding the command name and separated from it by a comma. This argument is synonymous with the argument `ia`.

Trees This set of arguments outputs tree representations in parenthetical, ascii (simple text), or PDF formats. The arguments specify the types of tree outputs. They include actual trees resulting from current searches, or trees imported from files, their consensus trees, or trees displaying support values.

To select the root terminal in the tree representation, the command `set` (Section 3.3.24) is used.

Most analyses produce more than a single tree and it is often desirable to report only some of them. To report particular trees (for instance all optimal trees, randomly-selected trees, or all unique trees, *etc.*), first the command `select` (Section 3.3.23) must be applied to specify (select) the desired trees from all those stored in memory.

`all_roots` In a tree with n vertices (and therefore $n-1$ edges), calculates the cost of the $n-1$ rooted trees as implied by a root located in the subdivision vertex at each edge in the unrooted tree in memory.

`asciitrees[:collapse[:LIDENT ]]` Draws ascii character representations of trees stored in memory. The argument `collapse` will collapse branches on the basis of the `LIDENT` specified (see `collapse` (Section 3.3.19)).

`clades:STRING` Output a set of Hennig86 files. Each file, named `file.hen`, where "file" is whatever string you pass to this function contains information on each clade for one of the trees currently stored. This is similar to the utility jack2hen of `POY3`.

`consensus[:INTEGER ]` Reports the consensus of trees in memory in parenthetical notation. If no integer value is specified, a strict consensus is calculated [43]; if an integer value is specified, a majority rule consensus is computed, collapsing nodes with occurrence frequencies less than the

specified integer [32]. If a value less than `51` is specified, `POY5` reports an error.

`graphconsensus[:INTEGER ]` This argument is the same as `consensus` except that the trees are reported in graphical format, either in the ascii format on screen or in the PDF format if redirected to a file.

`graphdiagnosis` Output the diagnosis in PDF format. The PDF is compressed, and contains the trees and links to see the diagnosis of each vertex in the tree.

`graphsupports[:argument]` This command outputs a tree with support values that have been previously calculated using the `calculate_support` (Section 3.3.2) either on screen in ascii format, or, if specified, to a file in PDF format. The argument values are the same as for `supports` (i.e. `bremer`, `jackknife`, and `bootstrap`) (see below).

`graphtrees[:collapse[:LIDENT ]]` This argument is similar to `trees` except that the trees are reported in graphical format, either in the ascii format on screen or in the PDF format if redirected to a file. The argument `collapse` will collapse branches on the basis of the `LIDENT` specified (see `collapse` (Section 3.3.19)).

`supports[:argument]` Outputs a newick format representation of a tree with the support values has previously been calculated using the command `calculate_support` (Section 3.3.2), either to the screen or to a file (if specified). If no argument is given, all calculated support values are printed. The arguments `bremer`, `jackknife`, and `bootstrap` specify which type of support tree to report.

To print the Bremer supports of the trees in memory, using as reference trees that are stored in a file, `bremer` accepts an optional string argument (as in `report(supports:bremer:("file1.txt", "file2.txt"))`. The argument's value specifies the files containing lists of trees and costs (as those generated by `visited` (Section 3.3.26)), that should be used with their annotated cost to assign the Bremer support values.

To print the Bremer supports of a tree that does not exists in memory (or a consensus tree) stored in a file, `bremer` will accept the value `of_file:(STRING, INTEGER, files)`, where the first argument value (`STRING`) is the file containing the tree for which Bremer supports

should be computed, the second argument (`INTEGER`) is the cost of the tree, and the files is that described in the previous paragraph.

If no input file is given, or if `bootstrap` or `jackknife` are requested, then the necessary information must have been calculated using the argument `calculate_support` (Section 3.3.2). The arguments `jackknife` and `bootstrap` accept an optional argument with two possible values: `individual`, `consensus`, or a `STRING`.

The argument `individual` reports the support value for each tree held in memory: if there are a hundred trees stored in memory, for each one, the support values for each tree are reported. `consensus` generates a "consensus" tree, with the clades that have support higher than 50 percent. `STRING` labels the branches in the input trees contained in the input file located in the path of the `STRING` (e.g. to assign support values to the branches of a consensus tree). The default behavior, when no `individual` or `consensus` value is provided, is `individual`.

`trees:(argument list)` Outputs the trees in memory in parenthetical notation. The argument `trees` receives an optional list of values specifying the format of the tree that has to be generated. Unless `hennig` is specified in the list of values, `trees` uses newick format in the tree output. The valid optional arguments are:

`branches[:LIDENT ]` Reports a tree with likelihood or parsimony branch lengths included. The optional `LIDENT` values `single`, `min` or `max` are available: if `single` (the default), the branch length is based on the single assignment of HTU states; if `min`, the length is based on the minimum possible length; and if `max`, the length is based on the maximum possible length.

`collapse[:LIDENT ]` The degree of collapse is determined by the lident specified. If `single` or `true`, a collapse will occur if the length of the single assignment on a branch is zero; if `min` the branches are collapsed if the minimum length is zero–this can potentially over-collapse nodes; if `max` branches are collapsed if the maximum length is zero, i.e. identical, unambiguous character state reconstructions on a branch–this causes minimal collapsing; if `false`, no collapsing will occur. For parenthetical trees, the default is `false`, while for trees that are 'drawn', i.e. graphtrees or ascii trees, the default is `single` or `true`.

`hennig` Prepends the `tread` command to the list of trees and separates them with a star; this format is suitable for Hennig86, NONA, and TNT files.

`margin:INTEGER` Sets the margin width of the generated trees.

`newick` Outputs the trees in the Newick format, with the terminals separated with commas, and trees separated with semicolons.

`nexus` Outputs the trees in the Nexus format, inside a TREE block.

> **NOTE**
>
> The `hennig` and `newick` arguments are mutually exclusive.

`nomargin` Outputs the trees in a single line. This is useful for some programs (such as TreeView) that cannot read trees broken in several lines.

`total` Includes the total cost of a tree in square brackets after each tree.

If the report is directed to a file with extension "nex" or "nexus" then the output is suitable for a nexus file (trees inside a TREES block). Hennig format is produced if the report is directed to a file with extension "ss" or "hen" or "hennig". In these two cases, all other formatting options are ignored.

Other arguments

`ci` Calculates the ensemble consistency index (CI [13, 31]) for additive, and nonadditive characters. Dynamic homology characters are ignored in calculating the CI, therefore, the dynamic homology characters must be converted to static homology characters using the argument `static_approx` of the command `transform` (Section 3.3.27).

`memory` Reports on screen, the statistics of the garbage collector. For a precise description of each memory parameter, see the Objective Caml documentation.

`ri` Calculates the ensemble retention index (RI; [13]) for additive, and nonadditive characters. Dynamic homology characters are ignored in calculating the RI, therefore, the dynamic homology characters must be converted to static homology characters using the argument `static_approx` of the command `transform` (Section 3.3.27).

NOTE

To gauge the amount of time it takes POY5 to perform a command, setting a timer between commands is useful:

```
read("Biv.fas")
report(timer:"Load Time")
build(1)
report(timer:"Build Time")
```

the output will look something like:

```
Information:  Reading file Biv.fas of type input
sequences
Information:  The file Biv.fas contains sequences of
5 taxa, each sequence holding 1 fragment.
Status:  Loading Trees Finished
Load Time:  0.0128080844879
Status:  Wagner:  2 of 5 - Wagner tree with cost 54.
Status:  Wagner:  3 of 5 - Wagner tree with cost 77.
Status:  Wagner:  4 of 5 - Wagner tree with cost 82.
Status:  Wagner Finished
Status:  Building Wagner Tree Finished
Status:  Running Pipeline:  1 of 1 - Estimated
finish in 0 s
Status:  Running Pipeline Finished
Build Time:  0.14551615715.
```

In this example, the timer outputs information relating to the time it took POY5 to load the data file and also the time taken to build one tree.

`script_analysis:STRING` Reports the order in which commands listed of the imported script (specified by the string argument) are going to be executed. Unlike executing individual commands interactively, when commands are submitted in a script, POY5 determines the logical interdependency of operations and processes the commands in the order that yields the same results as if they were executed sequentially. This substantially optimizes parallelization and reduces memory consumption.

The colored output in the *POY Output* window of the *ncurses*

interface facilitates reading the output of `script_analysis`: red lines mark hard constraints that allow neither parallelization nor memory optimizations, blue lines mark constraints that allow the program to pipeline commands in parallel, and green lines mark fully parallelizable commands. When `POY5` is compiled with parallel off, all the operations are sequential, therefore, each potentially parallel operation is done as sequential repetitions of the subscripts described in the output of the command, reducing memory consumption.

`timer:STRING` Reports the value and the user time (in seconds) elapsed between two consecutive timer reports. The string value provides a label (typically a textual description) that precedes the time report in the output produced. The first timer report displays the time elapsed since the beginning of the `POY5` session. This command is useful for monitoring the execution time of specific tasks.

`xslt:(STRING,STRING)` Applies a user-defined xslt stylesheet to the XML output. The first string is the filename of the output, the second string is the name of the stylesheet requested to generate it.

NOTE

Extensible Stylesheet Language Transformations (XSLT) are used for the transformation of XML output into other formats. Because the XML output contains all the information regarding data and trees, using XSLT stylesheets greatly expand the capabilities of `POY5` to use and display results. Examples of potential applications includes graphical display of trees with proportional branch lengths, integration of tree topologies with geographical coordinate data for spatial mapping, and generating input files for other programs.

Defaults

`report(data,diagnosis,trees)` By default, `POY5` will print on screen the following items: the tree(s) in parenthetical notation with corresponding tree cost(s), diagnosis of each tree, and a graphical representation on the tree(s) in ascii format. This output can be re-directed to a file by specifying a file name enclosed in quotation marks, for example: `report("filename")`.

Examples

- `report("script_analysis",script_analysis:"/Users/runs/script1.poy")`
  This command produces the file `script_analysis` that lists the commands from the input script file `script1.poy` in the order that optimizes parallelization and memory consumption. In this example the complete path (`/users/datafiles/script1.poy`) is provided, which is not necessary if the directory containing the file `script1.poy` has already been assigned using the command `cd` (Section 3.3.4) in the same `POY5` session.

- `report("my_results")`
  This commands outputs the data, diagnosis and trees (the default) to the file `my_results`. Because no path is specified, the file is located in the current working directory.

- `report(data)`
  This command displays on screen a list of included and excluded terminals, their names and codes, gene fragments, synonyms, file names, and other relevant data.

- `report("Bivalve_data.txt",data)`
  This command performs the same operation as mentioned in the previous example, but rather than reporting the data to the screen of the output window, the data is saved in the file `"Bivalve_data.txt"` in the current working directory.

- `report(treestats)`
  This example displays on screen the costs of all trees in memory and the number of trees for each cost.

- `report("filename",treestats)`
  This commands outputs the costs of all trees in memory and the number of trees for each cost to a file `filename`.

- `report("filename_cr.txt",cross_references)`
  This command outputs the file `filename_cr.txt`, which indicates the presence and absence of all the data contained in all the input files.

- `report(cross_references:names:("file1","file3"))`
  This command produces a table showing presence ("+") and absence ("−") of data corresponding to all terminals contained in files `file1`

and `file3`. Since an output file is not specified, the table is displayed on screen.

- `report("taxa",terminals)`
  This command generates a file `taxa` that contains the lists and numbers of excluded and included terminals for each of the previously imported data files.

- `report(trees)`
  This command displays (on screen) the trees in memory in parenthetical notation with terminals separated by commas. By default, collapsing occurred if the length of the single assignment on a branch was zero.

- `report(trees:(total))`
  This command produces the same output as the example above but also includes the total tree cost in square brackets following each tree.

- `report("Run2",trees:(total,branches:(collapse:false)))`
  This command produces a file `Run2` that contains all trees in parenthetical notation, with the total tree cost in square brackets following each tree. Branch lengths will also be reported. In generating these trees, zero length branches were not collapsed.

- `report("filename",trees:(collapse:false,newick))`
  This command produces a file `filename` that contains all trees in Newick format with zero-length branches *not* collapsed.

- `report("filename",graphtrees)`
  This command saves all trees in memory in PDF format to the file `filename.pdf`.

- `report(asciitrees,"file1",trees:(newick,nomargin),"file2", graphtrees)`
  This command displays a tree in ascii format on screen and outputs to `file1` trees with zero-length branches collapsed in Newick format in a single line (using no margin, the format compatible with *TreeView*). It also writes to `file2.pdf` the graphical representation of these trees in PDF format.

- `report("hennig.ss",phastwinclad,trees:(hennig,total))`
  This command outputs all the static homology characters, including their cost regime, in the file `hennig.ss`; then append to the same file the trees currently in memory using the Hennig format, including the

total cost of each tree in square brackets. The generated `hennig.ss` is compatible with NONA, TNT, and Hennig86.

- `report("results",data,diagnosis,consensus:75,consensus, "consensus",graphconsensus)`
  This command reports the requested types of outputs (i.e. reports on the data, diagnosis, and 75 percent majority-rule consensus trees and strict consensus in parenthetical notation) to the file `results`. It also outputs a strict consensus tree in PDF format to the file `consensus.pdf`.

- `report(graphsupports,"bremertree",graphsupports:bremer)`
  This command reports on screen all previously calculated support values placed at the nodes of ascii trees and outputs to file the `bremertree.pdf` only the tree(s) with Bremer support values.

- `report(implied_alignments)`
  This command reports the implied alignments for all dynamic homology characters on screen. This is equivalent to `report(ia)`.

- `report("align_file",ia:names:("SSU","LSU"))`
  This command generates the file `align_file` that contains the implied alignments only for characters contained in data files `SSU` and `LSU`.

- `report("swapping",timer:"swap_end")`
  This command generates the file `swapping` that contains the string `swap_end` followed by the number of seconds (in decimals) elapsed since the execution of the previous `timer` argument.

- `report("new_tree_diagnosis.xml",diagnosis)`
  This command reports the diagnosis to the `new_tree_diagnosis.xml` file in XML format.

See also

- `calculate_support` (Section 3.3.2)

3.3.20 run

Syntax

run`(STRING)`

Description

Runs POY5 script file(s). The filenames must be included in quotes and, if multiple files are included, they must be separated by commas. The script-containing files are executed in the order in which they are listed in the string argument. Executing scripts using `run` is useful in cases when operations take a long time or many scripts need to be executed automatically, e.g. when conducting a sensitivity analysis [58]. An in depth description of creating and running scripts is provided in the *Quickstart*. There are no default settings of `run`.

NOTE

Note that if any of the scripts contain the commands `exit()` or `quit()`, POY5 will quit after executing that file. Therefore, if multiple files are submitted, only the last one must contain `exit()` or `quit()`.

Examples

- `run("script1","script2")`
  This command executes POY5 command scripts contained in the files `script1` and `script2` in the same order as they are listed in the list of arguments of `run`. Recall: If the last line of `script1` ends in `quit` or `exit`, POY5 will finish before `script2` can be run.

See also

- `exit` (Section 3.3.6)
- `quit` (Section 3.3.13)

3.3.21 save

Syntax

save`(STRING [,STRING ])`

Description

Saves the current POY5 state of the program to a file (POY5 file). The first, obligatory string argument specified the name of the POY5 file. The second, optional string argument specifies a string included in the POY5 file, that can be retrieved using the command `inspect` (Section 3.3.9).

POY5 files are not intended for permanent storage; they are recommended for temporary storing of a POY5 session by a user, checkpointing the current state of a search to avoid lost work in case the computer or the program itself fails, or to report bugs. POY5 will also automatically generate the file in many cases when a terminating error occurs (an important exception is out-of-memory errors). The format of these files might differ among different versions of POY5; consequently, these files might not be interchangeable between all the versions of the program.

Examples

- `save("alldata.poy")`
  This command stores all the memory contents of the program in the file `alldata.poy` located in the current working directory, as printed by `pwd()`.

- `save("alldata.poy","Total_evidence_data")`
  This command performs the same operation as described in the example above, but, in addition, it includes the string `Total_evidence_data` with the file `alldata.poy`, which can later be retrieved using the command `inspect` (Section 3.3.9).

- `save("/Users/andres/alldata.poy","Total_evidence_data")`
  This command performs the same operation as the command described above with the important difference that the file `alldata.poy` generated in the directory `/Users/andres/` instead of the current working directory.

See also

- `inspect` (Section 3.3.9)

- `load` (Section 3.3.10)

3.3.22 search

Syntax

search`([argument list])`

Description

`search` implements a default search strategy that includes tree building, swapping using TBR, perturbation using ratchet, and tree fusing. The strategy involves specifying targets for a driven search, such as maximum and minimum execution times, maximum allowed memory consumption for tree storage, minimum number of times the shortest tree is found, and an expected cost for the shortest tree. When executing `search` using parallel processing, trees are exchanged upon the completion of the command (after fusing). Because the lowest cost unique trees generated are selected and stored at the end of a `search` (defined by the user with `max_time`), aggressive use of this command in a parallel environment consists of including few sequential `search` commands that will allow the processes to exchange trees and add the pool of selected best trees to subsequent iterations of the command (see the example for parallel processing).

Trees that exists in memory prior to the `search` command are included in the set of trees available for the `fuse` but are not swapped.

NOTE

The total execution time of a *Timed Search* will exceed the search time specified by the user, as once the search has been completed, time is required to write the resulting trees to a file and/or to exchange trees between computer nodes, if the analysis was performed in parallel.

Arguments

`constraint:STRING` A complete description of this argument can be seen in `constraint` (Section 3.3.1) associated with the command `build`.

`hits:INTEGER` Specifies the minimum number of times that the minimum cost must be reached before terminating the search. The `hits` argument is not used in parallel processing.

`max_time:FLOAT:FLOAT:FLOAT` Specifies the maximum total execution time for the search. The time is specified as days:hours:minutes. For example, executing the search for 1.5 days can be expressed as 1:12:00 or 1.5:00:00.

`memory:LIDENT:FLOAT` Specifies the maximum amount of memory allocated for the stored trees during the search per processor. `POY5` *attempts* to consume memory within the specified limit, but it may surpass it in

certain operations (most notably during the ratchet). The LIDENT value expresses the units of memory (gb for Gigabytes and mb for Megabytes), whereas the float value specifies the actual value. Keeping memory consumption within the limit is approximate and is used as a rough guide to POY5, preventing the program from overflowing the memory. Furthermore, it is important to note that when running POY5 in parallel the maximum amount of memory specified by the user is allocated to each process. Under certain circumstances, however, POY5 may use more memory to avoid program failures.

NOTE

In order to maximize computational efficiency when using search in parallel processing environments the hits argument is ignored. However, a diverse set of trees which include the current best trees found among all the processes is desirable to improve the potential of tree fusing.

POY will *only exchange trees between processes at the end of each search command*. Therefore, to guarantee that separate processes seed each other with the best trees they have found every number of hours, it is advisable to use few successive search commands when executing the program in parallel. Each search will still be run in parallel, but after each one, trees will be exchanged between processors, to initiate each successive round of search.

`min_time:FLOAT:FLOAT:FLOAT` Specifies the minimum total execution time for the search. The time is specified as days:hours:minutes. This command is useful when the number of hits is specified but the actual cost of the tree is unknown. In this case, POY5 performs the search for at least the time specified by this argument.

`target_cost:FLOAT` Specifies the upper limit for the cost of the shortest tree.

`visited[:STRING ]` For a complete description see visited (Section 3.3.26). Note that this argument has a significant execution time cost, as outputting the trees becomes a bottleneck for the application.

Defaults

`search(max_time:0:1:0,min_time:0:1:0,memory:gb:2)` Under default parameters, the program performs a search for at most one hour using at most 2 GB of memory. [Note: If the user does not specify the value of `max_time`, the search will be terminated after one hour.]

Examples

- `search(hits:100,target_cost:385,max_time:1:12:13)`
  This command will attempt as many builds, swaps, ratchets, and tree fusings as possible within the specified time of 1 day, 12 hours, and 13 minutes, finding at least 100 hits (whichever occurs first, the time limit or the number of hits), knowing that the expected cost of the best hits is at most 385 steps.

- `For Parallel Implementation of search`
  search(max_time:0:6:0)
  select()
  search(max_time:0:6:0)
  select()
  search(max_time:0:4:0)
  select()
  This series of commands will attempt as many builds, swaps, ratchets, and tree fusings as possible within the specified total time of 16 hours. Trees are exchanged among processors at the end of each `search` and the best unique trees are then selected and included in the following `search` command.

- `search(max_time:00:48:00,constraint:"best_tree.tre")`
  This command will attempt as many builds, swaps, ratchets, and tree fusings within the specified time period of 48 hours. In this example, however, these operations are constrained by the tree specified in the file `best_tree.tre`.

- `search(max_time:00:48:00,visited:"visited.txt")`
  This command will attempt as many builds, swaps, ratchets, and tree fusings as possible within the specified time of 2 days. During this time, every visited tree and its cost during the local search will be stored in the file `visited.txt`.

See also

- `build` (Section 3.3.1)
- `swap` (Section 3.3.26)
- `transform` (Section 3.3.27)

3.3.23 select

Syntax

select`([argument])`

Description

Specifies a subset of terminals, characters, or trees from those currently loaded in memory, to use in subsequent analysis.

Arguments

Characters and terminals selection Specifies terminals and characters to use in subsequent analysis. The arguments in this group specify whether terminals or characters are being selected. *Identifiers* are used to specify which characters or terminals are being selected (see the *Character and terminal identifiers* argument group below for the description of methods for selecting specific terminals or characters).

`characters` Specifies that the subsequently listed identifiers refer to *characters* to be selected.

`STRING` Selects terminals listed in the file specified by the string argument.

`terminals` Specifies that the subsequently listed identifiers refer to *terminals* to be selected. By default, `POY5` assumes that the specification refers to terminals. For example, to analyze only those terminals listed in the file `opiliones` using the character data currently loaded in memory, use the command `select(files:("opiliones"))`. This command is equivalent to `select(terminals,files:("opiliones"))`.

When the command is executed, the list of selected terminals is printed on screen. `terminals` is only valid as an argument of commands `select` and `rename` (Section 3.3.18).

NOTE

Note that once specific terminals and/or characters are selected, the excluded data cannot be restored. To be able to reconstitute the original data set or to experiment with various character and terminal selections within a given `POY5` session, employ the commands `store` (Section 3.3.25) and `use` (Section 3.3.28).

Character and terminal identifiers *Identifiers* specify which characters or terminals are analyzed. In addition to the command `select`, identifiers are used as arguments for other commands that require selection of specific terminals or characters, such as commands `report` (Section 3.3.19) and `transform` (Section 3.3.27).

`all` Specifies all characters or terminals. Unless a terminals or characters file is selected, all the data is read by the program.

`codes:(INTEGER list)` Specifies the codes of characters or terminals. The codes are unique numbers that are generated by `POY5` when data files are first imported. The codes can be reported using the argument `data` (Section 3.3.19) of the command `report`. The codes are generated anew when a given data file is reloaded; therefore, they can be used only within a current `POY5` session.

`dynamic` Specifies the dynamic homology characters.

`files:(STRING list)` Specifies the filename list containing lists of terminals or characters.

`missing:INTEGER` Selects terminals or characters to be included in the analysis based on the proportion of missing data. The integer value ([0,100]) sets the maximum percentage of missing data in the analysis. Terminals or characters that have *fewer* missing data than that of the defined value are included in the analysis (compare with `not missing` below)

`names:(STRING list)` Specifies the names of the characters or terminals.

`not codes:(STRING list)` Specifies the characters or terminals other than those the codes of which are listed in the string list.

NOTE

For dynamic homology characters, the missing data refer to sequence fragments, whereas for static characters it refers to individual matrix positions. Therefore, when excluding terminals with missing data, the resulting set of selected terminals depends on the character type and might, or might not, be identical. For example, if a data file (containing sequences corresponding to a single fragment) were to include a very short sequence, this sequence is not treated as missing data regardless of its length. This is because in the context of dynamic homology a fragment, rather than an individual nucleotide position, constitutes a character. On the other hand, if the same data are treated as static characters, the taxon represented by a very short sequence might be excluded if the length of the sequence exceeds the threshold defined by the value of `missing`.

