POY 4.0

Program Documentation
Version 4.0.2870

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

American Museum of Natural History
www.amnh.org

Andrés Varón
Division of Invertebrate Zoology, American Museum of Natural History, New York, NY, U.S.A.
Computer Science Department, The Graduate School and University Center, The City University of New York, NY, U.S.A.

Le Sy Vinh
Ward C. Wheeler
Ilya Tëmkin
Megan Cevasco
Division of Invertebrate Zoology, American Museum of Natural History, New York, NY, U.S.A.

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

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

Julián Faivovich
Departamento de Zoologia, Instituto de Biociências, Universidade Estadual Paulista, Brasil

Taran Grant
Faculdade de Biociências, Pontifcia Universidade Católica do Rio Grande do Sul (PUCRS), Brasil

William Leo Smith
Department of Zoology, The Field Museum of Natural History, Chicago, IL, U.S.A.

The American Museum of Natural History

Varón, A., L. S. Vinh, I. Bomash, W. C. Wheeler. 2008. `POY` 4.0.2870. New York, American Museum of Natural History. Documentation by Varón, A., L. S. Vinh, I. Bomash, W. Wheeler, I. Tëmkin, M. Cevasco, K. M. Pickett, J. Faivovich, T. Grant, and W. L. Smith.
`http://research.amnh.org/scicomp/projects/poy.php`

Available online at `http://research.amnh.org/scicomp/projects/poy.php` and `http://code.google.com/p/poy4/`

Contents

Chapter 1

What is POY4

POY4 is a flexible, multi-platform program for phylogenetic analysis of molecular and other data. An essential feature of POY4 is that it implements the concept of dynamic homology [26, 27] allowing optimization of unaligned sequences. POY4 offers flexibility for designing heuristic search strategies and implements an array of algorithms including multiple random addition sequence, swapping, tree fusing, tree drifting, and ratcheting. As output, POY4 generates a comprehensive character diagnosis, graphical representations of cladograms and their user-specified consensus, support values, and implied alignments. POY4 provides a unified approach to co-optimizing different types of data, such as morphological and molecular sequence data. In addition, POY4 can analyze entire chromosomes and genomes, taking into account large-scale genomic events (translocations, inversions, and duplications).

1.1 The structure of POY4 documentation

This first chapter, *POY4 Quick Start*, will get you started using POY4. The first few sections are intended to provide detailed instructions on how to obtain and install POY4, introduce the user to the program's two working environments, the *Graphical User Interface* and the *Interactive Console*. These sections also show how to initiate a POY4 session and point to the 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 second chapter, *POY4 Commands*, describes POY4 commands and their valid syntax. It also includes examples of simple operations for every command. The third chapter discusses

the heuristic procedures used in POY4. Their understanding helps creating building efficient search strategies. More advanced operations are described in the fourth chapter, *POY4 Tutorials*. In addition to the general index, this document contains a *POY3.0 Command Line Index*, intended to provide a link between the commands used in POY3 and the commands used in POY4.

Chapter 2

POY4 Quick Start

2.1 Requirements: software and hardware

2.1.1 Software

POY4 is a platform-independent, open-source program that can be compiled for many operating systems and hardware configurations, including Mac OSX, Microsoft Windows XP, and Linux. POY4 *binaries* (compiled application file) is the only piece of software necessary to run POY4. The intuitive graphical user interface of POY4 provides all the functionality for running analyses using pull-down menus and field selections, as well as creating and running POY4 scripts. Some utility programs (such as Notepad and Ghostscript for Windows, TextEdit for Mac, or Nano for Linux), can help preparing POY4 scripts and formatting datafiles, while others (such as Ghostscript, Ghostview, or GSview) can facilitate viewing the graphical output in PS (PostScript) format.

2.1.2 Hardware

POY4 runs on a variety of computers from laptops and desktops to Beowulf clusters of various sizes to 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 POY4

POY4 installers for Linux, Windows XP, and Mac OSX, source code, and documentation in PDF format are available from the POY4 website at the

American Museum of Natural History Computational Sciences:

`http://research.amnh.org/scicomp/projects/poy.php`

The latest source code can also be obtained from POY4 Google Group website:

`http://code.google.com/p/poy4/source`

The following detailed step-by-step instructions will guide you through downloading and installing POY4 binaries for various platforms.