`not missing:INTEGER` Selects terminals or characters to be included from the analysis based on the proportion of missing data. The integer value ([0,100]) sets the minimum percentage of missing data. Terminals or characters that have *more* missing data than defined by the value are included in the analysis. In effect, this selects a complement of data to the argument `missing` (compare with `missing` above).

`not names:(STRING list)` Specifies the characters or terminals other than those the names of which are listed in the string list.

`static` Specifies the static homology characters.

Select trees The following arguments are used to select trees from the pool of trees currently in memory.

`best:INTEGER` Selects the number of best trees specified by the integer value. Best trees are not equivalent to optimal trees because best trees can include suboptimal trees in case the value of `best` exceeds the number of optimal (minimal-cost) trees. If the number of optimal trees exceeds the value of `best`, only a subset of optimal trees (equal to the value of `best` is selected in an unspecified order).

`optimal` Selects all trees of minimum cost.

`random:INTEGER` Randomly selects the number of trees specified by the integer value irrespective of cost.

> **NOTE**
>
> There is no special command in `POY5` to clear trees from memory. However, selecting zero best trees using the command `select(best:0)` effectively removes all trees currently stored in memory.

`unique` Selects only topologically unique trees (after collapsing zero-length branches) irrespective of their cost.

`within:FLOAT` Selects all optimal and suboptimal trees the costs of which do not exceed the current optimal cost by the float value. For example, if the current optimal cost is 507 and the float value of `within` is `3.0`, all trees with costs 507–510 are selected.

Defaults

`select(unique,optimal)` By default `POY5` selects all unique trees of optimal (best) cost. The remainder of the trees are deleted from memory.

Examples

- `select(terminals,names:("t1","t2","t3","t4","t5"), characters,names:("chel.aln:0"))`
  This command selects only terminals `t1`, `t2`, `t3`, `t4`, and `t5` and use data only from the fragment `0` contained in the file `chel.aln`.

- `select(terminals,files:("STL_terminals.txt"))`
  This command selects only the terminals specified in the file `STL_terminals.txt"`. In the data files that are subsequently imported, taxa that do not appear in this terminals file will be excluded from the analysis.

- `select(terminals,missing:30)`
  This command excludes from subsequent analyses all the terminals that have fewer than 30 percent of characters missing. The list of included and excluded terminals is automatically reported on screen. scr

- `select(optimal)`
  Selects all optimal (best cost) trees and discards suboptimal trees from memory. The pool of optimal trees might contain duplicate trees (that can be removed using `unique`).

- `select(unique,within:2.0)`
  This command selects all topologically unique optimal and suboptimal trees the cost of which does not exceed that of the best current cost by more than 2. For example, if the best current cost is 49, all unique trees that fall within the cost range 49–51 are selected.

See also

- `characters` (Section 3.3.23)
- `transform` (Section 3.3.27)

3.3.24 set

Syntax

set`([argument list])`

Description

Changes the settings of `POY5`. This command performs diverse auxiliary functions, from setting the seed of the random number generator, to selecting a terminal for rooting output trees, to defining character sets for different partitions.

There is no default setting for `set` and the order of its arguments is arbitrary.

Arguments

Application settings Some generic application settings. These have no effect on the analyses themselves.

`history:INTEGER` Sets the size of the `POY5` output history displayed in the *POY Output* window to the number of lines specified by the integer value. The size of the history must be greater than zero. This command has effect only in the *ncurses* interface. The default size of the output history is 1000 lines.

`log:STRING` Directs a copy of a partial output to the file specified by the string argument. The output includes the information in the *POY Output*, *Interactive Console*, and *State of Stored Search* windows of *ncurses* interface. Timers and current state of the search are not included in the log. If the log file already exists, `POY5` will append the text to it; if the log file does not exist, then `POY5` creates a new file. If the user would like to delete the contents of a pre-existing file, then the argument `log:new:"logfile"` creates a new initially empty file named `logfile`. [Note: setting a log will increase execution time to some extent, as the application has to output this log file.]

`nolog` Stops outputting the log to any previously selected file. See the description of the argument `log` above. Unless specified, no log is set by the program.

`root:LIDENT` Specifies the terminal to root output trees. The terminal can either be indicated as a taxon name (a `STRING`, which must appear in quotes, such as `"Genus_species"`) or the code, that is automatically assigned to the taxon by `POY5` at the beginning of each `POY5` session (for example, `set(root:45)`. The codes can be obtained using the command `report(data)`). The terminal codes, however, are consistent only within a current session.

`timer:INTEGER` Specifies the lapse of time in seconds that has passed between reporting the total execution time of a swap and build command. If the timer is set to 0, then no time messages are generated.

Cost calculation These arguments set the tree cost estimation procedures and are applied to all character types. The arguments are mutually exclusive: only the last specified argument of `set` is used.

`exhaustive_do` Applies a standard Direct Optimization algorithm for the tree cost estimation [59, 63]. The difference between this argument and `normal_do` is that the calculation of the tree costs during a search is much more intense, always looking for the best possible optimization for every single topology (instead of a lazy and greedy strategy used by `normal_do`).

`iterative:LIDENT:[:INTEGER ]` Applies the Iterative Pass Optimization [65] for the tree cost calculations. There are two forms of iterative pass: if the argument value is `exact` (the default), then a complete three

dimensional optimization is computed. Otherwise, if the argument value is `approximate`, then the iterations approximate the three dimensional alignment using pairwise alignments.

If the argument value is `exact`, this method improves the tree cost estimation but at the expense of execution time (by a factor of the sequence length). When `approximate`, the execution time footprint is much smaller, and far less memory is consumed. A typical heuristic strategy is to apply `iterative` at the very end of an analysis to polish the final set of trees and perform a final search.

Both arguments accept an optional integer, stating the maximum number of iterations that can be performed. If no integer is given, then the procedure iterates until no further tree cost improvement is incurred.

`normal_do` Applies a standard Direct Optimization algorithm [55] for the tree cost estimation. This is the default and fastest technique.

`normal_do_plus` Applies a more exhaustive Direct Optimization algorithm [55] for the tree cost estimation. During branch swapping, a more exhaustive calculation of the tree cost is performed.

NOTE

Due to the complexity of heuristics of the Iterative Pass Optimization [65], there is no guarantee that the tree cost recovered from the search will be exactly the same as produced by the diagnosis of the same tree. However, the cost of the tree found during the search can be verified by outputting the medians from the diagnosis (see the description of the argument `diagnosis` (Section 3.3.19)) of the command `report` and determining edge costs by hand. The cost of the tree found during the search might differ from that obtained by the rediagnosing of the same tree (see `rediagnose` (Section 3.3.16)), but will recover the same tree cost in subsequent rediagnoses.

Likelihood Optimization These arguments relate to codon partitions and also to the level of granularity (significant digits) and thoroughness (number of iterations) of the optimization routines used for defining the branches and model for the likelihood characters.

`codon_partition:(STRING,identifiers)` Specifies that the data be partitioned as codon data, named for the `STRING` argument, wherein 3 partitions will be defined. Each partition will consists of every third nucleotide position. For example, `set(codon_partition:("pos",names:("file")))` will create three sets of partitions named pos1, pos2 and pos3. This command is equivalent to the NEXUS partitioning commands:

Begin SETS;
pos1 = 1 - N /3;
pos2 = 2 - N /3;
pos3 = 3 - N /3;
END;

where N is the aligned length of the static data. The data must begin at the first codon position and must be a multiple of three.

`opt:none` No optimization procedures are performed; the current model parameters are kept, and branch lengths are set to a JC69 distance approximation.

`opt:coarse[:INTEGER ]` The tolerance of the routines is set to the algorithms default, 1e-3 (half the log of a full, exhaustive search). By default, the branches and model are each optimized once, setting the `INTEGER` allows the user to control how many times a branch/model pass occurs (by default the number is set to 3).

`opt:exhaustive[:INTEGER ]` The tolerance level is set to 1e-6. The algorithms for optimizing branches and the models will alternate until no more improvement occurs, or until as many times as specified by the `INTEGER`.

`opt:exhaustive_dyn[:INTEGER ]` This optimization level is for dynamic likelihood only, but will act like the exhaustive option under other circumstances. Under exhaustive, a temporary implied alignment on the data for optimizing the rate parameters is performed, but are optimized directly. This option may significantly add to the time required to complete the analysis.

NOTE

Rediagnosing a tree after `search` or `swap` may result in a different likelihood score if the number of optimization passes is set lower than convergence. For example, if we do a search where we only optimize the branches, then optimize the final trees model under coarse through `rediagnose(clear)`, then `POY5` will clear the parameters before optimizing and may result in a tree with higher cost.

Randomized routines

`seed:INTEGER` Sets the seed for the random number generator using the integer value. If unspecified, `POY5` uses the system time as seed.

NOTE

To reproduce a given search trajectory, the same seed value must be set.

NOTE

The sequence of randomizations is dependent on the version of OCaml used to compile the binaries. The algorithm to create the random seed number changed in OCaml version 3.12.0, thereby generating different sequences of pseudo-random numbers. To guarantee reproducibility of search trajectory, the user must ensure that the same version of OCaml is used during compilation.

Defaults

`set(history:5000,normal_do)` Under default settings the size of the history buffer is limited to 5,000 lines, the Direct Optimization is used for tree cost calculation, and the current time is used to specify the seed.

Examples

- `set(history:10000,seed:45,log:"mylog.txt")`
  This command increases the size of the history in the *ncurses* interface to 10,000 lines, sets the seed of the random number generator to 45, and initiates a log file `mylog.txt`, located in the current working directory.

- `set(root:"Mytilus_edulis")`
  This commands selects the terminal `Mytilus_edulis` as the root for the output trees.

- `set(iterative:exact)`
  Turns on the iterative exact algorithm in all the nucleotide sequence characters. The program will iterate on each vertex of the tree until no further tree cost improvements can be made.

- `set(iterative:approximate:2)`
  Turns on the iterative approximate algorithm in all the nucleotide sequence characters. The program will iterate either two times, or until no further tree cost improvements can be made, whichever happens first.

- `set(iterative:exact:2)`
  Same as the previous, but using the exact algorithm.

- `set(codon_partition:("coleop",names:("coleoptera_nd2.fasta")))`
  Sets codon partitioning of the data file `coleoptera_nd2.fasta`. Three sets are partitions named `coleop1`, `coleop2`, and `coleop3` will be created.

- `set(opt:exhaustive:3)`
  Set floating point optimization to a tolerance of 1e-6, and specify that a maximum of three optimization iterations occur.

- `set(opt:coarse:10)`
  Set floating point optimization to a tolerance of 1e-3, and specify that a maximum of 10 optimization iterations occur.

See also

- `report` (Section 3.3.19)

3.3.25 store

Syntax

store`(STRING )`

Description

Stores the current state of POY5 session in memory. The stored information includes character data, trees, selections, *everything*. Specifying the name of the stored state of the search (using the string argument) does *not*, however, generate a file under this name that can be examined; the name is used only to recover the stored state using the command `use`.

In combination with `use`, the command `store` is extremely useful when exploring alternative cost regimes and terminal sets within a single POY5 session.

Arguments

`STRING` Specifies the name of the stored search state of the current POY5 session.

Examples

- ```
  store("initial_tcm")
  transform(tcm:(1,1))
  use("initial_tcm")
  ```
  The first command, `store`, stores the current characters and trees under the name `initial_tcm`. The second command, `transform`, changes the cost regime of molecular characters, effectively changing the data being analyzed. However, the third command, `use`, recovers the initial state stored under the name `initial_tcm`.

See also

- `use` (Section 3.3.28)
- `transform` (Section 3.3.26)

3.3.26 swap

Syntax

```
swap([argument list])
```

Description

`swap` is the basic local search function in POY5. This command implements a family of algorithms collectively known in systematics as branch swapping
```

and in combinatorial optimization as hill climbing. They proceed by clipping parts of a given tree and attaching them in different positions. It can be used to perform a local search from a set of trees loaded in memory.

Swapping is performed on all trees in memory. During a search, `swap` can collect information about the visited trees and perform various kinds of checkpoints to reduce information loss in case `POY5` crashes.

`swap` is also used as an argument for other commands to specify a local search strategy in other contexts, for example, in calculating support values using the command `calculate_support` (Section 3.3.2).

All arguments of `swap` are optional and their order is arbitrary. The argument of different groups can be combined to fine tune the search heuristics, but the arguments within each group are mutually exclusive.

[Note: If more than one arguments of one argument group, such as *Join method*, is listed, only the last one is executed.]

Arguments

Branch break order During the local search, a branch is broken and local branch swapping is performed (see the *Neighborhood* group of arguments). The precise choice of which branches are broken first can affect both the speed and the local optimum found by the program. The following arguments select among the different strategies available in `POY5`.

`once` Breaks each branch only once during a local search; that is, if a broken branch does not yield a better tree, it is never broken again, no matter how many changes occur along the search trajectory.

`randomized` Chooses branches uniformly at random for breakages.

`distance` Gives higher priority to those branches with the greatest length.

Character transformation Concerns the transformation of characters *prior* to using the command `swap`.

`transform` Specifies a type of character transformation to be performed prior to swapping. See the command `transform` (Section 3.3.27) for the description of the methods of character type transformations and character selection.

Join method After breaking a tree (using SPR or TBR), the following arguments control the selection of the positions to join the broken clades.

`constraint[:depth:INTEGER | file:STRING ]` Constrains the join locations during the search using both a tree and an optional maximum distance from the break branch. Only sets defined either in the input file, or in the strict consensus of the files in memory are considered during swapping. An integer value of `depth` specifies the maximum distance from the break branch to attempt joins. The string value for `file` specifies an input file containing a single tree that defines topological constraints. Under default settings, `constraint` will use a consensus tree from the files in memory.

`all[:INTEGER ]` Turns off all preference strategies to make a join, simply trying all possible join positions for each pair of clades generated after a break, in a randomized order. The integer value specifies the maximum distance from the break branch to attempt joins.

`sectorial[:INTEGER ]` Join in edges at distance equal or less than the value of the argument from the broken edge, where the distance is the number of edges in the path connecting them. If no argument is given, then no distance limit is set.

Likelihood Optimization Specifies when the likelihood model and how the branches of the tree are optimized during the swap routine. These options are also available in the commands `build` and `fuse`. In all cases, a complete round of optimization will occur after the completion of a build.

`optimize(model[:LIDENT ],branch[:LIDENT ])` Specifies when `POY5` optimizes the likelihood model and how `POY5` optimizes the branches. These options are also available in the `fuse` and `swap` commands. In all cases a complete round of optimization will occur after the completion of swapping.

- `model:never` Do not optimize the model during the swap (the default).
- `model:always` Optimize the model after every swap.
- `model:threshold:FLOAT` Optimize the model if the cost of the join under the current model is within `FLOAT` times the current best cost.

`branch:never` Do not optimize the branches during the swap process. Estimates are made based on the proportion of sites that would undergo a transformation.

`branch:all_branches` Optimize all branch lengths on each join.

`branch:join_region` Optimize a maximum of five branches; the new edge, and the two edges on either side (the default).

`branch:join_delta` Optimize the branches along the path from the break to the new join location.

Neighborhood A neighborhood is a subset of topologies reachable from a given area of the tree by a given search method. The basic standard procedures for local search in phylogenetic analysis are SPR and TBR [50]. The nearest-neighbor interchanges (NNI) [7] swapping strategy is implemented by combining the arguments `spr` and `sectorial` (see *Join method* group of arguments) within the `swap`, i.e. `swap(spr,sectorial:1).`

`alternate` Performs `spr` and `tbr` swapping iteratively until a local optimum is found. This is a specific strategy of performing `tbr`, as the trees visited by `spr` are a subset of those visited by `tbr`.

`spr[:once]` This argument performs `spr` swapping, starting from the current trees in memory and subsequently repeating the SPR procedure until a local optimum is found. If the optional value `once` is specified, `spr` stops once the first tree with better cost is found.

`tbr[:once]` This argument performs `tbr` swapping, starting from the current trees in memory and subsequently repeating the TBR procedure until a local optimum is found. If the optional value `once` is specified, `tbr` will stop once the first tree with better cost is found.

Reroot order During TBR, the following options control the order of the rerooting.

`bfs[:INTEGER ]` Reroots using breath first search [9] from the broken edge, within the arguments value distance from the root of the clade. If no value is given, there is no limit distance for the rerooting. By default, `bfs` is used with no limit distance for the rerooting.

Trajectory The following arguments define the direction of the search in the defined neighborhood.

`around` Changes the trajectory of a search by completely exploring the neighborhood of the current tree in memory and choosing the best swap position before continuing. The default in POY5 is to choose the first one available that shows a better cost than the current best cost.

`annealing:(FLOAT,FLOAT)` Uses simulated annealing [30]. If the argument's value is (a, b), POY5 accepts a tree with cost c when the best known tree has cost d with probability $\exp(-(c-d)/t)$, where $t = a \times \exp -i/b$ and i is the number of tree evaluated in the local search.

`drifting:(FLOAT,FLOAT)` Uses POY5 drifting function [20]. If the argument's value is (a, b), then POY5 always accepts a tree with better cost than the current best cost, with probability a a tree with equal cost, and with probability $1/(b+d)$ a tree with cost d greater than the current best cost.

Trajectory samples During the search, POY5 visits a large number of trees. For some applications it might be desirable to collect information about the trees examined during a search: for example, to provide backups of the state of a search (in an unlikely crash), or to examine the characteristics of the alignments. The difference from the `swap` arguments is that the user can choose any combination of trajectory samples, and that can be used during the search. None of the trajectory samples is used by default.

`recover` Stores the current best tree in memory that can be recovered in case of failure. If it is necessary to recover such trees after an aborted command, use `recover` (Section 3.3.15). If the program terminates normally, the stored trees are exactly those produced at the end of the `swap`. Using `recover`, however, requires twice as much memory compared to swapping without it.

`timeout:INTEGER` Specifies the number of seconds after which tree branch swapping is stopped. The current best tree is the result of the swap after the timeout.

`timedprint:(INTEGER,STRING)` `timedprint:(n,"trees.txt")` prints the current best tree in memory to the file `trees.txt`, at least every `n` seconds. However, POY5 typically underestimates the amount of time

and, therefore, the samples can be slightly sparcer. `timedprint` can only be used in combination with the argument `recover`.

`trajectory[:STRING ] trajectory:"better.txt"` will store every new tree found with a better score during the local search in the file `better.txt`. The string is the filename where the trajectory is to be stored, which is optional (indicated by brackets); if not added, the trees are printed in the standard output (*flat* interface) or the output window (*ncurses* interface).

`visited[:STRING ] visited:"visited.txt"` will store every visited tree and its cost during the local search in the file `visited.txt`. The (optional) string is the filename where the trajectory is to be stored. If not included, the trees are printed in the standard output (*flat* interface) or the output window (*ncurses* interface).

Tree selection As the tree search proceeds, a tree may or may not be selected to continue the search or to return as a result. The following arguments determine under what conditions can a tree be acceptable during the search.

`threshold:FLOAT` Sets the percentage cost for suboptimal trees that are more exhaustively evaluated during the swap, meaning that trees within the threshold are subject to an extra round of swapping. For example, if the current optimal tree has cost 450, and `threshold:10` is specified, trees with cost at most 495 are swapped. `threshold` is equivalent to *slop* of `POY3`.

`trees:INTEGER` Maximum number of best trees that are retained in a search round, per tree in memory.

Defaults

`swap(trees:1,alternate,threshold:0,bfs)` By default, current trees are submitted to a round of alternate rounds of SPR and TBR using breadth first search and one best tree per starting tree is kept.

Examples

- `swap()`
  This command performs swapping under default settings.

- `swap(trees:5)`
  Submits current trees to a round of SPR followed by TBR. It keeps up to 5 minimum cost trees for each starting tree.

- `swap(transform(all,(static_approx)))`
  Submits current trees to a round of SPR followed by TBR, using static approximations for all sequence characters.

- `swap(trees:4,transform(all,(static_approx)))`
  Submits current trees to a round of SPR followed by TBR, using static approximations for all characters, keeping up to 4 minimum cost trees for each starting tree.

- `swap(constraint:(depth:4))`
  Calculates a consensus tree of the files in memory and uses it as constraint file, then joins at a distance of at most 4 from the breaking branch. This is equivalent to `swap(constraint:(4))`.

- `swap(constraint:(file:"bleh"))`
  Reads the tree in file `bleh` and use it as constraint for the search. This is equivalent to `swap(constraint:("bleh"))`. This presumes that the file `bleh` is located in the current working directory.

- `swap(drifting:(0.5,2.0))`
  Defines the direction of search via drifting, such that there is a 50% probability of replacing the current tree with a new tree of equal cost. For suboptimal trees with a cost d greater than the current best tree, the probability of accepting this tree is $1/(2.0 + d)$. For example the probability of keeping a suboptimal tree of 3 steps longer $= 1/(2.0 + 3) = 0.2$.

- `swap(sectorial:4)`
  Submits current trees to a round of SPR followed by TBR. Join will take place at a distance equal or less than the value of the argument from the broken edge, where the distance is the number of edges in the path connecting them. If no argument is given, then no distance limit is set.

- `swap(spr,all,optimize:(model:never,branch:join_region))`
  Submits the current trees to a round of SPR swapping. Following each round of `spr`, the model is never optimized, but a maximum of five branches (the new edge, and the two edges on either side of the join site) are optimized.

- `swap(recover,timedprint:(5,"timedprint.txt"))`
  Saves the current best tree to the file `timedprint.txt` every 5 seconds.

See also

- `transform` (Section 3.3.27)

3.3.27 transform

Syntax

transform`([argument list])`

Description

Transforms the properties of the imported characters from one type into another. This includes changing costs for indels and substitution, modifying character weights, converting dynamic into static homology characters, transforming nucleotide into chromosomal characters, and specifying characters as either static or dynamic likelihood characters, among other operations.

The essential arguments of the command `transform` include identifiers and methods. The methods specify what type of transformation is applied to the set of characters, as specified by identifiers, as defined in the description of the command `select` (Section 3.3.23).

The methods, or string of methods (each separated by a comma), should be enclosed in parentheses. Identifiers precede the methods and are separated from them with a comma. It is important to remember that only identifiers of *characters* (such as `names`, `codes`, among others) can be used. Parentheses separate these essential arguments from all other optional arguments that might be included in the list. If only identifiers and methods are specified, the argument list of `transform` is included in parentheses. For example, the command `transform(all,(gap_opening:1))` contains only an identifier (`all`) and a method (`gap_opening`). [Note: this is different from previous versions of `poy` where double parentheses were required.] Minimally, only methods can be specified: in this case, the transformation is applied to *all* characters to which the transformation method can be applied. For instance, `transform(gap_opening:1)`, where `gap_opening` defines the transformation method.

There are no default values for `transform`, thus if no arguments are specified (`transform()`), the command does nothing.

Arguments

Identifiers Identifiers specify which characters are transformed. Only identifiers of characters (*not* terminals) can be used. If identifiers are omitted, the transformation to is applied to all applicable characters. For example, `transform(all,(tcm:(1,1)))` is equivalent to `transform(tcm:(1,1))`. See the command `select` (Section 3.3.23) for detailed description of identifiers.

Character selection methods This set of arguments specifies different transformations that can be applied to selected characters. If multiple transformation methods are applied sequentially in the same list of arguments, the effect of the methods listed earlier might be altered or canceled by methods listed after that. Thus, caution must be used in designing complex strategies with multiple character transformations. See the note on command order (Section 3.2).

`alphabetic_terminals` Alphabetizes the terminals in the data file that is input into `POY5`. This is used in conjunction with build or search. See the description of the argument `randomize_terminals`.

`auto_static_approx` Evaluates each selected fragment and, if the number of indels appear to be low and stable between topologies, then the character is transformed to the equivalent character using static homologies with the implied alignment [64]. This method greatly accelerates searching and is applicable only to nucleotide sequences under dynamic homology analysis.

`auto_sequence_partition` Evaluates each fragment in the data file and if a long region appears to have no indels, then the fragment is broken inside that region. Any number of partitions can occur along a fragment. Fragmenting long sequences greatly accelerates searching. This method is applicable only to dynamic homology characters that are not prealigned, and requires a tree in memory.

`direct_optimization` Transforms the characters specified so that the initial assignment of sequences to the internal vertices of a tree use direct optimization [59]. This method is recommended for small alphabets (fewer than 7 elements). It is only applicable to dynamic homology characters.

`do` Synonymous with `direct_optimization`.

`fixed_states[:(STRING,LIDENT)]` Transforms the characters specified in fixed state characters [60] where the initial assignment of sequences to the internal vertices of the tree is one of the observed sequences. If the observed sequences contain ambiguities, only those that resolve closest to another sequence are added to the set of valid states. It is only applicable to dynamic homology characters. The optional arguments may be specified for use with the `chromosome` character type. The `STRING` specifies the root name of Mauve [10] readable files ("`STRING` _i_j.alignment" where i and j are median states) to generate graphical sequence maps between median states. The default `LIDENT` is set to `full_polymorphism`, which will consider all sequence ambiguities. As one might expect, this option is also the most time consuming. Alternatively, if the `LIDENT` is set to `simpl_-polymorphism` or `ignore_polymorphism`, new potential median states will be generated that consider fewer sequence polymorphisms in their calculations, thus reducing execution time.

`gap_opening:INTEGER` Sets the cost of opening a block of gaps to the specified value. Note that this cost is ***in addition*** to the standard cost of the insertion, as specified by a given transformation cost matrix. The default in `POY5` is not to have extension gap cost (`gap_opening:0`). If the gap opening cost is a, and $indel(x)$ is the cost of inserting (or deleting) a base x according to the tcm assigned to the character, the total cost of inserting (or deleting) the sequence $s[0...n]$ is $a + indel(s[0]) + indel(s[1]) + ... + indel(s[n-1]) + indel(s[n])$. This method is applicable only to dynamic homology characters with the nucleotide alphabet. These affine gap costs can not be used in conjunction with `prealigned` data files.

`level:(INTEGER,LIDENT)` The integer argument specifies the heuristic level in median sequence calculation. This determines the number of possible states stored at each median sequence position. For nucleotide data, all possibilities are stored. This median states can be a single character (e.g. A, C, G, T) or a combination of `INTEGER` characters (e.g. A/C versus G). The default for amino acid sequences and for `custom_alphabet` characters it is 2. Storage and set up time increase combinatorially with level number. If the `LIDENT` is `first`, ties are broken (if the number of equally costly states is greater than the level number) by choosing the first median state examined; if `last`, the last state, and if `at_random` then uniformly at random. The default choice method is `first`. The

maximum `level` of any dataset is equal to the alphabet size + 1. [Note: Levels greater than the default levels can consume large amounts of memory.]

`multi_static_approx` Calculates the implied alignment for each tree in memory and convert them to static homology characters using the alignment's cost regime. The new character set will be the union of all those characters generated for all the trees [70]. This option is intended only for heuristic search purposes and is applicable only to dynamic homology characters.

`partitioned:LIDENT` Similar to `auto_sequence_partition`, the difference being that no tree is required prior to partitioning, and large sequence length variations at the ends of sequences are treated as missing data. If the `LIDENT clip` is chosen, then a large difference in length at the end of the sequence is assumed to be caused by missing data. For `clip` to take effect, at least two fragments must be found. If `LIDENT noclip` is selected, then sequence ends are treated like any other fragment.

`prealigned` Treats the sequences as prealigned and uses the cost regime according to the specified transformation cost matrix. All other cost parameters are ignored (including affine gap costs). This command requires that all the specified sequences have the same length (which can be achieved by the insertion of `N`'s or `X`'s at the 5 prime (`5'`) and/or 3 prime (`3'`) ends of the sequence if data are missing).

`randomize_terminals` Randomizes the terminals in the data file that is input into `POY5`. This is used in conjunction with build or search. See the description of the argument `alphabetic_terminals`.

`search_based:(STRING,STRING)` Transforms the optimization of `fixed_states` characters to `search_based` optimization [66] by adding the sequences found in the file specified by the second argument to the character specified by the first argument.

`sequence_partition:INTEGER` Partitions the sequences in the argument's value number of fragments of roughly the same length. This method is applicable only to dynamic homology characters.

`static_approx[:LIDENT ]` Transforms the sequences to the static homology characters corresponding to their implied alignments and their transformation cost matrix [64]. The resulting characters and their

number will vary depending on the characteristic of transformation cost matrix assigned to each sequence. For example, if the cost of both substitutions and indels is `1`, then one non-additive character is created per each homologous position in the implied alignment. If the cost of substitutions is `1` and the cost of indels is `2`, then one character is created for each homologous position, and one extra character for each homologous position with gaps. In more complex cases, a Sankoff character is created.

The `LIDENT remove` excludes all uninformative characters information (except autapomorphies), whereas the `LIDENT keep` retains these characters. The default is `remove`. This method is applicable only to dynamic homology characters. If a non-metric transformation cost matrix is in use, this transformation will assume that the non-metricity is due to the individual insertion and deletion cost.

NOTE

The transformation of dynamic into static homology characters cannot be reversed. Therefore, caution must be taken when the transformation is applied. For example, if sequence characters have been transformed into static characters at top level using the command `transform(all,(static_approx))`, all commands executed subsequently will be applied to the transformed data. However, if the transformation has been applied *within* another command (as an argument of `swap`, for instance, `swap(transform(all,(static_approx)))`), the characters will transformed only for that specific operation.

NOTE

It is important to remember that the local optimum for the dynamic homology characters can differ from that for the static homology characters based on the same sequence data. Therefore, performing additional searches on the transformed data (for example, in calculating support values based on individual nucleotides rather than on sequence fragments) can produce a discrepancy in tree costs.

`tcm:(INTEGER,INTEGER)` Defines the transformation cost matrix. The first integer value specifies substitution cost, the second integer value defines

indel cost. By default, the cost of substitutions and indels are both `1`. [Note: previous versions of `poy`, had the cost of an indel set to `2` (i.e. `tcm:(1,2)`).]

NOTE

When constructing a transformation cost matrix it is important to do so in a text editor such as Notepad (for Windows), TextEdit (for Mac), or Nano (for Linux). Generating a tcm in a word processing application such as Microsoft Word may lead to the insertion of hidden characters, which will result in an error.

`tcm:(STRING,INTEGER)` Defines the transformation cost matrix by importing a file (specified by the string value) that contains a user defined nucleotide transformation cost matrix. This method is applicable only to dynamic homology characters. The transformation cost matrix file contains five rows and columns with values listed in the following order (left to right and top to bottom): adenine, cytosine, guanine, thymine/uracil, and indel. A similar pattern is followed for amino acids where the matrix columns and rows reflect all the amino acid names in alphabetical order (read left to right and top to bottom) with the last row and column containing a gap cost. The costs must be symmetrical (that is, the cost of the A to T substitution is equal to the cost of T to A substitution). For example:

0	2	1	2	4
2	0	2	1	4
1	2	0	2	4
2	1	2	0	4
4	4	4	4	0

The integer argument specifies the heuristic level in median sequence calculation. This determines the number of possible states stored at each median sequence position. For nucleotide data, all possibilities are stored. The default for amino acid sequences and for `custom_alphabet` characters is 2. Storage and set up time increase combinatorially with level number.

`ti:STRING/(INTEGER list)` Synonym of `trailing_insertion`.

`trailing_insertion:STRING/(INTEGER list)` The tail and prepend costs specify the cost of having an insertion of each element in the alphabet at the beginning or end of a sequence. The string is the name of a file containing the cost of a trailing insertion corresponding to each of the elements in the alphabet separated by spaces. The last element in the list is the cost of the indel of a gap (should be 0). Instead of a file, the list can be the input of the argument, in the same order, separated by commas. Synonym of the argument `ti`. This method is applicable only to dynamic homology characters.

`td:STRING/(INTEGER list)` Synonym of `trailing_deletion`.

`trailing_deletion:STRING/(INTEGER list)` The tail and prepend costs specify the cost of having a deletion of each element in the alphabet at the beginning or end of a sequence. The string is the name of a file containing the cost of a trailing deletion corresponding to each of the elements in the alphabet separated by spaces. The last element in the list is the cost of the indel of a gap (should be 0). Instead of a file, the list can be the input of the argument, in the same order, separated by commas. Synonym of the argument `td`. This method is applicable only to dynamic homology characters.

`weight:INTEGER/ FLOAT` Changes the cost of specified characters to a constant or absolute value (weight), which is specified by either a float or an integer value. This method is applicable to any character type and can be applied to individual characters in a data set.

`weightfactor:INTEGER/ FLOAT` Changes the cost of specified characters by a multiplicative factor (weight factor), and is specified by either a float or an integer value. This method is applicable to any character or character set. This argument differs from `weight` in that this cost is applied to a class of characters.

Chromosome and genome transformation methods For these character types, `POY5` optimizes nucleotide-, locus-, and chromosome-level variation ***simultaneously***. The arguments in this group transform nucleotide characters into chromosomal characters and allow for translocations, inversions, and indel events both at the locus-level for chromosomal data and at the chromosome-level for genomic data.

The functions to calculate breakpoint and inversion distances between two sequences of gene orders are taken from the rearrangement software packages:

GRAPPA, Genome Rearrangements Analysis under Parsimony and other Phylogenetic Algorithms [2], and MGR, Multiple Genome Rearrangements [5], as well as from the Concord TSP Solver available at http://www.tsp.gatech.edu/concorde/.

`annotate:(LIDENT,FLOAT,FLOAT,FLOAT,FLOAT)` Used in conjunction with `chromosome`, specifies the annotation method for unannotated chromosome sequences. The `LIDENT` is `mauve`, which utilizes the Mauve algorithm [10]. As with other chromosome and genome alignment methods, the Mauve algorithm uses "anchoring" in order to speed up the alignment process. However, it differs from other such programs in that the anchor selection method relaxes the assumption of collinearity of the genomes and instead, identifies and aligns regions of local collinearity called locally collinear blocks (LCB). These LCBs represent a homologous region of a sequence shared by two or more genomes and are without internal rearrangements. The float parameters, which are order dependent, include the quality of the LCB (default value 35), LCB coverage (default 0.30), minimum length of LCB as a percentage of sequence length (default 0.01), maximum length of LCB as a percentage of sequence length (default 0.10). Percent calculations are based on the shorter of two sequences compared.

`annotated:([argument list])` Used in conjunction with `read`, specifies that the data are chromosomal sequences with pipes (" | ") separating individual loci. The locus homologies and rearrangements are then determined dynamically by the arguments specified in the argument list (`locus_breakpoint` and `locus_inversion` for rearrangements; `locus_indel` costs; `circular` and arguments associated with the determination of medians).

`chromosome:([argument list])` Specifies parameters for the creation of chromosome- and genome-level HTUs (medians). The arguments of `chromosome` define homologous blocks within unannotated chromosome sequences by specifying parameters within the Mauve aligner [10] using `annotate` and these Mauve parameters include block quality, block coverage, minimum and maximum block length. Users also specify the costs assigned to locus-level transformation events: (i.e. `locus_inversion` or `locus_breakpoint`, and `locus_indel`), take into account whether the chromosome is linear or circular (`circular`), and implement a number of heuristic procedures to accelerate computations (`median`, `swap_med`, and `med_approx`). Under default settings, the pairwise

distance between two chromosome segments or two chromosomes is determined using breakpoint rather than inversion calculations and the rest of the arguments are executed under their default settings.

`chrom_hom:FLOAT` Specifies the lower limit of distance between two chromosomes beyond which point the chromosomes are not considered to be homologous. The default value of `chrom_hom` is `0.75`.

`chrom_indel:(INTEGER,FLOAT)` Specifies the cost for insertion and deletion of a chromosome in analysis of multiple chromosomes. The integer value sets gap opening cost (o), whereas the float value sets gap extension cost (e). The indel cost for a fragment of length l is specified by the following formula: $o + (l \times e)$. The default values are $o = 10, e = 1.0$.

`circular:BOOL` Specifies whether the chromosome is circular (boolean value `true`) or linear (boolean value `false`). The default value of `circular` is `false` (linear chromosome).

NOTE

Note that the argument `locus_breakpoint` cannot be used simultaneously with the arguments `locus_inversion` and `chrom_inversion` as they designate *alternative* methods of calculating distance between two chromosomes. If both arguments are specified, the latter will be executed. The order of other locus level arguments is arbitrary.

`genome:([argument list])` Specifies parameters for creating genome-level HTUs (medians). The arguments of `genome` define homologous blocks within annotated genome sequences by specifying parameters within the Mauve aligner [10] including block quality, block coverage, minimum and maximum block length. Users also specify the costs assigned to locus-level transformation events: (i.e. `locus_inversion` or `locus_breakpoint`; `locus_indel`; and `chromosome_indel`), and implement a number of heuristic procedures to accelerate computations (`median`, `swap_med`, and `med_approx`). Under default settings, the pairwise distance between two genome segments or two genomes is determined using breakpoint rather than inversion calculations and the remainder of the arguments are executed under their default settings.

`locus_breakpoint:INTEGER` Calculates the breakpoint distance [4] between two pairs of chromosomes given the cost for rearrangement specified by an integer value. The breakpoint distance calculation considers a chromosome or genome $G = (x_1, \ldots, x_n)$ of n gene, wherein each appears exactly once and its orientation is either positive or negative. Gene orders are altered by gene rearrangement operations: gene inversion, gene translocation, gene inversion and translocation (see Figure 3.5).

Original order of gene	g_1	g_2	g_3	g_4	g_5
Inversion	g_1	g_2	$-g_5$	$-g_4$	$-g_3$
Translocation	g_3	g_4	g_5	g_1	g_2
Translocation and inversion	$-g_5$	$-g_4$	$-g_3$	g_1	g_2

Figure 3.5: Examples of gene rearrangements: inversions and translocations.

The breakpoint distance takes into account rearrangements but not inversions. Given G and G', a pair of genes (g_i, g_j) is a breakpoint if (g_i, g_j) occur consecutively in G but neither (g_i, g_j) nor $(-g_j, -g_i)$ occur consecutively in G' [42]. The breakpoint distance between G and G' is the number of breakpoints between them. Figure 3.6 shows two breakpoints between G and G'. The breakpoint can be calculated easily in linear time. This argument *cannot* be used in conjunction with `locus_inversion`. The default value of `locus_breakpoint` is `10`.

g_1	g_2	g_3	g_4	g_5	g_6
g_1	g_2	$-g_5$	$-g_4$	$-g_3$	g_6

Figure 3.6: Rearrangement calculations between chromosomal or genomic data of six genes $g_1, \ldots, g_6$, where the rearrangement events are detected as either two breakpoints $(g_2, g_3), (g_5, g_6)$ or a single inversion (g_3, g_4, g_5).

`locus_indel:(INTEGER,FLOAT)` Specifies the cost for insertion or deletion of a chromosome segment. The integer value sets the gap opening cost (o), whereas the float value sets the gap extension

cost (e). The indel cost for a fragment of length l is specified by the following formula: $o + (l \times e)$. The default values are $o = 10, e = 1.0$.

`locus_inversion:INTEGER` Calculates the inversion distance [23] between two chromosome segments given the cost for inversion specified by the integer value. The inversion distance takes in consideration rearrangements and inversions. Given G and G', the inversion distance between them is the number of inversions to convert chromosome or genome G into G' [23]. Figure 3.6 shows one inversion between G and G'. The inversion distance can be calculated in linear time. The breakpoint distance is normally larger than inversion distance. This argument *cannot* be used in conjunction with `locus_breakpoint`.

`max_kept_wag:INTEGER` Defines the maximum number of Wagner-based possible ancestral sequence alignments kept to create the next set of alignments during the pairwise alignment with rearrangement process. The default value of `max_kept_wag` is 1, however, at every step in the pairwise alignment with rearrangement process, the original order (1...n) is always considered as a potential solution.

`median:INTEGER` Specifies the number of alternative locus and chromosome rearrangements of the best cost selected (randomly) for each HTU (hypothetical taxonomic unit) or median. Limiting the number of rearrangements stored in memory (smaller value of `median`) is a heuristic strategy to accelerate calculations at the expense of thoroughness of the search. By default, only 1 rearrangement is retained (the first one found). If more than one rearrangement is specified, the selected number of rearrangements is selected in random order from the pool of all generated rearrangements.

`med_approx:BOOL` Approximates chromosome medians using a fixed-states approach. This is most useful to accelerating tree building and searching operations for large chromosomal data sets. The boolean value `true` applies the fixed-states optimization. The default value is `false`.

`median_solver:LIDENT` Specifies the median solver. User can choose from default(caprara),vinh,siepel,bbtsp,coaletsp,chainedlk and simplelk. The default median solver is *Caprara* [8], but the user can alternatively choose *BBTSP, ChainedLK, COALESTSP, MGR*

[5], *Siepel* [45], *SimpleLK* and *Vinh* (the TSP solvers BBTSP, CoalesTSP, ChainedLK and SimpleLK are taken from the Concorde package and this package is required for these median solvers. Download Concorde package from http://www.tsp.gatech.edu//concorde/downloads/downloads.htm.

`seq_to_chrom:([argument list])` Transforms nucleotide type data into chromosome type data to allow rearrangements, inversions, and locus-level indel operations. The chromosome-specific options (e.g. `locus_breakpoint`, `locus_inversion`, and `locus_indel`) can be specified by the argument.

`swap_med:INTEGER` Specifies the maximum number of swapping iterations to search for best pairwise alignment of two chromosomes taking into account locus-level rearrangement events. Limiting the number of swapping iterations accelerates the search at the expense of thoroughness. The default value is `1`.

`translocation:INTEGER` Specifies the cost associated with the *Translocation* of a region of one chromosome to another chromosome. By default, the cost is set to 10.

Custom alphabet and break inversion transformation methods This set of arguments govern the transformation of characters that employ a user-specified alphabet. This includes characters of the custom alphabet, as well as break inversion type.

`breakinv:(argument list)` An enhancement of the data file type `custom_alphabet` allowing rearrangement events. Syntactically, `breakinv` data type is identical to `custom_alphabet` data type. Users specify the costs assigned to locus-level transformation events: (i.e. `locus_inversion` or `locus_breakpoint`) and implement the heuristic procedure to accelerate computations (`median`). The `orientation` (denoted by the inclusion of "∼" symbols in the data file) is by default set to true and cannot be changed to false.

`breakinv_to_custom` Transforms `breakinv` character type into `custom_alphabet` characters. This transformation prevents the use of rearrangement operations.

Likelihood transformation methods This set of arguments enables analysis using several variants of the maximum likelihood criterion, including

Most Parsimonious Likelihood (MPL) and Maximum Average Likelihood (MAL).

`likelihood([:argument list])` Transforms the specified characters to either static or dynamic likelihood characters. All arguments are optional except for model or model selection, which must be specified. The defaults are presented in the sub-sections below.

`elikelihood([:argument list])` Transforms the specified characters to either static or dynamic likelihood characters while estimating the parameters based on a parsimony tree [71]. The initial rates (for example of GTR) are set to the proportion of those transformations found across branches of the tree. As with the standard likelihood command, all arguments are optional except for model or model selection, which have to be specified; the defaults are presented in the sub-sections below.

`parsimony` Transforms likelihood characters to parsimony characters. This command will revert the characters back to their original parsimony state, prior to transformation to likelihood (e.g. any transformation costs previously associated with these characters will be restored.

NOTE

Dynamic likelihood for unaligned sequence data can be implemented as MAL [71] (provided the characters have been transformed to `fixed_states` first) and MPL [3]. Both MAL and MPL can be applied to static characters.

NOTE

Dynamic likelihood characters necessarily include gaps or indels as a fifth state in the character alphabet and must be specified (see the description `Gap treatment` (Section 3.3.27) below.).

Character frequencies Estimate the equilibrium frequencies of the characters in the stationary Markov process. In order to yield the likelihood score, the conditional probability of the data is multiplied by these frequencies. In general, maximum likelihood applications in phylogenetics use the observed (empirical) frequencies of the characters in the data as an approximation to the likelihood-optimized frequencies. This approximation is sensible for sequences evolving nearly neutrally, but this approximation breaks down for sequences under moderate or strong selection.

`priors:equal` Constrains each of the equilibrium frequencies to be $\frac{1}{r}$, where r is the character alphabet size.

`priors:estimate` Uses observed frequencies of the characters to approximate the ML-optimized equilibrium frequencies. Under dynamic likelihood, the indel equilibrium frequency is estimated from the minimum number of indels required to align all the sequences:

$$\pi_{gap} = \frac{\sum_{i=1}^{n} |S_{\hat{i}}| - |S_i|}{n|S_{\hat{i}}|}$$

where $S_{\hat{i}}$ is the longest sequence and n is the number of sequences.

`given:(FLOAT list)` Uses equilibrium frequencies given as a comma-separated list specified by the user.

Cost Specifies the variant of relative likelihood (*sensu* [47]) used by `POY5` to calculate the costs at each node and score the topology.

`mal` Specifies the standard criterion of MAL [16], in which the likelihood is derived from the sum over all internal vertex labelings. Because the vertex assignments are marginalized, this version of relative likelihood is not compatible with the optimization of unaligned sequences (dynamic likelihood), unless of course the characters have been transformed to `fixed_states` first. This is the default likelihood criterion in `POY5`.

`mpl` Specifies the criterion of MPL [3], in which the likelihood is derived from the set of most probable state assignments at each node. Under this criterion, vertex state assignments are not marginalized, and so this variant of relative likelihood is compatible with dynamic likelihood. However, several studies have noted that parameters under this criterion may not be identifiable [73] and that the method may not be statistically consistent [34].

Determination of alphabet size This option applies to qualitative characters only, and is used to define the alphabet size for `transform`. While unobserved states do not affect the cost of the tree under parsimony, this is not necessarily the case with likelihood. This argument enables the user to specify the size of the alphabet.

`alphabet:min` Sets the alphabet size to be the minimum value that encompasses all character observations for each state.

`alphabet:max` Do not modify the alphabet; use the one currently set under the character type. This can be up to 32.

`alphabet:INTEGER` Specifies the size of the alphabet as defined by the `INTEGER` value. This value must be $\geq$ the number of observed states. If the specified value is greater than the number of observed states, this `INTEGER` will be used in calculation of the likelihood of the tree.

Gap treatment Defines how atomic gaps (indels) are treated in likelihood analysis. When employed, the `POY5` approach to indels on static likelihood characters is as a 5th state [33, 67]. POY is unique among likelihood implementations both by enabling indel parameterization for any reversible, stationary model applied to static characters and, more crucially, for implementing dynamic likelihood for true sequence optimizations and topology inference. Indel-aware models are scarce, but see Rivas and Eddy [40], and Redelings and Suchard [39, 38] for alternative parameterizations.

`gap:character` Specifies that atomic indels are a character state without adding an additional rate class, except under GTR, where `POY5` keeps the unrestricted nature of the model intact, and each indel-nucleotide rate has its own class (e.g. A $\leftrightarrow$ –, C $\leftrightarrow$ –).

`gap:coupled` Specifies that atomic indels are a character state, with rates of nucleotide-to-indel substitution constrained to be equal to one another. For example, the Q-matrix

$$Q = \begin{bmatrix} -3\alpha-\beta & \alpha & \alpha & \alpha & \beta \\ \alpha & -3\alpha-\beta & \alpha & \alpha & \beta \\ \alpha & \alpha & -3\alpha-\beta & \alpha & \beta \\ \alpha & \alpha & \alpha & -3\alpha-\beta & \beta \\ \beta & \beta & \beta & \beta & -4\beta \end{bmatrix}$$

specifies a Jukes-Cantor matrix augmented to estimate nucleotide-to-indel substitutions as a coupled parameter separate from nucleotide-to-nucleotide substitutions. In accordance with other likelihood implementations, the entries of the Q-matrix are normalized to scale their mean rate to 1.

`gap:missing` Specifies that indels be treated under standard assumptions, thus making analyses in POY for a given model and alignment comparable to other implementations such as RAxML [46] and GARLI

[74]. It applies to static or prealigned data only. This is the default of POY5, therefore for dynamic likelihood analyses, either gap:character or gap:couple must be chosen, or else an error will be reported.

Model choice Defines the reversible stochastic Markov model of evolution. POY5, as in other standard likelihood implementations, employs globally homogeneous models [26], which estimate rates using a single matrix over the entire topology. POY5 requires a model of character substitution to be specified during the transformation to likelihood characters.

Model parameters can be specified by following the model name with a colon, left parenthesis, and a list of numbers for the particular model, followed by a closing parenthesis, (i.e. k2p:(2.0)).

NOTE

All matrices are normalized so the mean rate is one. This allows branch lengths to represent the expected number of transformations averaged over the sequence length. This is standard in most (if not all) likelihood programs.

jc69/neyman The model of Jukes and Cantor [27] under nucleotide data, or Neyman [35] in general. A single rate class. All equilibrium frequencies are constrained to be equal. This model can be applied to any size alphabet.

f81 The model of Felsenstein 1981 [16]. A single rate class, proportional to the equilibrium frequencies. This model can be applied to any size alphabet.

k2p/k80 The model of Kimura [29]. Transitions and transversions are constrained to have separate rate classes. All equilibrium frequencies are constrained to be equal.

f84 The model of Felsenstein 1984 [15]. Rates are constrained as in k2p, but are also proportional to the equilibrium frequencies.

hky85 The model of Hasegawa *et al.* [24]. Transitions and transversions are constrained to have separate rate classes. Rates are also proportional to equilibrium frequencies.

`tn93` The model of Tamura and Nei [51]. Three rate classes: transversions (purine/pyrimidine), and independent classes for purine-purine and pyrimidine-pyrimidine transitions. Rates are also proportional to equilibrium frequencies.

`gtr` The model of Tavaré [52]. The most general reversible, stationary model of nucleotide substitution, and the one most frequently selected by information criteria. All rates are unconstrained. This model works for any size alphabet, but note that there are $\frac{N(N-1)}{2}$ rates, where N is the alphabet size.

`ncm` The model of Tuffley and Steel [53], is an extension of the Neyman model, but in this case each character is free to evolve at its own rate on every edge of the tree. This model can only be used with static data, including prealigned sequences. Because these characters evolve at their own rate, `gamma` options are ignored.

`file:STRING` An external file containing a matrix of estimated rate values. The diagonal values are ignored and may take any value. As mentioned above, the rates of the matrix are normalized. The model is not optimized further. For example:

$$\begin{array}{ccccc} 0.0 & 2.0 & 1.0 & 2.0 & 4.0 \\ 2.0 & 0.0 & 0.0 & 1.0 & 4.0 \\ 1.0 & 2.0 & 0.0 & 2.0 & 4.0 \\ 2.0 & 1.0 & 2.0 & 0.0 & 4.0 \\ 4.0 & 4.0 & 4.0 & 4.0 & 0.0 \end{array}$$

Will be normalized so that the mean rate is 1, and so that the rows sum to 0.0 to become (with equal equilibrium frequencies),

$$\begin{array}{rrrrr} -0.865 & 0.192 & 0.096 & 0.192 & 0.384 \\ 0.192 & -0.865 & 0.192 & 0.096 & 0.384 \\ 0.096 & 0.192 & -0.865 & 0.192 & 0.384 \\ 0.192 & 0.096 & 0.192 & -0.865 & 0.384 \\ 0.384 & 0.384 & 0.384 & 0.384 & -1.538 \end{array}$$

`custom:STRING` An external file wherein the rate class constraints of the Q-matrix are specified. As previously mentioned, diagonal elements are ignored, but necessary and any character can be used as a placeholder

(below we use a dash, '–'). Any ASCII character can be used to define an associated rate, but the matrix must be symmetric. For example, the following matrix will create a model where three parameters are ultimately optimized (given that the base parameter is not optimized):

$$\begin{array}{ccccc} - & a & c & d & e \\ a & - & c & d & e \\ c & c & - & d & e \\ d & d & d & - & e \\ e & e & e & e & - \end{array}$$

Model selection As an alternative to selecting a model directly (as above), information theoretic approaches to model selection are provided. These include the Akaike Information Criterion (AIC) [1], the corrected AIC (AICc) [48], and the Bayesian Information Criterion (BIC) [44].

These methods, which require a tree in memory, will optimize the model to the tree of all available models, and select the best (based on the information criterion selected) to be kept in memory. A report is also printed to show the scores and analysis of model selection (see Table 3.1). Delta, the difference between the row and the best score are given, along with a weight (ω_i) given by,

$$w_i = \frac{e^{\frac{-\Delta_i}{2}}}{\sum_{r=1}^{R} e^{\frac{-\Delta_r}{2}}}$$

where R equals the number of models and i equals the ith model.

`aic[:STRING ]` Uses the AIC (Akaike Information Criteron) to determine the best model in the set. The `STRING` is optional, and if specified, `POY5` will output the table of information to this file.

The formula to calculate the AIC is given by,

$$AIC = -2\log(\mathcal{L}(\hat{\theta}|data)) + 2K$$

Where K is the number of parameters for the model, $\hat{\theta}$ and n are the number of characters.

NOTE

If $\frac{n}{k} < 40.0$ then a warning message will be reported suggesting the use of AICc (below).

Table 3.1: Example of `POY5` output showing scores and analysis of model selection using `aicc`. Model type, negative log likelihood values (-ℓ), penalty parameter which include the number of branches (K), number of characters (n), AICc values, AICc differences (Δ), Akaike weights (ω) and cumulative Akaike weights (Cum (ω)) are reported. In this example, JC69 garners the best Information Theoretic score.

Model	-ℓ	K	n	AICc	Δ	ω	Cum(ω)
JC69	538.819	31	49	1256.345	0.000	0.863	0.863
JC69+G	532.219	32	49	1260.438	4.093	0.111	0.975
K81	533.777	32	49	1263.554	7.209	0.023	0.998
K81+G	526.763	33	49	1269.127	12.782	0.001	1.000
F81	539.512	34	49	1317.024	60.678	5.754e-14	1.000
HKY	532.270	35	49	1328.387	72.041	1.961e-16	1.000
F84	533.315	35	49	1330.477	74.131	6.899e-17	1.000
F81+G	533.682	35	49	1331.210	74.865	4.780e-17	1.000
HKY+G	526.033	36	49	1346.067	89.721	2.840e-20	1.000
F84+G	526.518	36	49	1347.037	90.691	1.749e-20	1.000
TN93	531.899	36	49	1357.799	101.453	8.052e-23	1.000
TN93+G	525.841	37	49	1381.318	124.973	6.290e-28	1.000
GTR	530.724	39	49	1486.116	229.770	1.102e-50	1.000
GTR+G	523.365	40	49	1536.731	280.386	1.125e-61	1.000

`aicc[:STRING ]` This is the second-order bias correction used for small samples sizes (as mentioned in the statement above, when $\frac{n}{k} < 40.0$). The correction is,

$$AIC_c = AIC + \frac{2K(K+1)}{n-K-1}$$

The `STRING` is optional, and if specified, `POY5` will output the table of information to this file.

`bic[:STRING ]` Bayesian information criterion, although included in the Information Theoretic approaches, it is in fact not related to information theory. BIC is given by,

$$BIC = -2\ln(\mathcal{L}(\hat{\theta}|data)) + K\log(n)$$

The `STRING` is optional, and if specified, `POY5` will output the table of information to this file.

Rate distributions Specifies the distribution of rates among sites. Rate variation distributions allow multipliers to be applied to separate groups of characters. This additional parameterization frequently improves estimated likelihood scores. Commonly employed distributions for the parameterization of among-site variation include the discrete gamma distribution (Γ) [72] and this distribution with an additional invariant rate class, the theta distribution (Θ) [22].

`rates:gamma:(N,`α`)` Applies a discrete gamma distribution of N classes, $\gamma(\alpha, \beta)$, of rate variation across characters. The mean rate for the distribution is set to 1.0 (i.e. $\alpha = \beta$), thus one parameter is estimated. The default for N is 4. If α is not given the parameter is optimized.

`rates:theta:(N,`α`,%)` Applies a discrete gamma distribution with an additional invariable rate class (thus $N + 1$ rate classes in total). The default for N is 4. α and % are optimized if not given.

`rates:none` Applies a single rate category to all sites.

NOTE

Under MAL, the rates are averaged across the discrete distribution, while under MPL we select the best rate class for the assignment of each character. Because dynamic likelihood characters' alignment matrix would select one rate class for alignment, and because this procedure is equivalent to a constant multiplication of the branch length, dynamic MPL characters do not support rate distribution. Rate distributions are ignored if applied during a transform and a warning to that affect is reported.

NOTE

It should be noted that although many researchers apply both a gamma and theta in likelihood analyses, the parameter values inferred are correlated [49]. It has therefore been recommended that a proportion of invariant sites be "pseudo"-estimated by increasing the number of discrete categories for the gamma distribution to account for very low rate character groups. The mean of each class is used to define the rate for the discrete category.

Defaults

`transform()` If no arguments are given, this command does nothing.

Examples

- `transform(all,(tcm:(1,1)))`
  Applies the transformation cost matrix (1,1) to all characters, meaning that substitutions and gaps receive the same weight.

- `transform(all,(tcm:"molmatrix"))`
  Applies the character transformation matrix "molmatrix" to all characters.

- `transform(tcm:(1,1),gap_opening:1)`
  Applies the transformation cost matrix and the gap opening cost to all characters. In this example the cost for substitutions is `1`, the gap opening cost is `2` (`1` set by `gap_opening` + `1` set by `tcm`), and the gap extension cost is `1` (set by `tcm`).

- `transform(tcm:(2,2),ti:(1,1,1,1,0),td:(1,1,1,1,0))`
  Assigns to all characters the symmetric transformation cost matrix with cost `2` for every indel and substitution, but for those insertions and deletions at the ends of the sequences, the cost assigned will only be `1`.

- `transform(tcm:("some_tcm_file",level:(1,first)))`
  This command applies the cost matrix as specified in tcm file "some_tcm_file". The heuristic median calculations are set to level `1` and tie breaks are broken by choosing the first median state examined.

- `transform(static,(weightfactor:2))`
  This command reweights all the static homology characters by a multiplicative factor of `2`, while keeping the weighting scheme that has been specified before.

- `transform(static,(weight:4.2))`
  Applies the same weight (a float value `4.2`) to all static homology characters.

- `transform(dynamic,(weight:4))`
  Applies the same weight (an integer value `4`) to all dynamic homology characters.

- `transform(names:("test.ss:*"),(weight:3))`
  Weights all the morphological characters (`*`) in the file `test.ss` by an integer value of 3.

- `transform(names:("gen1","gen4"),(fixed_states))`
  Transform only specified sequence characters (`gen1` and `gen4`) into fixed states characters.

- `transform((all,(tcm:(1,1))),(names:("gen1","gen2"),(static_approx)),(names:("gen3"),(tcm:"molmatrix")))`
  Applies the substitution and indel costs `1` to all characters, then applies static approximation using that tcm to characters in files `gen1` and `gen2`. For the file gen3, it invokes a different transformation cost matrix, contained in the file `molmatrix`. Beware that the file name should be exactly as it was reported with `report(data)`, which differs from the source file name (`report(data)` reports files as fileX:N).

- `transform((all,(tcm:(1,1))),(names:("gen1:3","gen2:10","gen3:1","gen4:5"),(static_approx)),(names:("gen5","gen6"),(tcm:"Molmatrix1")))`
  Applies `tcm (1,1)` to all characters, then applies static approximation to the sequence data contained in files `gen1`, `gen2`, `gen3`, and `gen4` according to this transformation cost matrix, and applies the custom transformation cost matrix contained in the file `Molmatrix1` to the sequence data contained in files `gen5` and `gen6`.

- `transform(names:("mycustom.fas"),(tcm:("mycustom.mat",2)))`
  This examples transforms the cost matrix used to optimize custom_alphabet character "mycustom.fas" to "mycustom.mat" and uses heuristic level 2.

- `transform(fixed_states)`
  Transforms all sequence characters into fixed states characters.

- `transform(likelihood:(jc69,rates:gamma:(2),mpl,priors:equal))`
  Transforms the characters to likelihood characters, using a `jc69` + `gamma`:(2) model, with equal equilibrium frequencies under MPL. In this model, gaps are treated as "missing" (the default value).

- `transform(likelihood:(tn93,rates:theta:(4),gap:coupled,mpl,priors:estimate))`

This command transforms the characters to likelihood characters, using a `tn93` + `theta`:(4) model, with estimated equilibrium frequencies under MPL. In this model, indels are treated as `coupled`.

- `transform(fixed_states:("shrimp",ignore_polymorphism))`
  This example shows the optional arguments for a Mauve-based `fixed_states` analysis. Here, Mauve genome alignment files will be generated with the names "shrimp_i_j.alignment" where i and j are median states. Sequence ambiguities will not be resolved to generate additional medians beyond those determined by the data.

- `transform(search_based:(("mycustom_char1.fas","mycustom_extra1.fas"),("mycustom_char2.fas","mycustom_extra2.fas")))`
  This command adds the sequences found in "mycustom_extra1.fas" and "mycustom_extra2.fas" to `fixed_states` characters "mycustom_char1.fas" and "mycustom_char2.fas" respectively.

- `transform(chromosome:(annotate:(mauve,35.0,0.35,0.01,0.10)))`
  This command specifies the use of Mauve annotation with 35.0 for quality of LCB, 35% coverage of all sequences by LCB's, 1% min length of LCB (100 for length 10,000 sequence), and 10% max length (1000 for length 10,000 sequence).

- `transform(seq_to_chrom:(locus_indel:(50,1.0),min_seed_length:15))`
  All applicable (i.e. sequence) data are transformed into chromosome data and the locus-level gap opening cost is set at `50` with a gap extension cost at `1.0`.

3.3.28 use

Syntax

use`(STRING)`

Description

Restores from memory the state of a `POY5` session (that includes character data, selections, trees, all other data and specifications) that had previously been saved during the session using the command `store` (Section 3.3.25). The recalled session replaces the current session. The string argument specifies the name of the stored state.

In combination with `store` (Section 3.3.25), the command `use` is useful for exploring alternative cost regimes and terminal sets within a single `POY5` session.

Examples