Windows

- Download *poy4* folder to the desktop by selecting *WinXP* download link.
- Open *POY_Installer.exe* from the *poy4* folder and follow the installation instructions. You will need Administrator privileges to install the application. By default, POY4 is installed without parallel execution support. If you have Windows XP SP2 or Windows Vista and more than one core or processor, you can take advantage of your processing power by installing the parallel components. To do so, instead of a typical installation, select the complete installation.

  Important: the complete installation includes MPICH2 1.06 p1. MPICH is used to communicate processes during parallel execution. If you already have MPICH2 installed (if you didn't know what it is you most likely don't have it), select the custom installation option and remove that component. During the first execution in parallel you will be asked by the Windows Firewall to unblock POY4 and MPICH, this is necessary for the successful execution of the program.

Mac OSX

- Download *poy_parallel_buildXXXX.dmg* disk image to the desktop.
- Drag the POY4 application from the disk *poy4* and drop it into the *Applications* folder on the hard drive.

Linux

- Download the gzipped file.

- Untar and ungzip the *poy4.tar.gz* file.
- Run the command `tar -Pxvzf poy4.tar.gz` as a super user in the newly created *poy4* directory. The GUI will be installed in `/opt/poy4/Contents/POY` directory and terminal binaries in `/opt/poy4/Resources/ncurses_poy` directory.

Compiling from the source

In order to compile `POY4` the following tools are required:

1. The GNU Make 3.8 utility;
2. OCaml version 3.10.0. or later;
3. A C compiler, for example The GNU Compiler Collection;
4. Optionally, the ncurses library for a nice interactive console or the read line library. If none is available, the flat interface does not require read line or ncurses.

Download, ungzip, and untar the `POY4` source code; In order to compile under default setting type:

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

All the configuration options can be found in `./configure --help`.

`POY4` can also be run in parallel environments using the Message Passing Interface. Your system administrator has likely already one installed and should be able to provide you with proper paths to set your config file.

2.3 The Graphical User Interface

`POY4` provides two working environments: the *Graphical User Interface* and the *Interactive Console*. The *Graphical User Interface* has a user-friendly appearance as other native stand-alone applications where different functions are accessible through menus and windows. Thus, the entire analysis can be carried out clicking on appropriate selections and, where necessary, typing specifications in designated fields. The *Interactive Console*, however, requires a detailed knowledge of `POY4` commands, their arguments, and the

conventions of POY4 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 Linux(GTK+) and Windows versions contain the same items and functionality, differing only in the generic window format specific to each platform.

When POY4 is first opened, two items appear on the screen: the menu bar across the top and the *POY Launcher* window (Figure 2.1). (Note that in Linux and Windows the menu bar is within the launcher window.)

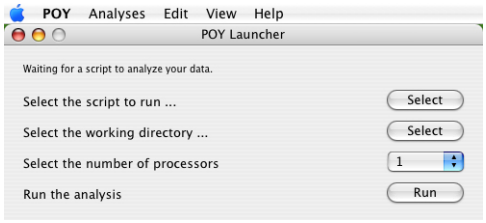


Figure 2.1: The POY4 menu bar and the *POY Launcher* window. These items appear when POY4 is opened.

2.3.1 POY menu bar

The menu bar contains the following drop-down menus:

POY (Mac OSX only) contains generic items as other Mac OSX applications. It includes *Quit POY* tab that closes the program. This menu also allows contains selection of the *About POY* window (Figure 2.2) that lists the current version of POY4, a copywrite statement, and the address of the POY4 website.

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* and the *Interactive Console*.

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

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

Help opens the `POY4` *Program Documentation* in PDF format (requires a PDF viewer).

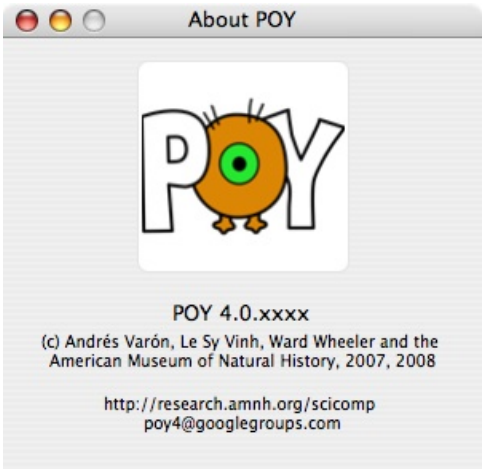


Figure 2.2: The *About POY* window.