- ```
  store("initial_tcm")
  transform(tcm:(1,1))
  use("initial_tcm")
  ```
  The first command, `store`, stores the current characters and trees under the name `initial_tcm`. The second command, `transform`, changes the cost regime of molecular characters, effectively changing the data being analyzed. However, the third command, `use`, recovers the initial state stored under the name `initial_tcm`.

See also

- `store` (Section 3.3.25)
- `transform` (Section 3.3.26)

3.3.29 version

Syntax

version`()`

Description

Reports the `POY5` version number in the output window of the *ncurses* interface, or to the standard error in the *flat* interface.

Examples

- `version ()`

3.3.30 wipe

Syntax

wipe`()`
  ```

Description

Deletes the data stored in memory (all character data, trees, *etc.*).

Examples

- `wipe ()`

Chapter 4

POY5 Heuristics: A Practical Guide

4.1 Introduction

As the level of phylogenetic analysis increases—from individual loci, to chromosomes, to genomes containing multiple chromosomes—so too does the computational complexity. In POY5, a significant increase in computational time results from combining cladogram searching with co-optimization of nucleotide pairwise alignments, rearrangements of loci within a chromosome, and rearrangements of chromosome fragments within the genome. As a result, a phylogenetic analysis involves a set of nested computationally "hard" (NP-complete) problems that makes finding exact solutions extremely unlikely. In addition, increasing sequence length heterogeneity (at the levels of nucleotides, loci, and chromosomes) and the ever-growing sizes of datasets, further contribute to computational complexity.

To cope with the problem of computational intractability, and hence, the speed of the analyses, POY5 can employ a battery of approximate, or heuristic methods that function at different levels of the analysis. As with all heuristic procedures, a tradeoff is involved: a substantial decrease in execution time comes at a price of obtaining solutions with reduced optimality (however, the extent of the tradeoff is difficult to evaluate in the analyses of real datasets). Therefore, it becomes important to understand the combined effect of different heuristic methods, so that the chosen search strategy balances the computational time with a "reasonable" quality of the result.

Here, we provide general guidelines for using different heuristic methods, exploring their combined effect, and suggesting the choice of parameters that

can be explored to provide the best result for specific cases. Real datasets differ greatly in size and complexity, so that no single optimal strategy can be suggested. These guidelines, however, should enable the investigator to design an efficient strategy that can be tailored to the peculiarities of a given dataset.

In addition to heuristic methods, this chapter attempts to assist with the selection of transformation cost regimes. Alternative cost regimes can significantly affect the outcome of the analysis, this becomes particularly apparent in dealing with large, genome-level datasets, where multiple cost regimes are used simultaneously to specify transformations at different levels of analysis. Many problems stem from the difficulty in selecting the most reasonable combination of parameters for the optimization of DNA sequence data at the levels of nucleotides, loci, and chromosomes.

4.2 Preparing data files for analysis

If molecular sequence files contain incomplete sequences, it is **highly** recommended that the file be partitioned prior to analysis in `POY5`. Partitioning or fragmenting the data can help to ameliorate the effects of poor sequences, missing data, or the lack of overlap of sequences—as is very often the case when incorporating sequences that were downloaded from an online database such as GenBank (as different studies may have utilized different priming regions) (Figure 4.1).

At the level of nucleotides, individual fragments in a locus can be separated by pound symbols ("#") or contained in individual files (that is, treated as partitions). When "#'s" are used, their number must be the same across homologous sequences. Alternatively, the argument of `auto_sequence_partition` of the command `transform` (Section 3.3.26) can partition the data. This command evaluates each fragment in the data file and if a long region appears to have no indels, then the fragment is broken inside that region. `auto_sequence_partition` works best when primer flanking regions are available. When primer flanking regions are not available it is recommended that the "#"s be inserted manually (Figure 4.2 and Figure 4.3).

4.3 Data treatment

Direct optimization (see *Character optimization* section below) involves comparing potential nucleotide homologies between two sequences. Consequently, the time it takes is proportional to the product of the lengths of the sequences

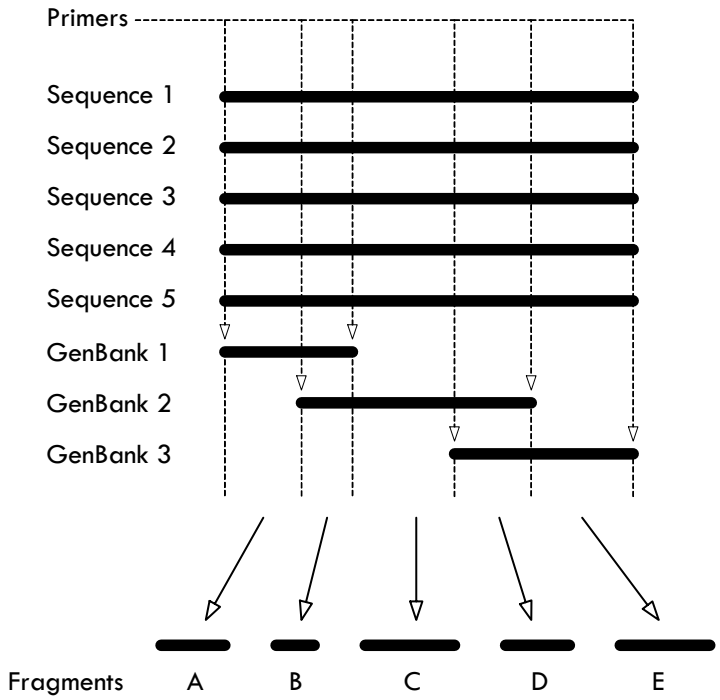

Figure 4.1: Separation of eight sequences into fragments, using flanking primers as a guide. This partitioning will help to ameliorate the effects of missing data in the sequences `GenBank1`, `GenBank2`, and `GenBank3`.

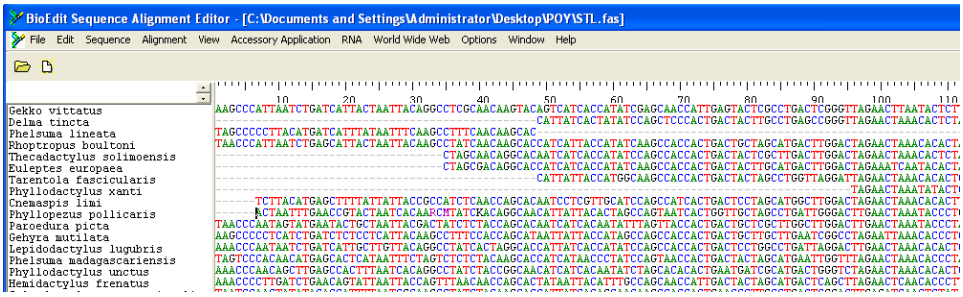

Figure 4.2: A nucleotide sequence alignment visualized in BioEdit.

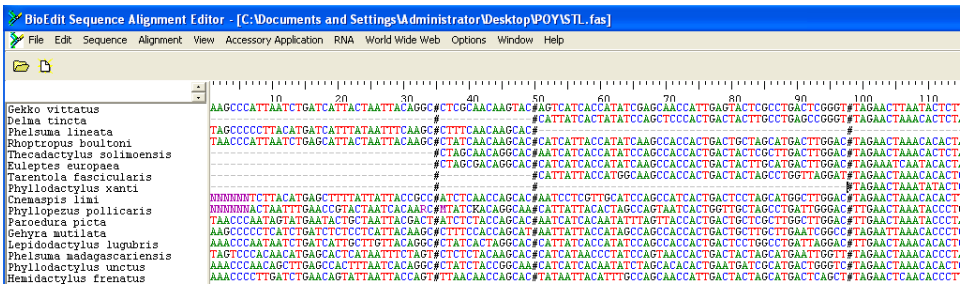

Figure 4.3: A nucleotide sequence alignment visualized in BioEdit, that has been 'chopped', via insertion of "#" symbols. "Cnemaspis_limi" and "Phyllopezus_pollicaris" have been modified with the insertion of N's. Note the lack of overlap between "Delma_tincta" and "Phelsuma_lineata".

Table 4.1: Heuristics Guide: Data Treatment.

Level of analysis	Heuristic	Implementation
Nucleotides and amino acids	Fragment sequences	Manually separate fragments with # symbols or transformed using `auto_sequence_partition`
Locus	Fragment chromosome	Manually insert pipes ("|") separating loci
Chromosomes	N/A	N/A

compared ($O(n^2)$) [68]. This procedure can be time consuming for long sequences and for those DNA fragments that greatly differ in length. In cases where unambiguous (such as long, completely conserved regions) sequence fragments can be identified, partitioning the long sequences into smaller fragments delimited by these regions can significantly reduce computational time. Such economy is reached because nucleotide homologies are not examined over the separate partitions. This strategy assumes that the fragments are mutually exclusive and are homologous across terminals.

As mentioned previously, nucleotide sequences can be partitioned by inserting pound symbols ("#"). Fragmenting long sequences greatly accelerates searching. At the chromosome level, individual loci can be separated by pipes ("|") (see Table 4.1).

4.4 Character optimization

Minimizing overall cladogram cost for unaligned sequences is an NP-Hard optimization that is dependent on the lowest cost assignment of HTU sequences. POY implements Direct Optimization (*DO*) [59]; Fixed-States Optimization (*FSO*) [60]; Iterative Pass Optimization (*IPO*) [65]; and Search-Based Optimization (*SBO*) [66] heuristics to determine the set of HTU sequences comprising the internal nodes of each cladogram constructed.

DO decomposes the problem into a series of two-node comparisons, calculating locally optimal solutions, which generates the total cladogram cost. An advantage of direct optimization is that it allows for the exploration of a large diversity of putative homologies and selects the scheme that yields the best solution. This is useful in analyzing sequences of different length, where site-to-site homologies are uncertain. Because the procedure is based on a greedy algorithm, it requires multiple iterations (independent initial cladogram builds) and extensive tree searches to reach a potentially global minimum.

In contrast, fixed-states optimization does not calculate HTU sequences but rather optimizes those observed in terminal taxa. These internal node sequences then are diagnosed using dynamic programming based on a matrix of edit costs between sequences. In the fixed-states implementation, cladogram optimization is independent of sequence lengths after initial state cost calculation, and as the number of sequences increase so to does the pool from which the HTU sequences are drawn, thereby improving cladogram cost estimation (this can be further improved using additional potential median sequences via *SBO*). Due to these properties fixed-states optimization is recommended as an initial approximation strategy for large data sets with large variation in sequence length.