2.3.2 POY Launcher

The *POY Launcher* is the only window that automatically opens upon starting `POY4`. This allows the user to import a previously created script, 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 `POY4` 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 `POY4` script.

Select the number of processors The selection of the number of processors is disabled for Linux and Windows platforms. Once specified, the selection is applied to all subsequent analyses in the current `POY4` session. (Parallelization is not supported in interactive sessions, see Section 2.4.)

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 POY4 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, POY4 issues an error message in the upper part of the *POY Launcher* window, such as `POY finished with an error`.

2.3.3 The *Analyses* menu

The *Analyses* menu is the main toolbox of the POY4 GUI interface (Figure 2.3, 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 generating. 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 the 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

Simple Search

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, subsequent to which a subset of trees is selected for the report. The *Simple Search* window (Figure 2.3, right) provides the most common and basic options for a standard tree search in POY4 that must either (in some cases or) be selected by clicking appropriate buttons or typed. Note that *all the empty fields must be filled in.* The window is subdivided into four sections:

Input Files Contains the list of files that are to be input into POY4. These include character files in nucleotide, Hennig86, and Nexus formats and tree files. (Character data in other formats can be imported by specifying additional argument in the script. See `read` (Section 3.3.14).)

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

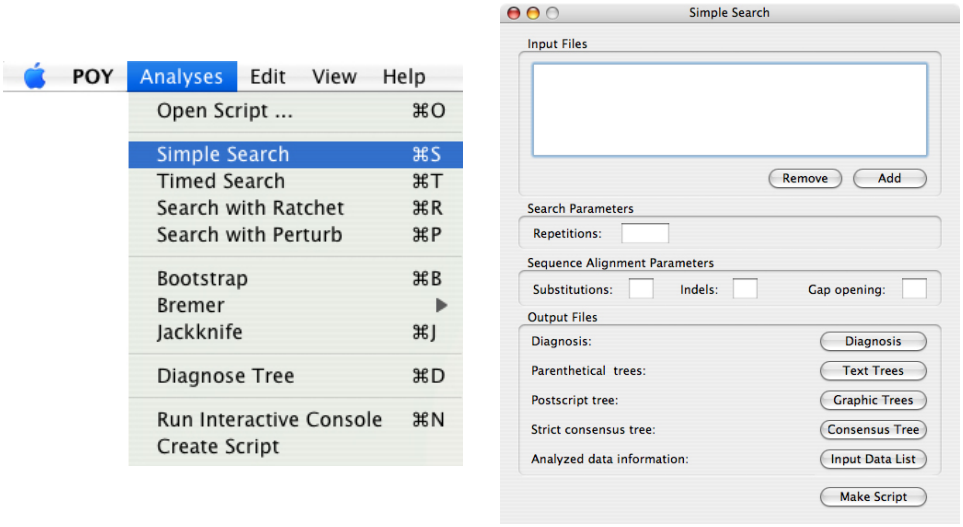


Figure 2.3: The *Simple Search* window. Selecting *Simple Search* from the *Analysis* menu (left) opens the *Simple Search* window options (right).

Sequence Alignment Parameters Holds fields to specify the substitution, indel, and gap opening costs. Enter `0` if no gap opening cost is desired. If the value of a parameter is not specified, the default values is used. (See the *POY Commands* chapter (3.1.1).)

Output Files Designates the names and locations of files containing different kinds of results (implied by their respective titles) of the analysis. If no names are specified, the default names are generated.

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 2.4). 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, `POY4` will issue a request to save the script. Thus, not only does `POY4` execute the script but it also creates the record of the type of analysis (including all user-defined specifications) that was performed.

Timed Search

Timed search (Figure 2.5) implements a default search strategy that effectively combines tree building with TBR branch swapping, parsimony