Further approximations and economies can be achieved by varying parameters of commands, such as selecting a limited subset of trees for subsequent analysis limiting the number of replicates, and examining intermediary results from an interrupted analysis.

4.5 Tree searching

Heuristic approaches to cladogram searching include random addition of taxa, branch swapping (TBR and SPR), simulated annealing (the ratchet and tree-drifting), and genetical algorithms (tree fusing). These techniques, frequently used in combination, allow a more efficient exploration of tree space

and provide the means of finding more globally optimal solutions. These methods are widely used in phylogenetics [18, 69], although POY5 implements additional modifications of these procedures.

Typical search strategy in POY5 involves consecutive application of tree search algorithms that begin with generating multiple, randomly selected starting points [Random Addition Sequences (RAS) or Wagner trees]. The importance of multiple starting trees cannot be overemphasized and a successful search will maximize the number of RAS. However, making a tree search more exhaustive by increasing the number of starting trees comes at a price of increased computation time. Therefore, it is advised here to estimate the amount of time it takes to complete a single replicate and takes this information in consideration when designing a more exhaustive strategy. The number of replicates used by POY5 practitioners for datasets of moderate size (70-100 terminals) ranges from 100 to 250 (or approximately n times the number of terminals. Here are some examples of search strategies:

RAS+SPR/TBR+Ratchet The strategy is for a thorough search for a data set of 100 or fewer taxa. A diversity of starting points is generated by multiple RAS, each refined by a local search (TBR or a combination of SPR and TBR, the latter is an efficient default strategy in POY5). Ratcheting is used to examine tree space that potentially has not been explored by the local searches.

RAS+SPR/TBR+Ratchet+Tree Fusing Adding a tree fusing step allows for combining the best subtrees of cladograms that can potentially yield a tree of shorter length. Empirical studies show that adding tree fusing after replicate rounds enhances the results only when dealing with data sets with more than 50 taxa.

RAS+SPR/TBR+Ratchet+Tree Drifting+Tree Fusing Tree Drifting can be used in place of or in addition to the Ratchet.

Input Trees+SPR/TBR+Ratchet+Tree Drifting+Tree Fusing For more exhaustive searches, the best trees obtained from the initial searches using the strategies outlines above, can be used as input trees for subsequent analyses. In doing so, the RAS step can be omitted because searching starts with locally optimal trees.

The aggressiveness of searches can be adjusted by varying parameters of the branch swapping, ratchet, tree fusing, and tree drifting commands.

Further economies can be reached by using a combination of different character optimization methods. For example, initial searches can be conducted under the faster static approximation (that converts sequence data into static homology characters; see *Character optimization* section), whereas the final refinement can be performed using *DO* or *IPO*.

4.6 Transformation cost regimes

In analyses at the level of nucleotides, here are some general approaches to selecting transformation cost regimes most commonly used by `POY5` practitioners:

Equal costs This approach assigns the same cost to all substitutions and indels, and does not take into account gap extension cost. For rationale for using this cost regime see Frost et al. [19].

Parameter sensitivity analysis This method, suggested by Wheeler [58], explores the effect of varying transformation costs by comparing results of analyses conducted under different cost regimes. Partition incongruence can subsequently be computed for each cladogram and the parameter set that minimizes incongruence is selected as best.

More specifically, sequence optimization parameters depend on the relative costs of nucleotide- and locus-level transformations. Nucleotide-level transformations are specified by the `tcm` argument, the locus-level rearrangements are specified by `locus_breakpoint` or `locus_inversion` costs. If `locus_level` rearrangement costs are extremely high, few rearrangements will be employed. On the other hand, if their cost is very low (equal or slightly above that of the nucleotide-level rearrangements), rearrangements can be frequent.

When DNA sequence data is combined with morphological data, the cost for morphological character transformations often is set to be the same as for nucleotide substitutions or indels.

4.7 Likelihood Analyses

The analysis of sequence data under likelihood, whether prealigned (static) or not, can be significantly more time consuming than similar analyses under parsimony. A basic strategy to improve execution times under likelihood is to perform initially less complex analyses and build up through a series

of increasingly more complex procedures until the desired level of search complexity is achieved.

Parsimony initial pass This approach begins with an initial build and search (of arbitrary complexity) under parsimony before transforming to a likelihood model and diagnosing the topology. This procedure saves time by avoiding RAS builds and swapping under likelihood. A potential caveat of this heuristic is that a parsimony-optimal topology may be far in tree space from the likelihood-optimal topology, and significant swapping after transformation to likelihood may be necessary.

Rough parameter estimation The granularity of model parameter estimation can be increased. For example, under a traditional likelihood-based swap, all branches (regardless of the distance from the join site) are re-optimized, and the model parameters are re-optimized after every swap. Time can be saved by optimizing the model only if the cost of a join is within a threshold number of the current best cost, or by optimizing branches within a specified distance of the join region. RAxML takes advantage of this heuristic method by using its GTRCAT model for topology search, and a more refined GTRGAMMA for final parameter estimation on the best topology.

Floating point granularity The coarseness of floating point calculations can be increased to limit the time spent on optimizing parameter values during swapping or even during transformation to likelihood characters. Coarse granularity operates by limiting both the precision calculated and the number of optimization iterations conducted when estimating parameter values. A caveat of this heuristic is that coarse granularity may adversely affect analyses for which multiple topologies and branch length schemes are close to equally optimal. Additionally, likelihood scores under coarse granularity are not comparable to those of other likelihood programs, and a full optimization should be conducted on the final topology.

Optimization schedule The stringency of the model parameter optimization schedule can be decreased. For example, under a traditional likelihood-based swap, all branch lengths (regardless of the distance from the join site) and model parameters are re-optimized after every swap. Time can be saved by limiting this schedule and optimizing the model only if the cost of a join is within a threshold number of the current best cost, or by optimizing branches within a specified distance

of the join region. A caveat of this heuristic is that the optimal schedule is difficult to predict, and is likely to be strongly data-dependent.

Limited rearrangement neighborhoods The size of the rearrangement neighborhood (variants in complexity between NNI and TBR) can be restricted in early search stages of topology search under likelihood. This approach restricts the number of calculations during rearrangements, whereas increasing granularity restricts the time spent on a given calculation. Once solutions have been identified that are suspected to be near optimal, more exhaustive model estimation and search strategies can be performed. A caveat of this heuristic is similar to that of a parsimony initial pass–limiting rearrangement neighborhoods may identify a local optimum under likelihood that may be difficult to escape.

Other possibilities for heuristics exist, including alternation between optimality criteria on static characters (transformations back and forth between static likelihood and parsimony), between variants of one optimality criterion (between MPL and MAL), or between character assumptions (between four- and five-state variants of a given model).

Chapter 5

POY5 Tutorials

These tutorials are intended to provide guidance for more sophisticated applications of POY5 that involve multiple steps and a combination of different commands. Each tutorial contains a POY5 script that is followed by detailed commentaries explaining the rationale behind each step of the analysis. Although these analyses can be conducted interactively using the *Interactive Console* or running separate sequential analyses using the *Graphical User Interface*, the most practical way to do this is to use a POY5 script (see *POY5 Quick Start* for more information (Section 2.3).

NOTE

It is important to remember that the numerical values for most command arguments will differ substantially depending on type, complexity, and size of the data. Therefore, the values used here should not be taken to be optimal parameters.

The tutorials use sample datasets that are provided with POY5 installation but can also be downloaded from the POY5 site at:

http://www.amnh.org/our-research/computational-sciences/research/projects/systematic-biology/poy

The minimally required items to run the tutorial analyses are the POY5 application and the sample data files. Running these analyses requires some familiarity with the POY5 interface and command structure that can be found in the preceding chapters.

5.1 Combining Search Strategies

The following script implements a strategy for a thorough search. This is accomplished by generating a large number of independent initial trees by random addition sequence and combining different search strategies that aim at exploring local tree space and escaping the effect of composite optima by comprehensively traversing the tree space. In addition, this script shows how to output the status of the search to a log file and calculate the duration of the search.

```
(*search using all data*)

read("9.fas","31.ss")
read(prealigned:(aminoacids:("41_aln.aa"),tcm:"s1t2.mat"))
set(log:"all_data_search.log",root:"t1")
report(timer:"search_start")
build(250)
swap(threshold:5.0)
select(unique)
perturb(transform(static_approx),iterations:15,ratchet:(0.2,3))
select()
fuse(iterations:200,swap())
select()
report("all_trees.tre",trees:(total),"all_trees_cs.pdf",
graphconsensus,"all_trees_diag.txt",diagnosis)
report(timer:"search_end")
set(nolog)
exit()
```

- `(*search using all data*)` This first line of the script is a comment. While comments are optional and do not affect the analyses, they are useful for housekeeping purposes.
- `read("9.fas","31.ss")` This command imports the nucleotide sequence files `9.fas` (in FASTA format), and a morphological data file `31.ss` (in Hennig86 format).
- `read(prealigned:(aminoacids:("41_aln.aa"),tcm:"s1t2.mat"))` This command imports the prealigned amino acids file `41_aln.aa` and sets the transformation cost matrix to be used in calculating the cost of the tree for these data. The tcm file `s1t2.mat` specifies that the cost of substitutions are 1 and that of indels 2.

- `set(log:"all_data_search.log",root:"t1")` The `set` command specifies conditions prior to tree searching. Specifying the log produces a file, `all_data_search.log`, that provides an additional means to monitor the process of the search. The outgroup (`t1`) is designated as the `root`, so that all the reported trees have the desired polarity. By default, the analysis is performed using direct optimization.

- `report(timer:"search_start")` In combination with `report(timer: "search end")`, this commands reports the amount of time that the execution of commands enclosed by `timer` takes. In this case, it reports how long it takes for the entire search to finish. Using `timer` is useful for planning a complex search strategy for large datasets that can take a long time to complete: it is instructive, for example, to know how long a search would last with a single replicate (one starting tree) before starting a search with multiple replicates.

- `build(250)` This command begins tree-building step of the search that generates 250 random-addition sequence Wagner trees. A large number of independent starting points helps to ensure that a reasonable portion of tree space will be examined.

- `swap(threshold:5.0)` `swap` specifies that each of the 250 trees is subjected to alternating SPR and TBR branch swapping routines (the default of `POY5`). In addition to the most optimal trees, all the suboptimal trees found within 5% of the best cost are swapped. This step helps to ensure that the local searches settled on local optima.

- `select(unique)` Upon completion of branch swapping, this command retains topologically unique trees. Contra `select()`, which selects topologically *unique* and *optimal* trees, `select(unique)` selects *all* unique trees, regardless of cost, thus ensuring that a larger tree space is explored.

- `perturb(transform(static_approx),iterations:15,ratchet: (0.2,3))` Having transformed to static data (`static_approx`), 20% of the characters are selected at random and are then upweighted by a factor of 3. This process is repeated 15 times.

- `fuse(iterations:200,swap())` In this step, up to 200 exchanges of subtrees identical in terminal composition but different in topology, are performed between pairs of best trees recovered in the previous step. This is another strategy for further exploration of tree space.

Each resulting tree is further refined by local branch swapping under the default parameters of `swap`. [Note: performing `swap` after every iteration of `fuse` can be computationally expensive for larger datasets. An alternative strategy would include a separate round of `swap` following `fuse`.]

- `select()` Upon completion of branch swapping, this command retains only optimal and topologically unique trees; all other trees are discarded from memory.

- `report("all_trees.tre",trees:(total),"all_trees_cs.pdf", graphconsensus,"all_trees_diag.txt",diagnosis)` This command produces a series of outputs of the results of the search. It includes a file containing best trees in parenthetical notation (`all_trees.tre`) and their costs (`trees:(total)`), a graphical representation (in PDF format) of the strict consensus (`all_trees_cs.pdf`), and the diagnoses for all best trees (`all_trees_diag.txt`).

- `report(timer:"search_end")` This command stops timing the duration of search, initiated by the command `report(timer:"search_start")`.

- `set(nolog)` This command stops reporting to the log file, `all_data_search.log`.

- `exit()` This command ends the `POY5` session.

5.2 Timed Search Analysis

The following script implements a strategy for a search using the timed search option. The timed search option applies a default strategy that performs as many rounds of tree building, followed by TBR branch swapping, parsimony ratchet and tree fusing. When performing a timed search, it is crucial to set the maximum time such that the program has a reasonable amount of time to perform a search. Thus, it is important to have some approximation as to the length of time it would take to perform a single round of searching (e.g. build (1), followed by TBR, ratchet and fusing in the case of a parsimony analysis of DNA sequence data). With this information, the user can then estimate the amount of time necessary to perform a thorough search. The amount of time set for the search is clearly data dependent.

While this tutorial calls for two successive 12 hour timed searches, if more than one processor is available to the user, each of these 12 hour rounds can be further divided into shorter runs, depending on the number of processors available, e.g. if 2 processors are available, then a 6 hour timed search across this number of processors will afford quicker results.

```
(*search using a Timed Search*)

read("9.fas","31.ss")
read(prealigned:(aminoacids:("41_aln.aa"),tcm:"s1t2.mat"))
set(root:"t1")
search (max_time:00:12:00)
select(unique)
report("Run1a.tre",trees)
search(max_time:00:12:00)
select(unique)
report("Run1b.tre",trees)
fuse(iterations:250)
select()
swap(trees:100)
select()
report("Run1c_H86.tre",trees:(hennig,total),"Run1c_cs.tre",
consensus,"Run1c.pdf",graphconsensus)
quit()
```

- `(*search using a Timed Search*)` This first line of the script is a comment. While comments are optional and do not affect the analyses, they are useful for housekeeping purposes.

- `read("9.fas","31.ss")` This command imports the nucleotide sequence files `9.fas` (in FASTA format), and a morphological data file `31.ss` (in Hennig86 format).

- `read(prealigned:(aminoacids:("41_aln.aa"),tcm:"s1t2.mat"))` This command imports the prealigned amino acids file `41_aln.aa` and sets the transformation cost matrix to be used in calculating the cost of the tree for these data. The tcm file `s1t2.mat` specifies that the cost of substitutions are 1 and that of indels 2.

- `set(root:"t1")` The `set` command specifies conditions prior to tree searching. The outgroup (`t1`) is designated as the `root`, so that all the

reported trees have the desired polarity. By default, the analysis is performed using direct optimization.

- `search(max_time:00:12:00)` The `search` command will perform a timed search, performing as many successive rounds of build, swap, ratchet and fusing during the 12 hour period. For this dataset, it has been determined that 24 hours (from two successive rounds of a 12 hour timed search) is sufficient time to perform a thorough search.

- `select(unique)` Upon completion of the timed search, this command retains topologically unique trees. Contra `select()`, which selects topologically *unique* and *optimal* trees, `select(unique)` selects *all* unique trees, regardless of cost, thus ensuring that a larger tree space is explored.

- `report("Run1a.tre",trees)` Having selected all unique trees, these trees are reported to a file. Outputting trees at different stages of longer runs is advisable. These reports can act as checkpoints in case of hardware problem, computer crashes, power outages etc.

- `search(max_time:00:12:00)` A second timed search is performed.

- `select(unique)` All topologically *unique* trees (including suboptimal trees) are selected.

- `report("Run1b.tre",trees)` Having selected all unique trees, these trees are reported to a file.

- `fuse(iterations:250)` In this step, up to 250 exchanges of subtrees identical in terminal composition but different in topology, are performed between pairs of best trees recovered in the previous step. This is another strategy for further exploration of tree space.

- `select()` Upon completion of fusing, this command retains only optimal and topologically unique trees; all other trees are discarded from memory.

- `swap(trees:100)` Submits current trees to a round of SPR followed by TBR. It keeps up to 100 minimum cost trees for each starting tree.

- `select()` Upon completion of branch swapping, this command retains only optimal and topologically unique trees; all other trees are discarded from memory.

- `report(Run1c_H86.tre",trees:(hennig,total),"Run1c_cs.tre", consensus,"Run1c.pdf",graphconsensus)` This command reports a series of outputs of the results of the search. It includes a file containing the most optimal trees in parenthetical notation (`Run1c_H86.tre`) with the associated costs (`trees:(hennig,total)`. These trees have been prepended with `tread` and are separated by asterisks. In addition, a strict consensus in parenthetical notation (`Run1c_cs.tre`) and a graphical representation of this strict consensus (`Run1c.pdf`) are also output.

- `quit()` This command ends the `POY5` session.

5.3 Iterative Pass Analysis

The following script implements a strategy for a thorough search under iterative pass optimization. Iterative pass optimization is a very time consuming procedure that makes it impractical to conduct under this kind of optimization (save for very small datasets that can be analyzed within reasonable time). The iterative pass, however, can be used for the most advanced stages of the analysis for the final refinement, when a potential global optimum has been reached through searches under other kinds of optimization (such as direct optimization). Therefore, this tutorial begins with importing an existing tree (rather than performing tree building from scratch) and subjecting it to local branch swapping under iterative pass.

```
(*search using all data under ip*)

read("9.fas","31.ss")
read(prealigned:(aminoacids:("41_aln.aa"),tcm:"s1t2.mat"))
read("Run1c_H86.tre")
set(iterative:approximate:2)
swap(around)
select()
report("all_trees.tre",trees:(total),"all_trees_cs.pdf",
graphconsensus,"all_trees_diag.txt",diagnosis)
exit()
```

- `(*search using all data under ip*)` This first line of the script is a comment. While comments are optional and do not affect the analyses, they are useful for housekeeping purposes.

- `read("9.fas","31.ss")` This command imports the nucleotide sequence files `9.fas` (in FASTA format), and a morphological data file `31.ss` (in Hennig86 format).
- `read(prealigned:(aminoacids:("41_aln.aa),tcm:"s1t2.mat"))` This command imports the prealigned amino acids file `41_aln.aa` and sets the transformation cost matrix to be used in calculating the cost of the tree for these data. The tcm file `s1t2.mat` specifies that the cost of substitutions are 1 and that of indels 2.
- `read("Run1c_H86.tre")` This command imports a tree file, `Run1c_H86.tre`, that contains a tree from a previous analyses.
- `set(iterative:approximate:2)` This command sets the optimization procedure to iterative pass such that approximated three dimensional alignments generated using pairwise alignments will be considered. The program will iterate either two times, or until no further tree cost improvements can be made.
- `swap(around)` This commands specifies that the imported tree is subjected to alternating rounds of SPR and TBR branch swapping (the default of `POY5`) following the trajectory of search that completely evaluates the neighborhood of the tree (by using `around`).
- `select()` Upon completion of branch swapping, this command retains only optimal and topologically unique trees; all other trees are discarded from memory.
- `report("all_trees.tre",trees:(total),"all_trees_cs.pdf", graphconsensus,"all_trees_diag.txt",diagnosis)` This command produces a series of outputs of the results of the search. It includes a file containing best trees in parenthetical notation (`all_trees_ip.tre`) and their costs (`trees:(total)`, a graphical representation (in PDF format) of the strict consensus (`all_trees_cs.pdf`), and the diagnoses for all best trees (`all_trees_diag.txt`).
- `exit()` This command ends the `POY5` session.

5.4 Calculating supports: Bremer

This tutorial illustrates the calculation of Bremer support values for trees constructed from dynamic homology characters. It is strongly recommended

that this more exhaustive approach is used for calculating Bremer supports rather than simply using the `calculate_supports(bremer)` defaults. As this tutorial utilizes the `visited` option, this tutorial can not be run in parallel.

```
(*Bremer support part 1: generating trees*)

read("9.fas","31.ss")
read(prealigned:(aminoacids:("41_aln.aa"),tcm:"s1t2.mat"))
set(root:"t1")
read("Run1c_H86.tre")
swap(all,visited:"tmp.trees",timeout:3600)
select()
report("bremertrees.tre",trees)
wipe()

(*Bremer support part 2: Bremer calculations*)

read("9.fas","31.ss")
read(prealigned:(aminoacids:("41_aln.aa"),tcm:"s1t2.mat"))
set(root:"t1")
read("bremertrees.tre")
report("Bremer_trees.pdf",graphsupports:bremer:"tmp.trees")
exit()
```

- `(*Bremer support part1: generating trees*)` This first line of the script is a comment. While comments are optional and do not affect the analyses, they are useful for housekeeping purposes.

- `read("9.fas","31.ss")` This command imports the nucleotide sequence files `9.fas` (in FASTA format), and a morphological data file `31.ss` (in Hennig86 format).

- `read(prealigned:(aminoacids:("41_aln.aa"),tcm:"s1t2.mat"))` This command imports the prealigned amino acids file `41_aln.aa` and sets the transformation cost matrix to be used in calculating the cost of the tree for these data. The tcm file `s1t2.mat` specifies that the cost of substitutions are 1 and that of indels 2.

- `set(root:"t1")` The `set` command specifies conditions prior to tree searching. The outgroup (`t1`) is designated as the `root`, so that all the reported trees have the desired polarity.

- `read("Run1c_H86.tre")` This command will read in the tree file `Run1c_H86.tre` that was generated in `Tutorial 2`.
- `swap(all,visited:"tmp.trees",timeout:3600)` The `swap` command specifies that each of the trees be subjected to an alternating SPR and TBR branch swapping routine (the default of `POY5`). The `all` argument turns off all swap heuristics. The `visited:"tmp.trees"` argument stores every visited tree in the file specified. Although the visited tree file is compressed to accommodate the large number of trees it will accumulate, the argument `timeout` can be used to limit the number of seconds allowed for swapping also limiting the size of the file (although this has potential to inflate the Bremer values reported). Alternatively, the `swap` command can be performed as a separate analysis and terminated at the users discretion to maximize the number of trees generated. [Note: as this `visited` file is a compressed file, it will appear to contain 'garbage' when opened. To access a 'human readable' file, it must be uncompressed.]
- `select()` Upon completion of branch swapping, this command retains only optimal and topologically unique trees; all other trees are discarded from memory.
- `report("bremertrees.tre",trees)` This command will save the optimal and unique tree(s) to a file `bremertrees.tre`.
- `wipe()` This command eliminates all the data stored in memory.

- `(*Bremer support part 2: Bremer calculations*)` A comment indicating the intent of the commands which follow.
- `read("9.fas","31.ss")`
- `read(prealigned:(aminoacids:("41_aln.aa"),tcm:"s1t2.mat"))`
- `set(root:"t1")` The `set` command specifies conditions prior to tree searching. The outgroup (`t1`) is designated as the `root`, so that all the reported trees have the desired polarity.
- `read("bremertrees.tre")` This command imports the tree file `bremertrees.tre` for which the support values will be generated. It is important to only read the selected `"bremertrees.tre"` file rather

than the expansive `"tmp.trees"` file which will be used in bremer calculations.

- `report("Bremer_trees.pdf",graphsupports:bremer:"tmp.trees")` The `report` command in combination with `graphsupports` and a file name, generates a pdf file (`Bremer_trees.pdf`) with bremer values for the selected trees held in `tmp.trees`.

- `exit()` This command ends the `POY5` session.

5.5 Calculating supports: Jackknife

This tutorial illustrates the calculation of Jackknife support values for trees constructed from static homology characters— these characters are prealigned. Although it is possible to calculate Jackknife support values for trees constructed using dynamic homology characters, it is highly recommended against doing so as resampling of dynamic characters occurs at the fragment (rather than nucleotide) level (e.g. calculating jackknife supports for a dataset that contains a single fragment would be meaningless).

```
(*Jackknife support for static homology trees*)

read(prealigned:("28s.aln",tcm:(1,2)))
set(root:"Americhernes")
build()
swap()
select()
calculate_support(jackknife:(remove:0.50,resample:1000),
build(5),swap(tbr,trees:3))
report("jackknives.pdf",graphsupports)
exit()
```

- `(*Jackknife support for static homology trees*)` This first line of the script is a comment. While comments are optional and do not affect the analyses, they are useful for housekeeping purposes.

- `read(prealigned:("28s.aln",tcm:(1,2)))` This command imports the prealigned nucleotide sequence file `28s.aln`, and treats the characters as static with the prescribed transformation costs, such that substitutions are assigned a cost of 1 and indels a cost of 2.

- `set(root:"Americhernes")` The `set` command specifies conditions prior to tree searching. The outgroup (`Americhernes`) is designated as the `root`, so that all the reported trees have the desired polarity.
- `build()` This command begins the tree-building step of the search that generates by default 10 random-addition Wagner trees. It is essential that trees are either specified from a file or that trees are built and loaded in memory before attempting to calculate support values.
- `swap()` The `swap` command specifies that each of the trees be subjected to an alternating SPR and TBR branch swapping routine (the default of `POY5`).
- `select()` Upon completion of branch swapping, this command retains only optimal and topologically unique trees; all other trees are discarded from memory.
- `calculate_support(jackknife,(remove:0.50,resample:1000), build(5),swap(tbr,trees:3))` The `calculate_support` command generates support values as specified by the `jackknife` argument for each tree held in memory. During each pseudoreplicate half of the characters will be deleted as specified in the argument `remove:0.50`. A search consisted of 5 Wagner tree builds (by random addition sequence) and swapping these trees under `tbr`, keeping three minimum-cost trees after each round, follows. This procedure is repeated 1000 times.
- `report("jackknives.pdf",graphsupports)` The `report` command in combination with a file name and the `graphsupports` generates a pdf file with jackknife values designated by the name specified (i.e. `jackknives.pdf`).
- `exit()` This command ends the `POY5` session.

5.6 Calculating supports: Bootstrap

This tutorial illustrates the calculation of Bootstrap support values for trees constructed from static homology characters. As these characters are not pre-aligned, the dynamic homology characters are converted to static characters using the argument `static_approx` prior to calculation of support.

```
(*Bootstrap support for static homology trees*)
```

```
read("28s.fas")
transform(tcm:(1,2))
set(root:"Americhernes")
build()
swap()
select()
transform(all,(static_approx))
swap()
calculate_support(bootstrap:100,build(5),swap(tbr,trees:5))
report("bootstraps.pdf",graphsupports)
exit()
```

- `(*Bootstrap support for static homology trees*)` This first line of the script is a comment. While comments are optional and do not affect the analyses, they are useful for housekeeping purposes.

- `read("28s.fas")` This command imports the nucleotide sequence file `28s.fas`.

- `transform(tcm:(1,2))` The file `28s.fas` is transformed such that the cost of substitutions are 1 and that of indels 2. The transformation cost matrix is used in calculating the cost of the tree for these data.

- `set(root:"Americhernes")` The `set` command specifies conditions prior to tree searching. The outgroup (`Americhernes`) is designated as the `root`, so that all the reported trees have the desired polarity.

- `build()` This command begins the tree-building step of the search that generates by default 10 random-addition Wagner trees. It is essential that trees are either specified from a file or that trees are built and loaded in memory before attempting to calculate support values.

- `swap()` The `swap` command specifies that each of the trees be subjected to an alternating SPR and TBR branch swapping routine (the default of `POY5`).

- `select()` Upon completion of branch swapping, this command retains only optimal and topologically unique trees; all other trees are discarded from memory.

- `transform(all,(static_approx))` This command transforms all the dynamic characters into static characters.

- `swap()` The local optimum for dynamic homology characters can differ from that for the static homology characters based on the same sequence data. Therefore, an extra round of swapping on the transformed data is performed in order to reach the local maximum for the static homology characters prior to calculating support values.

- `calculate_support(bootstrap:100,build(5),swap(tbr,trees:5))` The `calculate_support` command generates support values as specified by the `bootstrap` argument for each tree held in memory. During each pseudoreplicate the characters are randomly sampled and replaced, followed by 5 Wagner tree builds (by random addition sequence) and swapping these trees under `tbr`, keeping five minimum-cost trees after each round. The procedure is repeated 100 times.

- `report("bootstraps.pdf",graphsupports)` The `report` command in combination with a file name and the `graphsupports` generates a pdf file with bootstrap values designated by the name specified (i.e. `bootstraps.pdf`).

- `exit()` This command ends the `POY5` session.

5.7 Sensitivity Analysis

This tutorial demonstrates how data (tree costs) for parameter sensitivity analysis are generated. Sensitivity analysis [58] is a method of exploring the effect of relative costs of substitutions (transitions and transversions) and indels (insertions and deletions), either with or without taking gap extension cost into account. The approach consists of multiple iterations of the same search strategy under different parameters (i.e. combinations of substitution and indel costs). Obviously, such analysis might become time consuming and certain methods are shown here how to achieve the results in reasonable time. This tutorial also shows the utility of the command `store` and how transformation cost matrixes are imported and used.

`POY5` does not comprehensively display the results of the sensitivity analysis or implements the methods to select a parameter set that produces the optimal cladogram, but the output of a `POY5` analysis (such as the one presented here) generates all the necessary data for these additional steps.

For the sake of simplicity, this script contains commands for generating the data under just two parameter sets. Using a larger number of parameter sets can easily be achieved by replicating the repeated parts of the script and substituting the names of input cost matrixes.

```
(*Sensitivity Analysis*)

read("9.fas")
set(root:"t1")
store("original_data")
transform(tcm:"s1t1.mat")
build(100)
swap(timeout:3600)
select()
report("9_11.tre",trees:(total),"9_11cs.tre",consensus,
"9_11cs.pdf",graphconsensus)
load("original_data")
transform(tcm:"s1t2.mat")
build(100)
swap(timeout:3600)
select()
report("9_12.tre",trees:(total),"9_12cs.tre",consensus,
"9_12cs.pdf",graphconsensus)
exit()
```

- `(*Sensitivity Analysis*)` This first line of the script is a comment. While comments are optional and do not affect the analyses, they are useful for housekeeping purposes.
- `read("9.fas")` This command imports the nucleotide sequence file `9.fas` (in FASTA format).
- `set(root:"t1")` The outgroup (`t1`) is designated as the `root`, so that all the reported trees have the desired polarity.
- `store("original_data")` This commands stores the current state of analysis in memory in a temporary file, `original_data`.
- `transform(tcm:"s1t1.mat")` This command applies a transformation cost matrix from the file `s1t1.mat` to the data file stored in the file `original_data` for subsequent tree searching. In this cost matrix both substitutions and indels are assigned a cost of 1.
- `build(100)` This commands begins the tree-building step of the search that generates 100 random-addition Wagner trees. A large number of independent starting points will help to ensure that a useful portion of tree space will be examined.

- `swap(timeout:3600) swap` specifies that each of the 100 trees generated in the previous step is subjected to alternating SPR and TBR branch swapping routine (the default of `POY5`). The argument `timeout` specifies that 3600 seconds are allocated for swapping and the search is going to be stopped after reaching this limit. Because sensitivity analysis consists of multiple independent searches, it can take significant time to complete each one of them. In this example, `timeout` is used to prevent the searches from running too long. Using `timeout` is optional and can obviously produce suboptimal results due to insufficient time allocated to searching. A reasonable timeout value can be experimentally obtained by the analysis under one cost regime and monitoring time it takes to complete the search (using the argument `timer` of the command `set`). The advantage of using `timeout` is saving time in cases where a local optimum is quickly reached.

- `select()` Upon completion of branch swapping, this command retains only optimal and topologically unique trees; all other trees are discarded from memory.

- `report("9_11.tre",trees:(total),"9_11con.tre",consensus, "9_11con.pdf",graphconsensus)` This command produces a file containing the best tree(s) in parenthetical notation and their costs (`9_11.tre`), a file containing the strict consensus in parenthetical notation (`9_11con.tre`), and a graphical representation (in PDF format) of the strict consensus (`9_11con.pdf`).

- `load("original_data")` This command restores the original (non-transformed) data from the temporary file `original_data`, which was previously generated by `store`.

- `transform(tcm:"s1t2.mat")` This command applies a different transformation cost matrix in the file `s1t2.mat` to the data stored in the file `original_data` for another round of tree searching under this new cost regime.

- `build(100)` This commands begins the tree-building step of the search that generates 100 random-addition trees. A large number of independent starting point ensures that a large portion of tree space will be examined.

- `swap(timeout:3600) swap` specifies that each of the 100 trees generated in the previous step is subjected to alternating SPR and TBR branch

swapping routine (the default of `POY5`) to be interrupted after 3600 seconds (see the description in the previous iteration of the command above).

- `select()` Upon completion of branch swapping, this command retains only optimal and topologically unique trees; all other trees are discarded from memory.

- `report("9_12.tre",trees:(total),"9_12con.tre",consensus, "9_12con.pdf",graphconsensus)` This command produces a set of the same kinds of outputs as generated during the first search (see above) but under a new cost regime.

- `exit()` This command ends the `POY5` session.

5.8 Chromosome Analysis: Unannotated Sequences

This tutorial illustrates the analysis of chromosome-level transformations using unannotated sequences, i.e., contiguous strings of sequences without prior identification of independent regions.

```
(*Chromosome analysis of unannotated sequences*)

read(chromosome:("11mito.fas"))
transform(tcm:(1,2),gap_opening:3)
transform(chromosome:(annotate:(mauve,25.0,0.30,0.01,0.08)))
transform(fixed_states:("mauveout",ignore_polymorphism))
transform(chromosome:(locus_inversion:100,locus_indel:(10,0.9)))
build(100)
swap(threshold:5.0)
select()
set(root:"Taxon11")
report("unann_chrom_diag.txt",diagnosis)
report("unann_chrom_cs.pdf",graphconsensus)
exit()
```

- `(*Chromosome analysis of unannotated sequences*)` This first line of the script is a comment. While comments are optional and do not affect the analyses, they are useful for housekeeping purposes.

- `read(chromosome:("11mito.fas"))` This command imports the file `11mito.fas`, which consists of chromosomal sequences. The argument `chromosome` specifies that the characters are unannotated chromosomes.

- `transform(tcm:(1,2),gap_opening:3)` The file `11mito.fas` is transformed such that the cost of substitutions are 1, indels 2, and there is a gap opening cost of 3.

- `transform(chromosome:(annotate:(mauve,25.0,0.30,0.01,0.08)))` The argument `annotate:(mauve)` specifies that the program will use the Mauve aligner [10] to determine locally collinear homologous blocks (LCB) within the unannotated chromosomal sequences (`chromosome`). The float parameters that follow `mauve` are order dependent, and set the parameters for determining the LCB homologies: quality, coverage, minimum and maximum LCB length relative to overall sequence length. In this case, the LCB quality parameter, which represents the cost of the LCB divided by its length, is set to the relatively low value of 25 to facilitate the detection of blocks within the sequences. Higher LCB quality values will result in more stringent LCB determination and most likely, fewer local collinear blocks recovered. The second parameter within the argument `annotate:(mauve)` sets the minimum LCB sequence coverage at 30% meaning that if the total length of an input sequence is, for example, 100, a minimum coverage of 0.30 would require the total length of all LCBs be at least 30. The default value of 0.01 or 1% sets the minimum length of a given LCB relative to the length of the entire sequence (e.g. 100 for a 10,000 nucleotide sequence). The maximum length allowed for an LCB, in this example 8%, sets the maximum length of a given LCB relative to the length of the total sequence.

- `transform(fixed_states:("mauveout",ignore_polymorphism))` The `transform` command in combination with `fixed_states` is used to produce alignment files (`mauveout`), which when read into Mauve [10], can track the movement of LCBs between sequences. Here, Mauve alignment files will be generated with the names "mauve_i_j.alignment" where i and j are median states. Sequence ambiguities will not be resolved to generate additional medians beyond those determined by the data (`ignore_polymorphism`). These files can be used in conjunction with the `diagnosis` output to determine inferred rearrangement events. In the analysis of unannotated chromosomes, the data **must** be transformed to `fixed_states` when using the Mauve aligner.

- `transform(chromosome:(locus_inversion:100,locus_indel:(10,0.9)))` The `transform` command in combination with the argument `chromosome` signifies the conditions to be applied when calculating chromosome-level HTUs (medians). The argument `locus_inversion` applies an inversion distance between chromosome loci with the integer value (`100`) determining the rearrangement cost. The argument `locus_indel` specifies the indel costs for the chromosomal segments, such that the integer 10 sets the gap opening cost and the float 0.9 sets the gap extension cost. When selecting appropriate cost parameters for transformation events it is important to remember that the lowest cost option for an event will be applied. For example, in the sample mitochondrial data set used in this tutorial it is biologically feasible that locus level transformations may have occurred in short (<100) nucleotide strings (e.g. tRNA genes). To allow for locus transformations to be detected in these data an appropriate locus indel cost must be less than the relative cost of explaining these transformations by nucleotide indels and substitutions.

- `build(100)` This command begins the tree-building step of the search that generates 100 random-addition Wagner trees. It is highly recommended that a greater number of Wagner builds be implemented when analyzing data for purposes other than this demonstration.

- `swap(threshold:5.0)` The `swap` command specifies that each of the trees is subjected to an alternating SPR and TBR branch swapping routine (the default of `POY5`). In addition to the optimal trees, all suboptimal trees found within 5% of the best cost, are thoroughly evaluated.

- `select()` Upon completion of branch swapping, this command retains only optimal and topologically unique trees; all other trees are discarded from memory.

- `set(root:"Taxon11")` The outgroup (`Taxon11`) is designated by the `root`, so that all the reported trees have the desired polarity.

- `report("unann_chrom_diag.txt",diagnosis)` The `report` command, in combination with a file name and the argument `diagnosis`, outputs the optimal median states and edge values to the specified file (`unann_chrom_diag.txt`).

- `report("unann_chrom_cs.pdf",graphconsensus)` The `report` command, in combination with a file name and the argument `graphconsensus`, generates a strict consensus file (in PDF format) of the trees generated and selected in the analysis (`unann_chrom_cs.pdf`).
- `exit()` This command ends the `POY5` session.

5.9 Chromosome Analysis: Annotated Sequences

This tutorial illustrates the analysis of chromosome-level transformations using annotated sequences, i.e., contiguous strings of sequences with prior identification of independent regions delineated by pipes `"|"`.

```
(*Chromosome analysis of annotated sequences*)

read(annotated:("aninv2.fas"))
transform(annotated:(locus_inversion:20,median_solver:caprara,
locus_indel:(10,1.5),circular:false,median:1,swap_med:1))
build(20)
swap()
select()
report("ann_chrom_diag.txt",diagnosis)
report("ann_chrom_diag.pdf",graphdiagnosis)
report("ann_chrom_cs.pdf",graphconsensus)
exit()
```

- `(*Chromosome analysis of annotated sequences*)` This first line of the script is a comment. While comments are optional and do not affect the analyses, they are useful for housekeeping purposes.
- `read(annotated:("aninv2.fas"))` This command imports the annotated chromosomal sequence file `aninv2`. The argument `annotated` specifies that the characters are annotated chromosomes (not to be confused with `annotate`).
- `transform(annotated:(locus_inversion:20,median_solver:caprara,locus_indel:10,1.5),circular:false,median:1,swap_med:1)))` The `transform` followed by the argument `annotated` specifies the conditions to be applied when calculating chromosome-level HTUs (medians). The argument `locus_inversion` applies an inversion distance between chromosome loci with the integer value of 50

determining the rearrangement cost while using the default `caprara` median solver. The argument `locus_indel` specifies the indel costs for chromosomal segments, where the integer 10 sets the gap opening cost and the float 1.5 sets the gap extension cost. The default values are applied to the `circular`, `median` and `swap_med` arguments to minimize the time require for these nested search options. To more exhaustively perform these calculations, trees generated from initial builds can be imported to the program and reevaluated with values greater than 1 entered for the median and swap med arguments.

- `build(20)` This commands begins the tree-building step of the search, generating 20 Wagner trees by random-addition sequence. It is highly recommended that a greater number of Wagner builds be implemented when analyzing data for purposes other than this demonstration.

- `swap()` The `swap` command specifies that each of the trees be subjected to an alternating SPR and TBR branch swapping routine (the default of `POY5`).

- `select()` Upon completion of branch swapping, this command retains only optimal and topologically unique trees; all other trees are discarded from memory.

- `report("ann_chrom_diag.txt",diagnosis)` The `report` command, in combination with a file name and the argument `diagnosis`, outputs the optimal median states and edge values to the file specified (`ann_-chrom_diag.txt`).

- `report("ann_chrom_diag.pdf",graphdiagnosis)` The `report` command, in combination with a file name and the argument `graphdiagnosis`, outputs a file (in PDF format) with labeled medians that allow users to link to the diagnosis file to reconstruct the median states at the internal tree nodes.

- `report("ann_chrom_cs.pdf",graphconsensus)` The `report` command, in combination with a file name and the argument `graphconsensus`, generates a strict consensus file (in PDF format) of the trees generated and selected in the analysis.

- `exit()` This command ends the `POY5` session.

5.10 Chromosome Analysis: Unannotated and annotated

This tutorial illustrates the analysis of chromosome-level transformation using both unannotated and annotated sequences. Though similar to the previous two tutorials, this tutorial differs in that the identifier `names` must be utilized when transforming the unannotated chromosomes.

```
(*Chromosome analysis: Unannotated and annotated sequences*)

read(annotated:("aninv2.fas"),chromosome:("11mito.fas"))
transform(annotated:(locus_inversion:20,median_solver:caprara,
locus_indel:(10,1.5),circular:false,median:1,swap_med:1))
transform((names:("11mito.fas"),(chromosome:(annotate:
(mauve,25.0,0.30,0.01,0.08)))))
transform((names:("11mito.fas"),(fixed_states:("mauveout",
ignore_polymorphism))))
transform(chromosome:(locus_inversion:100,locus_indel:(10,0.9)))
transform(tcm:(1,2),gap_opening:3)
build(20)
swap()
select()
report("annunann_chrom_diag.txt",diagnosis)
report("annunann_chrom_diag.pdf",graphdiagnosis)
report("annunann_chrom_cs.pdf",graphconsensus)
exit()
```

- `(*Chromosome analysis: Unannotated and annotated sequence*)` This first line of the script is a comment. While comments are optional and do not affect the analyses, they are useful for housekeeping purposes.

- `read(annotated:("aninv2.fas"),chromosome:("11mito.fas"))` This command imports the sequence file `aninv2.fas` containing annotated chromosomal sequences (contiguous strings of sequences with prior identification of independent regions delineated by pipes `"|"`) and the unannotated chromosomal sequence file `11mito.fas`.

- `transform(annotated:(locus_inversion:20,median_solver:caprara,locus_indel:(10,1.5),circular:false,median:1,swap_med:1))` The `transform` command followed by the argument `annotated`

specifies the conditions to be applied when calculating chromosome-level HTUs (medians). The argument `locus_inversion` applies an inversion distance between chromosome loci with the integer value of 50 determining the rearrangement cost while using the default `caprara` median solver. The argument `locus_indel` specifies the indel costs for chromosomal segments, where the integer 10 sets the gap opening cost and the float 1.5 sets the gap extension cost. The default values are applied to the `circular`, `median` and `swap_med` arguments to minimize the time require for these nested search options. To more exhaustively perform these calculations, trees generated from initial builds can be imported to the program and reevaluated with values greater than 1 entered for the median and swap med arguments.

- `transform((names:("11mito.fas"),(chromosome:(annotate:(mauve,25.0,0.30,0.01,0.08)))))` Contrary to the transformation of unannotated chromosomes in `tutorial 5.8`, in this script it is necessary to identify, using the *identifier* `names`, which file the Mauve aligner will be applied to. Without this *identifier*, the script will not run. Once identified, the argument `annotate:(mauve)` specifies that the program will use the Mauve aligner [10] to determine locally collinear homologous blocks (LCB) within the unannotated chromosomal sequences (`chromosome`). The float parameters that follow `mauve` are order dependent, and set the parameters for determining the LCB homologies: quality, coverage, minimum and maximum LCB length relative to overall sequence length. In this case, the LCB quality parameter, which represents the cost of the LCB divided by its length, is set to the relatively low value of 25 to facilitate the detection of blocks within the sequences. Higher LCB quality values will result in more stringent LCB determination and most likely, fewer local collinear blocks recovered. The second parameter within the argument `annotate:(mauve)` sets the minimum LCB sequence coverage at 30% meaning that if the total length of an input sequence is, for example, 100, a minimum coverage of 0.30 would require the total length of all LCBs be at least 30. The default value of 0.01 or 1% sets the minimum length of a given LCB relative to the length of the entire sequence (e.g. 100 for a 10,000 nucleotide sequence). The maximum length allowed for an LCB, in this example 8%, sets the maximum length of a given LCB relative to the length of the total sequence.

- `transform((names:("11mito.fas"),(fixed_states:("mauveout",`

`ignore_polymorphism))))` The `transform` command in combination with `fixed_states` is used to produce alignment files (`mauveout`), which when read into Mauve [10], can track the movement of LCBs between sequences. As in the previous line of the script, the unannotated chromosome file to which `fixed_states` will be applied, is identified using the *identifier* `names`. Here, Mauve alignment files will be generated with the names "mauveout_i_j.alignment" where i and j are median states. Sequence ambiguities will not be resolved to generate additional medians beyond those determined by the data (`ignore_polymorphism`). These files can be used in conjunction with the `diagnosis` output to determine inferred rearrangement events. In the analysis of unannotated chromosomes, the data **must** be transformed to `fixed_states` when using the Mauve aligner.

- `transform(chromosome:(locus_inversion:100,locus_indel:(10,0.9)))` The `transform` command in combination with the argument `chromosome` signifies the conditions to be applied when calculating chromosome-level HTUs (medians). The argument `locus_inversion` applies an inversion distance between chromosome loci with the integer value (`100`) determining the rearrangement cost. The argument `locus_indel` specifies the indel costs for the chromosomal segments, such that the integer 10 sets the gap opening cost and the float 0.9 sets the gap extension cost. When selecting appropriate cost parameters for transformation events it is important to remember that the lowest cost option for an event will be applied. This transformation will be applied to both data files.

- `transform(tcm:(1,2),gap_opening:3)` Both files are transformed such that the cost of substitutions are 1, indels 2, and there is a gap opening cost of 3.

- `build(20)` This commands begins the tree-building step of the search, generating 20 Wagner trees by random-addition sequence. It is highly recommended that a greater number of Wagner builds be implemented when analyzing data for purposes other than this demonstration.

- `swap()` The `swap` command specifies that each of the trees be subjected to an alternating SPR and TBR branch swapping routine (the default of `POY5`).

- `select()` Upon completion of branch swapping, this command retains only optimal and topologically unique trees; all other trees are discarded

from memory.

- `report("annunann_chrom_diag.txt",diagnosis)` The `report` command, in combination with a file name and the argument `diagnosis`, outputs the optimal median states and edge values to the file specified (`annunann_chrom_diag.txt`).
- `report("annunann_chrom_diag.pdf",graphdiagnosis)` The `report` command, in combination with a file name and `graphdiagnosis`, outputs a file (in PDF format) with labeled medians that allow users to link to the diagnosis file to reconstruct the median states at the internal tree nodes.
- `report("annunann_chrom_cs.pdf",graphconsensus)` The `report` command, in combination with a file name and the argument `graphconsensus`, generates a strict consensus file (in PDF format) of the trees generated and selected in the analysis.
- `exit()` This command ends the `POY5` session.

5.11 Genome Analysis: Multiple Chromosomes

This tutorial illustrates the analysis of genome-level transformations using data from multiple chromosomes. Genome data consists of multi-locus, multi-chromosomal nucleotide sequences, wherein transformations (i.e. indels, substitutions, and rearrangements) are optimized at the sequence, locus and chromosomal level. Within the genome data file, individual chromosomes are separated by the "@" symbol and the individual chromosomes remain unannotated.

```
(*Genome Analysis: Multiple Chromosomes*)

read(genome:("gen7.fas"))
transform(tcm:(1,1),gap_opening:1)
transform(chromosome:(annotate:(mauve,25.0,0.30,0.01,0.08)))
transform(fixed_states:("genomeout",ignore_polymorphism))
transform(chromosome:(locus_breakpoint:80,locus_indel:(15,2.5),
median_solver:caprara))
transform(genome:(translocation:100,chrom_indel:(10,0.9)))
build(100)
swap()
```

```
select()
set(root:"taxon5")
report("genome_diag.txt",diagnosis)
report("gen_diag.pdf",graphdiagnosis,"gen_cs.pdf",graphconsensus)
exit()
```

- `(*Genome Analysis: Multiple Chromosomes*)` This first line of the script is a comment. While comments are optional and do not affect the analyses, they are useful for housekeeping purposes.