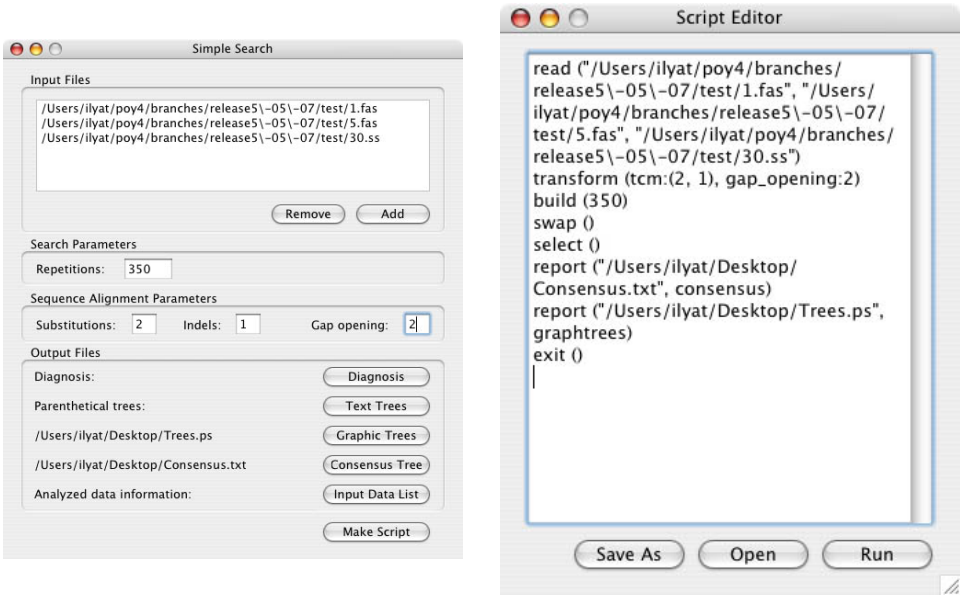


Figure 2.4: The *Simple Search* window with specified search parameters and output files (left) and the corresponding *Script Editor* window.

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 aborting the search.

Search with Ratchet

The parsimony ratchet is a heuristic strategy to escape the local optima during tree searching [16]. The *Search with Ratchet* (Figure 2.6) follows

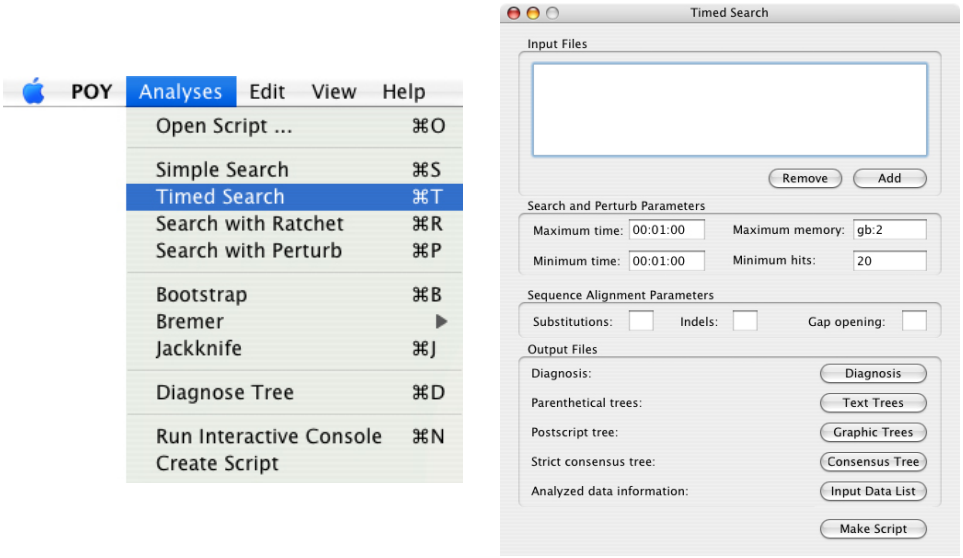


Figure 2.5: The *Timed Search* window. Selecting *Timed Search* from the *Analysis* menu (left) and viewing the *Timed Search* window options (right).

the same basic steps of a simple search but includes the ratchet step after the swap. In addition to the same four parameter groups 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 parsimony 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 2.7) provides an alternative means to escape local optima by changing the transformation cost matrix of the molecular characters, a procedure similar in spirit to the parsimony ratchet. In addition to the same four parameter groups 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:

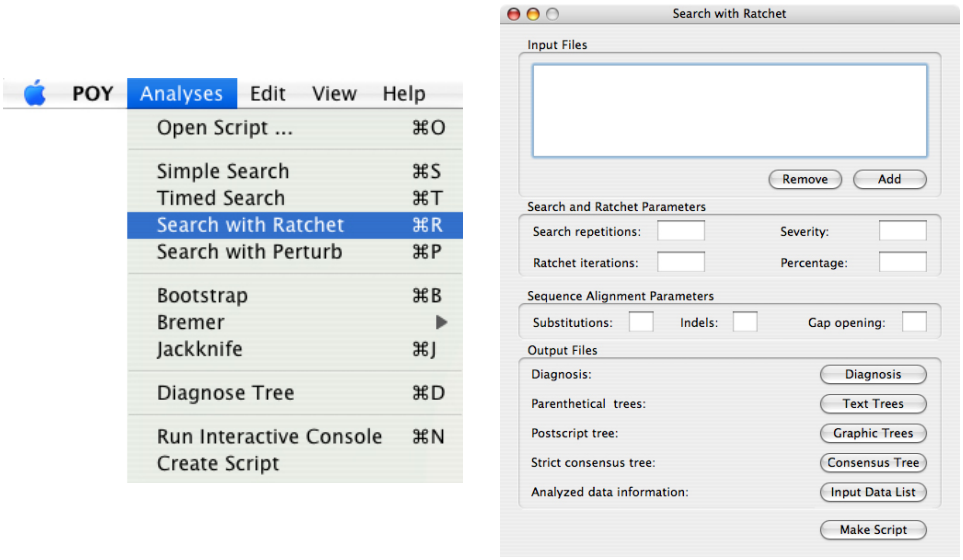


Figure 2.6: The *Search with Ratchet* window. Selecting *Search with Ratchet* from the *Analysis* menu (left) and viewing the *Search with Ratchet* window options (right).

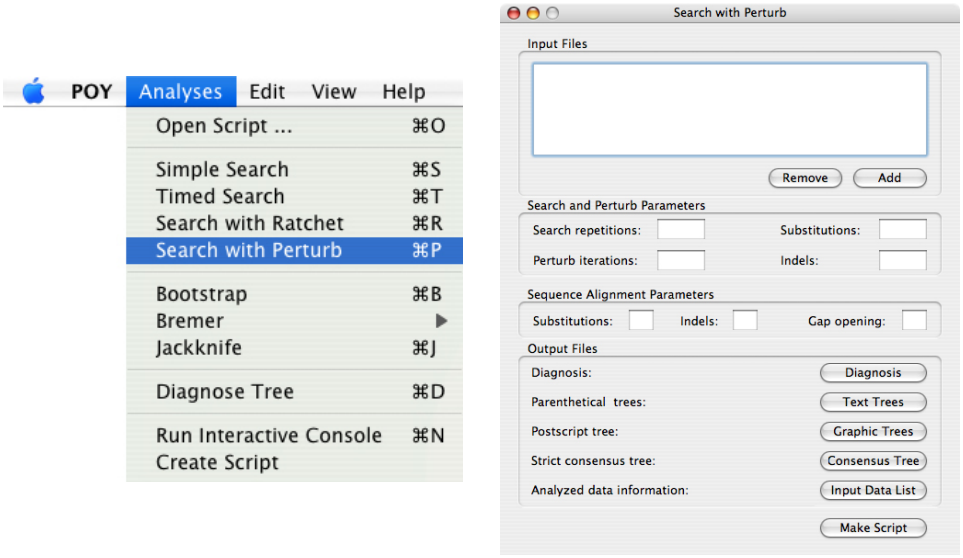


Figure 2.7: The *Search with Perturb* window. Selecting *Search with Perturb* from the *Analysis* menu (left) and viewing the *Search with Perturb* window options (right).

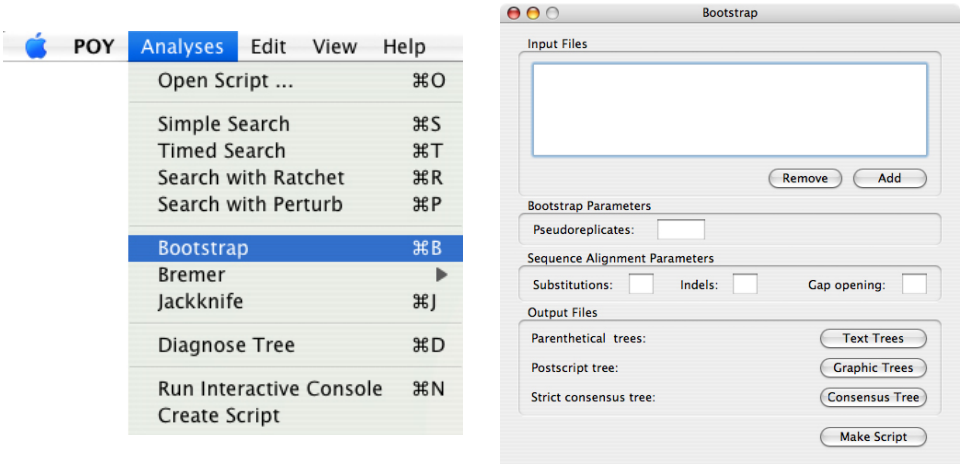


Figure 2.8: The *Bootstrap* window. Selecting *Bootstrap* from the *Analysis* menu (left) and viewing the *Bootstrap* window options (right).