- `read(genome:("gen7.fas"))` This command imports the genomic sequence file `gen7.fas`. The argument `genome` specifies that the characters consist of multi-chromosomes.

- `transform(tcm:(1,1),gap_opening:1))` The file `gen7.fas` is transformed such that during the subsequent analysis of this data file, the cost of substitutions, indels and gap opening cost are all set to 1.

- `transform(chromosome:(annotate:(mauve,25.0,0.30,0.01,0.08)))` The argument `annotate:(mauve)` specifies that the program will use the Mauve aligner [10] to determine locally collinear homologous blocks (LCB) within the chromosomal sequences. The float parameters that follow the `mauve` option, are order dependent and set the parameters for determining the LCB homologies: quality, coverage, minimum and maximum LCB length relative to overall sequence length. In this case, the LCB quality parameter, which represents the cost of the LCB divided by its length, is set to the relatively low value of 25 to facilitate the detection of blocks within the sequences. Higher LCB quality values will result in more stringent LCB determination and most likely, fewer local collinear blocks recovered. The second parameter within the argument `annotate:(mauve)` sets the minimum LCB sequence coverage at 30% meaning that if total length of an input sequence is, for example, 100, a minimum coverage of 0.30 would require the total length of all LCBs to be at least 30. The default value of 0.01 or 1% sets the minimum length of a given LCB relative to the length of the entire sequence (e.g. 100 for a 10,000 nucleotide sequence). The maximum length allowed for an LCB, in this example 8%, sets the maximum length of a given LCB relative to the length of the total sequence.

- `transform(fixed_states:("genomeout",ignore_polymorphism))` The `transform` command in combination with `fixed_states` is used

to produce alignment files (`"genomeout"`) that can be read into Mauve to track the movement of LCBs between sequences. Here, Mauve genome alignment files will be generated with the names "genomeout_-i_j.alignment" where i and j are median states. Sequence ambiguities will not be resolved to generate additional medians beyond those determined by the data (`ignore_polymorphism`). These files can be used in conjunction with the `diagnosis` output to determine inferred translocation and rearrangement events. In the analysis of unannotated chromosomes, the data **must** be transformed to `fixed_states` when using these Mauve aligners.

- `transform(chromosome:(locus_breakpoint:80,locus_indel:(15,2.5),median_solver:caprara))` The command `transform` followed by `chromosome` specifies the conditions to be applied when calculating genome-level HTUs HTUs (medians). `locus_breakpoint` applies a breakpoint distance between chromosomes with the integer value determining the rearrangement cost. `locus_indel` specifies the indel costs for chromosomal segments, where the integer 15 setting the gap opening cost and the float 2.5 sets the gap extension cost. The median solver `caprara` will be employed in the determination of rearrangement costs.

- `transform(genome:(translocation:100,chrom_indel:(10,0.9)))` The argument `translocation` sets the breakpoint cost for the movement of LCBs from one chromosomal segment to another. The argument `chrom_indel` specifies the indel costs for each entire chromosome, whereby the integer sets the gap opening cost and the float sets the gap extension cost.

- `build(100)` This commands begins the tree-building step of the search that generates by default 100 Wagner trees (by random-addition sequence). It is highly recommended that a greater number of Wagner builds be implemented when analyzing data for purposes other than this demonstration.

- `swap()` The `swap` command specifies that each of the trees be subjected to an alternating SPR and TBR branch swapping routine (the default of `POY5`).

- `select()` Upon completion of branch swapping, this command retains only optimal and topologically unique trees; all other trees are discarded from memory.

- `set(root:"taxon5")` The `set` command specifies the outgroup taxon (`taxon5`) is designated as the `root`, so that all the reported trees have the desired polarity.

- `report("genome_diag.txt",diagnosis)` The `report` command, in combination with a file name and `diagnosis`, outputs the optimal median states and edge values to a specified file (`genome_diag.txt`).

- `report("gen_diag.pdf",graphdiagnosis,"gen_cs.pdf",graphconsensus)` The `report` command, in combination with a file name and `graphdiagnosis`, outputs a pdf tree file with labeled medians that allow users to link to the diagnosis file to reconstruct the median states at the internal tree nodes. A strict consensus of the trees generated (`gen_cs.pdf`) is also reported.

- `exit()` This command ends the `POY5` session.

5.12 Custom Alphabet Character Analysis

This tutorial illustrates the analysis of the custom alphabet character type. Custom Alphabet characters are those that employ a user-specified alphabet. With this data type, only insertion-deletion and substitution events are allowed.

```
(*Custom Alphabet Character Analysis*)

read(custom_alphabet:("ca1.fas",tcm:("m1.mat")))
transform(level:3)
build(all,10)
swap()
fuse(iterations:5,replace:best,keep:5,swap())
select()
set(root:"One")
report("CA1_trees.tre",trees:(total),"CA1_cs.pdf",graphconsensus)
quit()
```

- `(*Custom Alphabet Character Analysis*)` This first line of the script is a comment. While comments are optional and do not affect the analyses, they are useful for housekeeping purposes.

- `read(custom_alphabet:("ca1.fas",tcm:("m1.mat")))`
  This command imports the user-defined `custom_alphabet` character file `ca1.fas` and the accompanying transformation matrix `m1.mat`.

- `transform(level:3)` This command specifies the heuristic `level` of the median sequence calculation.

- `build(all,10)` This command builds ten trees and turns off all preference strategies for adding branches and tries all possible addition positions for all terminals.

- `swap()` Submits current trees to a round of SPR followed by TBR, the default settings.

- `fuse(iterations:10,replace:best,keep:5,swap())` In this step, up to 10 swaps of subtrees identical in terminal composition but different in topology, are performed between pairs of best trees recovered in the previous step. The cost of the resulting trees is compared to that of the trees in memory and a subset of the trees containing up to 5 trees of best cost are retained in memory. These trees are subjected to swapping under the default settings of swap. This procedure is repeated nine more times.

- `select()` Upon completion of branch swapping, this command retains only optimal and topologically unique trees; all other trees are discarded from memory.

- `set(root:"One")` The `set` command specifies the outgroup taxon (`One`) is designated as the `root`, so that all the reported trees have the desired polarity.

- `report("CA1_trees.tre",trees:(total),"CA1_cs.pdf", graphconsensus)` This command reports a series of outputs of the results of the search. It includes a file containing the optimal trees in parenthetical notation (`CA1_trees.tre`) with the associated costs (`trees:(total)`). In addition, a strict consensus tree (in PDF format) of all the most optimal trees, will be reported `CA1_cs.pdf`.

- `quit()` This command ends the `POY5` session.

5.13 Break Inversion Character Analysis

This tutorial illustrates the analysis of the break inversion character type. Break inversion characters are an extension of user-defined `custom_alphabet` characters, with insertion-deletion, substitutions **and** rearrangement events being considered among alphabet elements.

```
(*Break Inversion Character Analysis*)

read(breakinv:("ca2.fas",tcm:("m1.mat"),orientation:true))
transform(breakinv:(locus_inversion:20,median_solver:
siepel,median:1,swap_med:1))
build(20)
swap()
select()
report("BInv_diag.txt",diagnosis)
report("BI_trees.tre",trees,"BI_cs.pdf",graphconsensus)
exit()
```

- `(*Break Inversion Character Analysis *)` This first line of the script is a comment. While comments are optional and do not affect the analyses, they are useful for housekeeping purposes.

- `read(breakinv:("ca2.fas",tcm:("m1.mat"),orientation:true))` This command imports the user-defined `breakinv` character file `ca2.fas` and the accompanying transformation cost matrix `m1.mat`. The user will note that tildes precede some of the characters in the data file. These tildes, which indicate negative orientation in a Break Inversion analysis. The argument `orientation` for these character types is set as `true`, such that the tilde ("∼") that precede some of the characters in the data file `ca2.fas` indicates negative orientation. [Note it is not possible to change this to false.]

- `transform(breakinv:(locus_inversion:10,median_solver:siepel, locus_inversion:20,median:1,swap_med:1))` The `transform` followed by the argument `breakinv` specifies the conditions to be applied when calculating medians. The argument `median _solver:siepel` specifies that the Siepel median from the GRAPPA software package [2] will be employed. The argument `locus_inversion` applies an inversion rearrangement cost of 20 for `breakinv` elements. The default values are applied to the `median` and `swap_med` arguments to minimize the time

require for these nested search options. To more exhaustively perform these calculations trees generated from initial builds can be imported to the program and reevaluated with values greater than 1 designated for the `median` and `swap_med` arguments.

- `build(20)` This commands begins the tree-building step of the search that generates 20 Wagner trees by random-addition sequence. It is highly recommended that a greater number of Wagner builds be executed when analyzing data for purposes other than this demonstration.
- `swap()` The `swap` command specifies that each of the trees be subjected to an alternating SPR and TBR branch swapping routine (the default of `POY5`).
- `select()` Upon completion of branch swapping, this command retains only optimal and topologically unique trees; all other trees are discarded from memory.
- `report("BInv_diag.txt",diagnosis)` The `report` command in combination with a file name and the `diagnosis` outputs the optimal median states and edge values to the specified file (`BInv_diag.txt`).
- `exit()` This commands ends the `POY5` session.

5.14 Maximum Likelihood Analysis: Static

This tutorial illustrates the analysis of static characters under the maximum average likelihood (MAL) criterion. This analysis is of similar intensity to that of a search using the *GTR* model in `PhyML`. Full maximum likelihood analyses, (i.e. analyses that include builds under likelihood, *sensu* PAUP*) can be computationally intensive, therefore parsimony alternatives to RAS under likelihood are provided.

```
(*Static ML analysis: Initial parsimony search*)

read(prealigned:("9.fas",tcm:(1,1)))
search(max_time:00:00:20)
select()

(*Transform static to LK characters. Heuristics follow*)
```

```
set(opt:coarse)
transform(likelihood:(gtr,rates:gamma:(4),priors:estimate,
gap:missing,mal))
swap(all:5,spr,optimize:(model:never,branch:never))
fuse(optimize:(model:never,branch:join_region))
select(best:1)
set(opt:exhaustive)
report("9_MAL.tre",trees:(branches),"9_MAL_lkm.txt",lkmodel)
quit()
```

- `(*Static ML analysis: Initial parsimony search*)` This first line of the script is a comment. While comments are optional and do not affect the analyses, they are useful for housekeeping purposes.
- `read(prealigned:("9.fas",tcm:(1,1)))` This command imports the nucleotide sequence data file `9.fas` as prealigned characters and specifies the transformation cost matrix to be used in calculating the cost of the tree for these data, such that the cost of substitutions and indels are 1.
- `search(max_time:00:00:20) search` is a default strategy that will perform as many builds, swaps, perturbation using ratchet, and tree fusing for the defined time of 20 minutes. [Note: 20 minutes was determined to be an adequate amount of time to before this search, however, in most cases a timed search of 20 minutes would not be enough time for an in-depth search.]
- `select()` This command retains only optimal and topologically unique trees; all other trees are discarded from memory.
- `(*Transform static to LK characters. Heuristics follow*)`
- `set(opt:coarse)` This command sets the floating point optimization strategy for subsequent swapping under likelihood. In this case, the tolerance of the routines is set to 1e-3 (half the log of a full, exhaustive search).
- `transform(likelihood:(gtr,rates:gamma:(4),priors:estimate, gap:missing,mal))` The command `transform`, followed by `likelihood`, specifies the conditions to be applied when transforming these static parsimony characters to static likelihood characters. A $GTR + \Gamma 4$ model (`gtr,rates:gamma:(4)`), with empirical equilibrium frequencies (`priors:estimate`) under static `mal` will be employed. Within this transformation, indels are treated as missing (`gap:missing`).

- `swap(all:5,spr,optimize:(model:never,branch:never))` This command swaps using `spr` within a distance of 5 branches from the join point. Following each round of SPR, neither the model nor the branches are optimized during the swap process (`optimize:(model:never, branch:never)`).

- `fuse(optimize:(model:never,branch:join_region))` This command performs tree fusing, specifying that the likelihood model is never optimized after each round of fusing (`optimize:(model:never)`), but that a maximum of five branches are optimized each round of fusing (`optimize:(branch:join_region)`).

- `select(best:1)` This command saves a single optimal topology (with branch lengths) in memory. All other trees are discarded from memory.

- `set(opt:exhaustive)` This command sets the floating point optimization strategy for subsequent swapping under likelihood. The tolerance level is set to 1e-6.

- `report("9_MAL.tre",trees:(branches),"9_MAL_lkm.txt", lkmodel)` This command reports a series of outputs from the analysis. It includes a file, in parenthetical notation, containing the optimal topological tree (`9_MAL.tre`), along with the branch lengths `trees:(branches)`), as well as a file containing the parameter estimates generated by dynamic MPL (`9_MAL_lkm.txt`). These estimates include the likelihood score, the variant of likelihood used, the tree length (sum of branch lengths), the values of the parameter estimates for the entries of the substitution rate matrix (**Q**), and the estimate of the value of the rate variation shape parameter.

- `exit()` This command ends the `POY5` session.

5.15 Maximum Likelihood Analysis: Dynamic

This tutorial illustrates the analysis of dynamic characters under the most parsimonious likelihood (MPL) criterion. The frequentist model-based simultaneous alignment and topology search is a largely unexplored area, and the heuristics may not be sufficiently. Therefore, it is likely that all but very simple dynamic MPL analyses will not be possible on basic computers, and that significant parallelization will need to be implemented to make larger datasets amenable to analysis under this criterion. For comparative purposes,

the user is encouraged to run alternative analyses, transforming the data using `elikelihood` and to explore the effect of different models on the results of the analysis.

```
(*Maximum likelihood analysis: Dynamic*)

read("9.fas")
search(max_time:00:00:20)
select()

(*Transform parsimony DO characters to dyn MPL characters*)

set(opt:coarse)
transform(likelihood:(gtr,rates:gamma:(4),priors:estimate,
gap:coupled,mpl))
swap(spr,all:5)
select(best:1)
set(opt:exhaustive)
report("9_dMPL.tre",trees:(branches),"9_dMPL_lkm.txt",lkmodel,
"9_dMPL.ia",ia)
transform(static_approx)
report("9_sMPL.tre",trees:(branches),"9_sMPL_lkm.txt",lkmodel)
exit()
```

- `(*Maximum likelihood analysis:Dynamic*)` This first line of the script is a comment. While comments are optional and do not affect the analyses, they are useful for separating different components of an analysis, especially if the script is long.

- `read("9.fas")` This command imports the nucleotide sequence data file `9.fas`. [Note: unlike the previous tutorial, the characters are not imported as prealigned.]

- `search(max_time:00:00:20)` `search` is a default search strategy that will perform as many builds, swaps, perturbation using ratchet, and tree fusing for the defined time of 20 minutes. [Note: 20 minutes was determined to be an adequate amount of time to before this search, however, in most cases a timed search of 20 minutes would not be enough time for an in-depth search.]

- `select()` Upon completion of branch swapping, this command retains

only optimal and topologically unique trees; all other trees are discarded from memory.

- `(*Transform parsimony DO characters to dyn MPL characters*)`

- `set(opt:coarse)` Sets coarse granularity for floating point optimization. In this case, the tolerance of the routines is set to 1e-3 (half the log of a full, exhaustive search).

- `transform(likelihood:(gtr,rates:gamma:(4),priors:estimate, gap:coupled,mpl))` The command `transform`, followed by `likelihood`, specifies the conditions to be applied when transforming these dynamic parsimony characters to dynamic likelihood characters. A `gtr` model, with empirical equilibrium frequencies (`priors:estimate`) under dynamic `mpl` will be employed. Within this transformation, atomic indels are a character state, with rates of nucleotide-to-indel substitution constrained to be equal to one another (`gap:coupled`). [Note: Under dynamic MPL rate variation distribution is not enabled.]

- `swap(spr,all:5)` This command swaps the tree using `spr`, specifying that joins occur within 5 branches from the break point. Iteration of likelihood model parameters occurs after every join.

- `select(best:1)` Upon completion of branch swapping, this command saves 1 of the most optimal (`best`) topological trees in memory. All other trees are discarded from memory.

- `set(opt:exhaustive)` Sets machine precision granularity for floating point optimization. Optimization is run over multiple iterations until convergence.

- `report("9_dMPL.tre",trees:(branches),"9_dMPL_lkm.txt", lkmodel,"9_dMPL.ia",ia)` This command reports a series of outputs of the results of the search. It includes a file containing the optimal topological tree (`9_dMPL.tre`), along with the branch lengths (`trees:(branches)`), as well as a file containing the parameter estimates generated (`9_dMPL_lkm.txt`). In addition, a file containing the implied alignment generated by dynamic MPL (9 dMPL.ia) is also generated.

- `transform(static_approx)` This command transforms all the dynamic characters into static characters.

- `report("9_sMPL.tre",trees:(branches),"9_sMPL_lkm.txt", lkmodel)` This command reports a series of outputs of the results of the search. It includes a file containing the optimal topological tree (`9_sMPL.tre`), along with the branch lengths (`trees:(branches)`), as well as a file containing the parameter estimates generated (`9_sMPL_-lkm.txt`). These estimates include the likelihood score, the variant of likelihood used, the tree length (sum of branch lengths), the values of the parameter estimates for the entries of the substitution rate matrix ($\mathbf{Q}$), and the estimate of the value of the rate variation shape parameter.

- `exit()` This commands ends the POY5 session.

5.16 ML Analysis: Partitions and Model Selection

The following script cover the analyses of model selection and partitioned analysis under the maximum likelihood criterion. The first section of this tutorial covers the selection of models for, and analysis of, partitioned codons of protein coding sequences.

```
(*ML Analysis: Partitions and Model Selection*)

read(prealigned:("coleoptera_nd2.fasta",tcm:(1,1)))
set(codon_partition:("coleop",names:("coleoptera_nd2.fasta")))
build(100)
swap()
select(best:1)
transform(likelihood:(aicc:"coleoptera_cp",rates:gamma:(4)))
swap(spr,all:5,optimize:(model:threshold:1.33,branch:join_delta)
report("codon_LK.tre",trees:(branches),"codon_LK_lkm.txt",lkmodel)
exit()
```

- `(*ML Analysis: Partitions and Model Selection*)` This first line of the script is a comment. While comments are optional and do not affect the analyses, they are useful for housekeeping purposes.

- `read(prealigned:("coleoptera_nd2.fasta",tcm:(1,1)))` This command imports the nucleotide sequence data file `coleoptera_nd2.fasta` as prealigned characters. The `tcm` sets the transformation cost matrix to be used in calculating the cost of the tree for these data (the cost of substitutions and indels are 1).

- `set(codon_partition:("coleop",names:("coleoptera_nd2.fasta")))` Specifies that the data be partitioned as *codon* data, wherein 3 partitions will be defined. Each partition will consists of every third nucleotide position. Three codon partitions named `coleop1`, `coleop2`, and `coleop3` will be created. The *identifier* `names` signals that the file `coleopteran_nd2.fasta` is that from which the partitioning will be applied.

- `build(100)` This command begins the tree-building step of the search by random-addition trees. 100 trees are built.

- `swap()` The `swap` command specifies that each of the trees be subjected to an alternating SPR and TBR branch swapping routine (the default of `POY5`).

- `select(best:1)` Upon completion of branch swapping, this command selects 1 of the most optimal (`best`) topological trees in memory. All other trees are discarded from memory.

- `transform(likelihood:(aicc:"coleoptera_cp",rates:gamma:(4)))` The command `transform`, followed by `likelihood`, specifies the conditions to be applied when transforming these static parsimony characters to static likelihood characters. This command runs model selection, including both the named-rate-matrix-only (RMO) models and RMO + `gamma:(4)` models, and using the corrected AIC (`aicc`) as the model selection criterion. The results with all model fits are output to the file `coleoptera_cp`, and the best-fit models for each codon position are automatically stored in memory for subsequent analysis.

- `swap(spr,all:5,optimize:(model:(threshold:1.33),branch:join_delta))` This command swaps the tree using `spr`, with joins occurring within five branches of the break site. The model parameters are optimized if the cost of the join under the current model is within 1.33 times the current best cost (proportion 0.33 worse). Only the branches along the path from the break to the new join location are optimized (`branch:join_delta`).

- `report("codon_LK.tre",trees:(branches),"codon_LK_lkm.txt",lkmodel)` This command reports a series of outputs of the results of the search. A file containing a tree, in parenthetical notation, containing the optimal topological tree (`codon_LK.tre`), along with the branch lengths `trees:(branches)` is reported. In addition, a file containing

the parameter estimates of the likelihood analysis, i.e. likelihood score, the variant of likelihood used, the tree length (sum of branch lengths), the values of the parameter estimates for the entries of the substitution rate matrix (**Q**), and the estimate of the value of the rate variation shape parameter, is also reported (`codon_LK_lkm.txt`).

- `exit()` This commands ends the `POY5` session.

5.17 Maximum Likelihood Analysis: Morphology

This tutorial illustrates the analysis of morphological data under the maximum likelihood criterion. In this analysis, qualitative characters are transformed to likelihood using different models. In addition, different alphabet sizes are applied to different ranges of characters within the same dataset.

```
(*Maximum Likelihood Analysis: Morphology*)

read("morpho.ss")
transform(likelihood:(ncm))
search()
report("Morpho_ncm.tre",trees:(total))
transform(range:("morpho.ss",0,99),(likelihood:(jc69,
alphabet:min)))
transform(range:("morpho.ss",100,173),(likelihood:(jc69,
alphabet:5)))
report("Morpho_jc69.tre",trees:(total,branches))
exit()
```

- `(*Maximum Likelihood Analysis: Morphology*)` This first line of the script is a comment. While comments are optional and do not affect the analyses, they are useful for housekeeping purposes.
- `read("morpho.ss")` This command imports the Hennig86 morphological data file `morpho.ss`.
- `transform(likelihood:(ncm))` The `transform` command followed by `likelihood`, specifies the conditions to be applied when transforming these morphological characters to likelihood characters. A `ncm` model, which is an extension of the Neyman model, will be employed. With this model, each character is free to evolve at its own rate on every edge of the tree. Because these characters evolve at their own rate, gamma options are ignored.

- `search()` The `search` command followed by empty parentheses will perform a timed search under the default parameters, i.e. for at most one hour using at most 2 GB of memory.

- `report("Morpho_ncm.tre",trees:(total))` This command reports a tree file, in parenthetical notation, containing the most optimal topological tree (`Morpho_ncm.tre`) under the `ncm` model, with the associated costs in square brackets (`trees:(total)`).

- `transform(range:("morpho.ss",0,99),(likelihood:(jc69,alphabet:min)))` The `transform` command followed by the `likelihood` argument transforms the first 100 characters of the morphological data file `morpho.ss` to a Neyman model (`jc69`) with the alphabet size being the observed number of states in the dataset.

- `transform(range:("morpho.ss",100,173),(likelihood:(jc69, alphabet:5)))` The `transform` command followed by the `likelihood` argument transforms the next 74 characters of the morphological data file `morpho.ss` to a Neyman model (`jc69`) with the alphabet size being 5, which could include unobserved states in the dataset.

- `report("Morpho_jc69.tre",trees:(total,branches))` This command reports a tree file, in parenthetical notation, containing the optimal topological tree (`Morpho_jc69.tre`) under the `jc69` model, along with the branch lengths `trees:(branches)`.

- `exit()` This command ends the `POY5` session.