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.

Support calculation options

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

The *Bootstrap* window (Figure 2.8) specifies parameters for estimating the Bootstrap support values. In addition to the *Simple Search* window fields, it contains a *Pseudoreplicates* field to specify the number of bootstrap pseudoreplicates.

Bremer

The *Bremer* option (Figure 2.9) is divided into two windows: the *Search for Bremer* window, that specifies the Bremer support [2, 12] calculation parameters, and the *Report Bremer* window to format the output of the results (Figure 2.10).

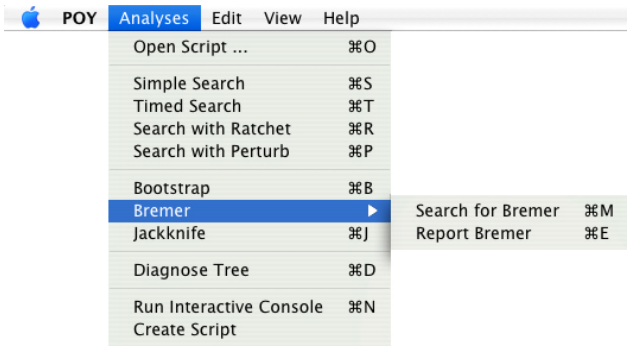


Figure 2.9: Selecting the *Bremer* windows from the *Analysis* menu.

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

Jackknife

The *Jackknife* window (Figure 2.11) specifies parameters for estimating the Jackknife support values. In addition to the *Simple Search* window fields, *Jackknife Parameters* it contains fields to specify the number of Jackknife

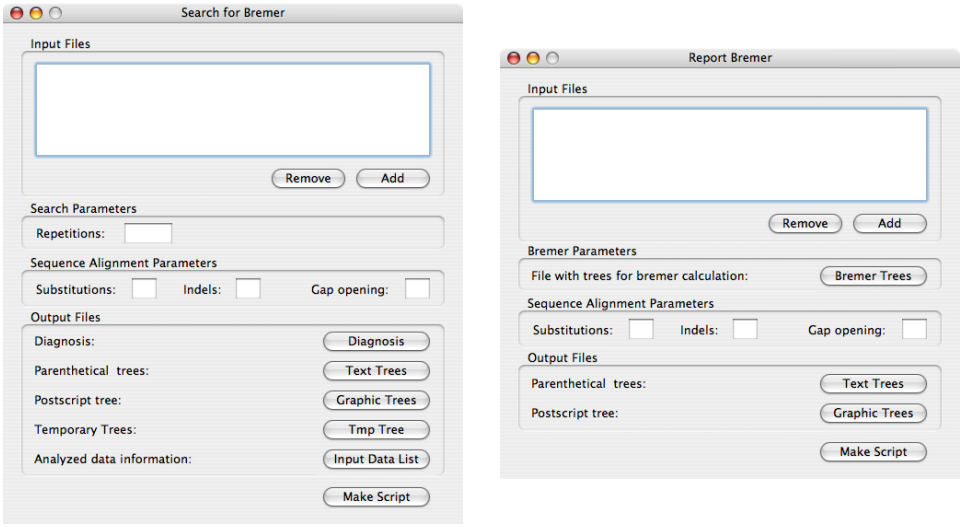


Figure 2.10: Viewing the options of the *Search for Bremer* (left) and the *Report Bremer*(right) windows.

pseudoreplicates (*Pseudoreplicates*) and the number of characters to remove (*Remove*).

Pseudoreplicates Specifies the number of resampling iterations.

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

Diagnosis

Diagnose Tree

The *Diagnose Tree* window (Figure 2.12) 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.

Script editing and the Interactive Console

Open POY Script

Selecting *Open POY Script* (Figure 2.13) displays the *POY Launcher* window (Figure 2.1), the function of which is described above.