5.18 ML Analysis: Morphology and Molecular

This tutorial illustrates the analysis of both morphological and molecular data under the maximum likelihood criterion. Within this analysis an `ncm` model is applied to 'static' data, while a `tn93` is applied to unaligned data.

```
 (*Maximum Likelihood Analysis: Combined Data*)

read("morpho.ss")
read(prealigned:("28s.aln"),tcm:(1,1))
transform(likelihood:(ncm))
read("18s.fas")
transform((names:("18s.fas"),(likelihood:(tn93,rates:none,
```

```
priors:estimate,gap:coupled,mpl))))
search(max_time:00:01:00)
swap(all,optimize:(model:never,branch:join_region))
report("All_LK.tre",trees:(branches),"All_lkm.txt",lkmodel)
exit()
```

- `(*Maximum Likelihood Analysis: Combined Data*)` This first line of the script is a comment. While comments are optional and do not affect the analyses, they are useful for housekeeping purposes.
- `read("morpho.ss")` This command imports the morphological data file `morpho.ss` in Hennig86 format.
- `read(prealigned:("28s.aln"),tcm:(1,1))` This command imports the `prealigned` nucleotide sequence file `28s.aln`, and sets the transformation cost matrix to be used in calculating the cost of the tree for these data, such that the cost of both substitutions and indels are set to 1.
- `transform(likelihood:(ncm))` The command `transform`, followed by `likelihood`, specifies the conditions to be applied when transforming these dynamic parsimony characters to dynamic likelihood characters. An `ncm` model, where each character is free to evolve at its own rate on every edge of the tree, will be employed.
- `read("18s.fas")` This command reads in the molecular data file of unaligned sequences (`18s.fas`).
- `transform((names:("18s.fas"),(likelihood:(tn93,rates:none, priors:estimate,gap:coupled,mpl))))` The command `transform`, followed by `likelihood`, specifies the conditions to be applied when transforming these dynamic parsimony characters to dynamic likelihood characters. Because this script includes both static (morphology and prealigned) and dynamic data, it is necessary to identify, using the *identifier* `names`, which file the `mpl` model will be applied to—in this case the `18s.fas` data file. A `tn93` model, with estimated equilibrium frequencies will be employed. In this model, indels will be treated as `coupled`.
- `search(max_time:00:01:00)` Specifies that the program will attempt as many builds, swaps, ratchets and tree fusings as possible within the specified time of one hour. All trees with the optimal score found are stored in memory.

- `swap(all,optimize:(model:never,branch:join_region))` Submits the current trees to a round of SPR and TBR swapping. Following each round, the model is `never` optimized, but a maximum of five branches (the new edge, and the two edges on either side of the join site) are optimized (`branch:join_region`).

- `report("All_LK.tre",trees:(branches),"All_lkm.txt",lkmodel)` This command reports a series of outputs of the results the search. A file containing a tree, in parenthetical notation, containing the optimal topological tree (`All_LK.tre`), along with the branch lengths `trees:(branches)` is reported. In addition, a file containing the parameter estimates of the likelihood analysis, i.e. likelihood score, the variant of likelihood used, the tree length (sum of branch lengths), the values of the parameter estimates for the entries of the substitution rate matrix ($\mathbf{Q}$), and the estimate of the value of the rate variation shape parameter, is also reported (`All_lkm.txt`).

- `exit()` This commands ends the `POY5` session.

Chapter 6

POY5 Frequently Asked Questions

This FAQ is supplementary documentation that aims to answer the most frequently poised questions to the POY5 developers and on the POY5 google groups.

Q. In running an analysis using a *.ss or Hennig86 file, I encountered a 'syntax' error, however, the characters continued to load and the analysis seemed to run. Is something wrong?

Q. My FASTA file contains sequences that are of poor 'quality', especially in the 5 prime and 3 prime regions of the sequences. How should these data files be prepared for analysis in `POY5`?

Q. I read in a tree that was generated from a previous analysis, however, the cost reported in the output window of the `Interactive Console` is different, why is this the case?

Q. Having run an analysis in `POY5`, I imported my tree file into TNT, but the tree costs are different. Is this an error?

Q. In trying to calculate Jackknife support values, I believe that all the values are inflated for the resulting tree. Why?

Q. Why is my prealigned data not treated as prealigned?

Q. Is it possible to exclude certain terminals from an analysis?

Q. Is it possible to report parsimony branch lengths in `POY5`?

Q. I would like to import trees from an earlier run, at what stage of the analysis should this be performed?

Q. Why is the root in the diagnosis file that I reported at the end of my analysis, not the same as the root that I `set`?

Q. What are "Numerical.linesearch; Very large slope in optimization function" warnings?

Q. What does the "Numerical.brent; hit max number of iterations" warning mean?

Q. Why are likelihood scores worse/different than other applications?

Q. What does POY stand for?

A. POY is a meta-acronym, which comes from an older program YAPP (Yet Another Phylogeny Program) that was written in `C`. This program, which was an extension of `MALIGN`, was the first designed around direct optimization. This program was rewritten in Ocaml (Ocaml YAPP), which was shortened to OY. The subsequent parallelization of this program yielded POY.

Q. I would like to visualize the implied alignment generated from my analysis, how do I do this?

A. The implied alignment can be exported from `POY5` in two ways:

The first is to export the implied alignment, using the command report. The implied alignment will be output in FASTA format (see `implied_alignments` (Section 3.3.19)).

The second is to transform using static_approx and report using the phastwinclad option (see `phastwinclad` (Section 3.3.19)). This will produce a file Hennig86 format.

These files can subsequently be imported into other programs such as Winclada or Mesquite, for visualization.

Q. Looking at the implied alignment generated from my `POY5` analysis, it looks very 'gappy'. Why?

A. POY is not, nor has it ever been an alignment program. Non-homologous, independent insertions are assigned their own columns with `POY5`, hence, the number of columns will expand with the number of insertions.

Q. I encountered a problem while running an analysis that I think might be a bug in the program, how should I report this?

A. All error and bug reports should be made directly to the `POY5` Mail Group. Before posting to this group, it is advised that the user search the history of previously posed questions, to make sure that it has not been answered previously. When reported to the Mail Group, users should include the following information:

– What steps will reproduce the problem;
– What is the expected output and what do you see instead;
– What version of the program are you using and on which operating system?

Q. My script won't run and I don't know where I went wrong. What should I do?

A. If a script won't run, the first thing to do is to check that there are no hidden characters in the script file. When constructing a script or a transformation cost matrix, it is important to do so in a text editor such as Notepad (for Windows), TextEdit (for Mac), or Nano (for Linux). Generating these files in a word processing application such as Microsoft Word may lead to the insertion of hidden characters, which can result in an error.

Secondly, the user is advised to check the log. If none was set, the user is advised to do so and rerun the script. The log will give some indication as to which errors were encountered, or which warnings were issued.

Q. When I run `POY5` in parallel, I get multiple, identical outputs to the screen, why?

A. It is likely that `POY5` was not properly compiled in parallel. You should check the `make` options.

Q. In running an analysis of custom alphabet characters, with the characters being transformed to `level` 5, I got the following error `"seg fault:11"`. In a previous analysis, the characters were transformed to `level` 4, and that worked without issue. What's wrong?

A. This is most likely an 'out of memory' error and is system specific and difficult to predict. This is beyond the control of the program. Storage and set up time increase combinatorially with level number.

Q. In running an analysis using a *.ss or Hennig86 file, I encountered a 'syntax' error, however, the characters continued to load and the analysis seemed to run. Is something wrong?

A. Syntax errors are of the form:
Error: Syntax error

```
Error: Unrecognized command between characters 93 and 94
Information: The file Fly.ss defines 9 static homology characters,
0 unaligned sequences, and 0 trees, containing 5 taxa
```

Hennig86 or *.ss files, have no formal definition. `POY5` does its best to parse the file properly, but it is incumbent upon the user to make sure that the data was read/parsed correctly. If an error such as this is encountered, the user should report the data (`report(data)`) to verify. A similar caution should be taken with NEXUS files, as very often files that have been generated and exported from other programs are not in the correct NEXUS format.

Q. My FASTA file contains sequences that are of poor 'quality', especially in the 5 prime and 3 prime regions of the sequences. How should these data files be prepared for analysis in `POY5`?

A. In cases such as this, partitioning the data is **highly** recommended. Partitioning or fragmenting the data can help to ameliorate the effects of poor sequences or missing data. Moreover, when a data file contains sequences that were downloaded from a data base such as GenBank, very often there is a lack of overlap of many of the sequences, as different studies may have utilized different priming regions. How best to 'chop' up your data is discussed in the `POY5` Heuristics chapter.

Q. I read in a tree that was generated from a previous analysis, however, the cost reported in the output window of the `Interactive Console` is different, why is this the case?

A. If you are reading in a tree generated from a previous analysis it is important to make sure that the same transformation cost matrix has been applied to the data. In addition, the tree must be fully resolved, otherwise it will be resolved arbitrarily.

Q. Having run an analysis in `POY5`, I imported my tree file into TNT, but the tree costs are different. Is this an error?

A. When importing `POY5` tree files into another program, such as TNT, it is important to mirror the same 'conditions' as to those in `POY5` during the time of analysis, i.e. same cost matrix, gaps treated as a fifth state. The correct data file associated with this tree file, must also be imported—this corresponds to the implied alignment that is associated with this tree.

In addition, the tree must be fully resolved, otherwise it will be resolved arbitrarily.

Q. In trying to calculate Jackknife support values, I believe that all the values are inflated for the resulting tree. Why?

A. Although it is possible to calculate Jackknife and Bootstrap support values for trees constructed using dynamic homology characters, it is not recommended since resampling of dynamic characters occurs at the fragment, rather than nucleotide, level. (Of course this is a mute point if the dataset consists of a multitude of fragments.) Consequently, the bootstrap and jackknife support values calculated for dynamic characters are not directly comparable to those calculated based on static character matrices. In order to perform character sampling at the level of individual nucleotides, the dynamic characters **must** be transformed into static characters using `static_approx` argument of the command transform (Section 3.3.26) prior to executing calculate support. The static_approx is conditioned or based on that tree.

Q. Why is my prealigned data not treated as prealigned?

A. By default, upon importing prealigned sequence data, the gaps are removed and the sequences are treated as dynamic homology characters. To preserve the alignment the data must be imported using the `prealigned` argument of the command `read`. Unless specified using the `prealigned`, data that is read by the program is UNALIGNED and the gaps are stripped from the data file.

Q. Is it possible to exclude certain terminals from an analysis?

A. The exclusion of terminals (or for that matter, characters) is easily achieved by selecting, with the use of the identifiers. For example `select(terminals,not files:("Taxa_removed.txt"))` will exclude all the taxa that are included in this file. This is the inverse of `select(terminals, files:("Taxa_keep.txt"))`. Alternatively, if the user does not wish to generate a terminals file, the taxon names can be specified using the `not names` identifier, e.g. `select(terminals,not names:("Taxon1","Taxon4"))`.

Q. Is it possible to report parsimony branch lengths in `POY5`?

A. Yes it is. This is achieved by reporting `branches`, along with the

`trees`, and specifying the collapse mode. For example `report("Run1.tre", trees:(branches:single))` will report a tree to the file `Run1.tre` with the parsimony branch lengths included. The argument `single` specifies that zero length branches will be collapsed.

Q. I would like to import trees from an earlier run, at what stage of the analysis should this be performed?

A. When running a script that includes reading in trees from a previous analysis, these trees **must** be read in **after** the build stage. If the trees are read in before the build they will be replaced by the trees generated during the build.

Q. Why is the root in the diagnosis file that I reported at the end of my analysis, not the same as the root that I `set`?

A. This is because the tree length heuristics may be based on an alternate rooting scheme than that used for the `newick` or `graphic trees` output.

Q. What are "Numerical.linesearch; Very large slope in optimization function" warnings?

A. These messages normally appear in the Dynamic likelihood routines. They indicate that the gradient of the parameters for the current data-set has a large absolute slope and the numerical routine may not converge properly. Normally, because of multiple passes of the optimization routine, we easily break out of these regions and the routine will stabilize.

Under static likelihood data this warning message is rarely seen and may be an issue. One should rediagnose the tree for stabilization of the parameters of the model.

Q. What does the "Numerical.brent; hit max number of iterations" warning mean?

A. This happens when the numerical routine does not converge in a maximum number of steps. This is usually an acceptable situation to happen as more optimization rounds will likely occur over the data.

Q. Why are likelihood scores worse/different than other applications?

A. There are a number of issues to consider. If the values are close under the same model of evolution then numerical issues due to finite precision arithmetic of decimal numbers can cause slight rounding errors in an analysis to build up. Although we use standard techniques to limit the accumulation of these errors, they inevitably occur and small differences in likelihood scores are absolutely normal and should not be a concern. These differences even occur within the same application run on different architectures and compiler options.

If the model is not hierarchical then comparisons are not relevant. Unfortunately, the number of states in the model of evolution matters, as well as cost assignments of Maximum Parsimonious Likelihood (MPL) and Maximum Average Likelihood (MAL). Thus, a four state model of evolutions' likelihood score cannot be compared directly with a five state model. There is added cost that result from the probability of an additional state in an analysis.

One should also check that the `exhaustive` option is set for the optimization routines. It is set by default but ensure that, if it changed somewhere in the script, one sets it back–especially if one plans to do application comparisons.

Bibliography

[1] H. Akaike. Information theory and an extension of the maximum likelihood principle. *Second International Symposium on Information Theory*, pages 267–281, 1973.

[2] D. A. Bader, B. M. E. Moret, T. Warnow, S. K. Wyman, M. Yan, J. Tang, A. C. Siepel, and A. Caprara. Grappa, version 2.0. http://www.cs.unm.edu/m̃oret/grappa. Technical report, University of New Mexico, 2002.

[3] D. Barry and J. Hartigan. Statistical analysis of hominid molecular evolution. *Statistical Science*, 2:191–210, 1987.

[4] M. Blanchette, G. Bourque, and D. Sankoff. *Genome Informatics*, chapter Breakpoint phylogenies, pages 25–34. Universal Academy Press, Tokyo, 1997. S. Miyano and T. Takagi–eds.

[5] G. Bourque and P. A. Pevzner. Genome-scale evolution: Reconstructing gene orders in the ancestral species. *Genome Research*, 12:26–36, 2002.

[6] K. Bremer. The limits of amino acid sequence data in angiosperm phylogenetic reconstruction. *Evolution*, 42:795–803, 1988.

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

[8] A. Caprara. On the practical solution of the reversal median problem. In *Proc. 1st Int'l Workshop Algorithms in Bioinformatics (WABI'01)*, volume 2149 of *Lecture Notes in Computer Science*, pages 238–251. Springer-Verlag, 2001.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[10] A. C. E. Darling, B. Mau, F. R. Blattner, and N. T. Perna. Mauve: Multiple alignment of conserved genomic sequence with rearrangements. *Genome Research*, 14:1394–1403, 2004.

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

[12] J. S. Farris. Hennig86, 1988.

[13] J. S. Farris. The retention index and the rescaled consistency index. *Cladistics*, 5:417–419, 1989.

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

[15] J. Felsenstein. *PHYLIP*, 1980.

[16] J. Felsenstein. Evolutionary trees from dna sequences: a maximum likelihood approach. *Journal of Molecular Evolution*, 17:368–376, 1981.

[17] J. Felsenstein. Confidence limits on phylogenies: An approach using the bootstrap. *Evolution*, 39(4):783–791, 1985.

[18] J. Felsenstein. *Inferring Phylogenies*. Sinauer, Sunderland, MA, 2004.

[19] D. R. Frost, M. T. Rodrigues, T. Grant, and T. A. Titus. Phylogenetics of the lizard genus *Tropidurus* (Squamata: Tropiduridae: Tropidurinae): direct optimization, descriptive efficiency, and sensitivity anaylsis of congruence between molecular data and morphology. *Molecular Phylogenetics and Evolution*, 21(3):352–371, 2001.

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

[21] P. A. Goloboff, C. I. Mattoni, and A. S. Quinteros. Continuous characters analyzed as such. *Cladistics*, 22:589–601, 2006.

[22] X. Gu, Y. X. Fu, and W. H. Li. Maximum likelihood estimation of the heterogeneity of substitution rate among nucleotide sites. *Molecular Biology and Evolution*, 12:546–557, 1995.

[23] S. Hanenhalli and P. A. Pevzner. Transforming a cabbage into a turnip (polynomial algorithm for sorting signed permutations by reversals). In

Proceedings of the 27th Annual ACM-SIAM Symposium on the Theory of Computing, pages 178–189, 1995.

[24] M. Hasegawa, H. Kishino, and T. Yano. A new molecular clock of mitochondrial dna and the evolution of hominoids. *Proceedings of the Japan Academy, series B*, 60:95–98, 1984.

[25] M. D. Hendy and D. Penny. Branch and bound algorithms to determine minimal evolutionary trees. *Mathematical Biosciences*, 60:133–142, 1982.

[26] V. Jayaswal, L. S. Jermiin, and J. Robinson. Estimation of phylogeny using a general markov model. *Evolutionary Bioinformatics Online*, 1:62, 2005.

[27] T. H. Jukes and C. R. Cantor. Evolution of protein molecules. In N. H. Munro, editor, *Mammalian Protein Metabolism*, pages 21–132. Academic Press, New York, 1969.

[28] M. Källersjö, J. S. Farris, A. G. Kluge, and C. Bult. Skewness and permutation. *Cladistics*, 8:275–287, 1992.

[29] M. Kimura. A simple method for estimating evolutionary rate of base substitution through comparative studies of nucleotide sequences. *Journal of Molecular Evolution*, 16:111–120, 1980.

[30] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[31] A. G. Kluge and J. S. Farris. Quantitative phyletics and the evolution of anurans. *Systematic Zoology*, 30:1–32, 1969.

[32] T. Margush and F. R. McMorris. Consensus n-trees. *Bulletin of Mathematical Biology*, 43:239–244, 1981.

[33] G. McGuire, M. C. Denham, and D. J. Balding. Models of sequence evolution for dna sequences containing gaps. *Molecular Biology and Evolution*, 18(4):481–490, 2001.

[34] E. Mossel, S. Roch, and M. Steel. Shrinkage effect in ancestral maximum likelihood. *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, 6(1):126–133, 2009.

[35] J. Neyman. Molecular studies in evolution: a source of novel statistical problems. In S. S. Gupta and J. Yackel, editors, *Statistical Decision Theory and Related Topics*, pages 1–27, 1971.

[36] K. C. Nixon. The parsimony ratchet, a new method for rapid parsimony analysis. *Cladistics*, 15(4):407–414, 1999.

[37] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences of the United States of America*, 85:2444–2448, 1988.

[38] B. Redelings and M. Suchard. Incorporating indel information into phylogeny estimation for rapidly emerging pathogens. *BMC evolutionary biology*, 7(1):40, 2007.

[39] B. D. Redelings and M. A. Suchard. Joint bayesian estimation of alignment and phylogeny. *Systematic Biology*, 54(3):401–418, 2005.

[40] E. Rivas and S. R. Eddy. Probabilistic phylogenetic inference with insertions and deletions. *PLoS Computational Biology*, 4(9):e1000172, 2008.

[41] N. Saitou and M. Nei. The neighbor-joining method: A new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4:406–425, 1987.

[42] D. Sankoff and M. Blanchette. The median problem for breakpoints in comparative genomics. *Computing and Combinatorics 3rd Annual Int. Conf. COCOON 97*, 1276:251–263, 1997.

[43] R. T. Schuh and J. T. Polhemus. Analysis of taxonomic congruence among morphological, ecological, and biogeographic data sets for the leptopodomorpha (Hemiptera). *Systematic Zoology*, 29:1–26, 2011.

[44] G. Schwarz. Estimating the dimension of a model. *Annals of Statistics*, 19:205–221, 1978.

[45] A. C. Siepel and B. M. E. Moret. Finding an optimal inversion median: experimental results. In *Proc. 1st Int'l Workshop Algorithms in Bioinformatics (WABI'01)*, volume 2149 of *Lecture Notes in Computer Science*, pages 189–203. Springer-Verlag, 2001.

[46] A. Stamatakis, T. Ludwig, and H. Meier. Raxml-vi-hpc: Maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models. *Bioinformatics*, 22(21):2688–2690, 2006.

[47] M. Steel and D. Penny. Parsimony, likelihood, and the role of models in molecular phylogenetics. *Molecular Biology and Evolution*, 17(6):839–850, 2000.

[48] N. Sugiura. Further analysis of the data by akaike's information criterion and the finite corrections. *Communications in Statistics, Theory and Methods*, A7:13–27, 1978.

[49] J. Sullivan, D. L. Scofford, and G. J. P. Naylor. The effect of taxon-sampling on estimating rate heterogeneity parameters on maximum-likelihood models. *Molecular Biology and Evolution*, 52:7649–664, 1999.

[50] D. L. Swofford and G. J. Olsen. Phylogeny reconstruction. In D. Hillis and C. Moritz, editors, *Molecular Systematics*, chapter 11, pages 411–501. Sinauer Ass. Inc., Sunderland, Massachusetts, USA, 1990.

[51] H. Tamura and M. Nei. Estimation of the number of nucleotide substitutions in the control region of mitochondrial dna in humans and chimpanzees. *Molecular Biology and Evolution*, 10:512–526, 1993.

[52] S. Tavaré. Some probabilistic and statistical problems on the analysis of dna sequences. *Lectures on Mathematics in the Life Sciences*, 17:57–86, 1986.

[53] C. Tuffley and M. Steel. Links between maximum likelihood and maximum parsimony under a simple model of site substitution. *Bulletin of Mathematical Biology*, 59(3):581–607, 1997.

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

[55] A. Varón and W. C. Wheeler. Local search for the generalized tree alignment problem. *BMC Bioinformatics*, 14:66, 2013.

[56] L. S. Vinh, A. Varón, D. Janies, and W. C. Wheeler. Towards phylogenomic reconstruction. In *Proceedings of the International Conference on Bioinformatics and Computational Biology*, pages 98–104, Las Vegas, Nevada, USA, 2007. CSREA Press.

[57] L. S. Vinh, A. Varón, and W. C. Wheeler. Pairwise alignment with rearrangements. *Genome informatics*, 17(2):141–151, 2006.

[58] W. C. Wheeler. Sequence alignment, parameter sensitivity, and the phylogenetic analysis of molecular data. *Systematic Biology*, 44(3):321–331, 1995.

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

[60] W. C. Wheeler. Fixed character states and the optimization of molecular sequence data. *Cladistics*, 15(4):379–385, 1999.

[61] W. C. Wheeler. Homology and DNA sequence data. In G.P. Wagner, editor, *The Character Concept in Evolutionary Biology*, pages 303–318. Academic Press, New York, 2001.

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

[63] W. C. Wheeler. *Optimization Alignment: down, up, error, and improvements*. Techniques in Molecular Systematics and Evolution. Birkhäuser, Basel, Boston, Berlin, 2002.

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

[65] W. C. Wheeler. Iterative pass optimization of sequence data. *Cladistics*, 19:254–260, 2003.

[66] W. C. Wheeler. Search-based character optimization. *Cladistics*, 19:348–355, 2003.

[67] W. C. Wheeler. Dynamic homology and the likelihood criterion. *Cladistics*, 22:157–170, 2006.

[68] W. C. Wheeler. *Systematics: A Course of Lectures*. Wiley-Blackwell, Oxford, 2012.

[69] W. C. Wheeler, L. Aagesen, C. P. Arango, J. Faivoich, T. Grant, C. D'Haese, D. Janies, W. L. Smith, A. Varón, and G. Giribet. *Dynamic Homology and Systematics: A Unified Approach*. American Museum of Natural History, 2006.

[70] W. C. Wheeler, J. Gatesy, and R. DeSalle. Elision: A method for accommodating multiple molecular sequence alignments with alignment-ambiguous sites. *Molecular Phylogenetics and Evolution*, 4(1):1–9, 1995.

[71] W. C. Wheeler and N. Lucaroni. Maximum likelihood and the general tree alignment problem. *In prep.*, 2013.

[72] Z. Yang. Maximum likelihood phylogenetic estimation from dna sequences with variable rates over sites: Approximate methods. *Journal of Molecular Evolution*, 39:105–111, 1994.

[73] L. Zou, E. Susko, C. Field, and A. J. Roger. The parameters of the barry and hartigan general markov model are statistically nonidentifiable. *Systematic Biology*, 60(6):872–875, 2011.

[74] D. J. Zwickl. *Genetic algorithm approaches for the phylogenetic analysis of large biological sequence datasets under the maximum likelihood criterion.* PhD thesis, University of Texas at Austin, 2005.

General Index