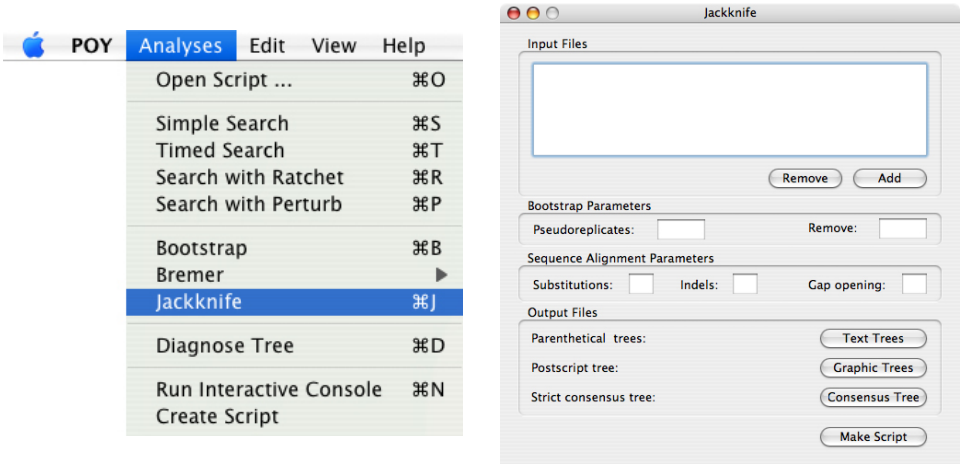


Figure 2.11: The *Jackknife* window. Selecting *Jackknife* from the *Analysis* menu (left) and viewing the *Jackknife* window options (right).

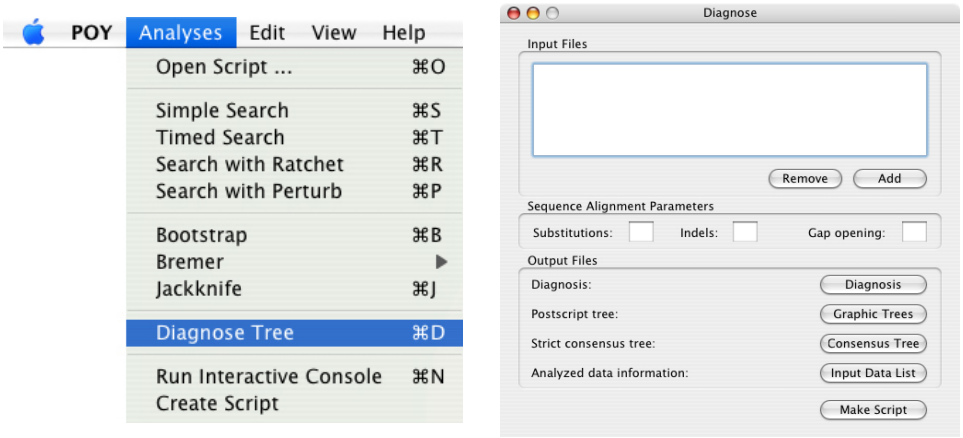


Figure 2.12: The *Diagnose* window. Selecting *Diagnose Tree* from the *Analysis* menu (left) and viewing the *Diagnose* window options (right).

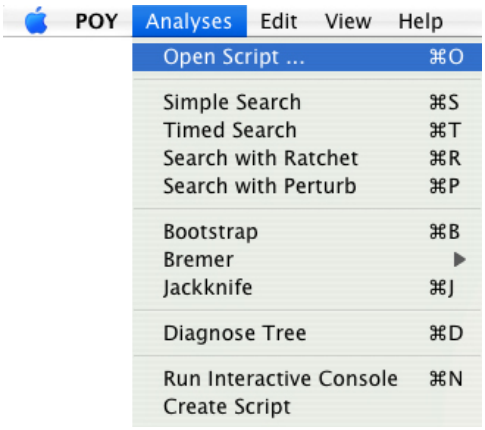


Figure 2.13: The *Open POY script* selection opens the *POY Launcher* window.

Run Interactive Console

Selecting *Run Interactive Console* (Figure 2.14) opens the ncurses interface that enables the user to run the analysis interactively by entering `POY4` commands directly via the command-line interface of the *Interactive Console*. Note that the *Interactive Console* does *not* run in parallel.

Create Script

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

2.3.4 The *View* menu

The *View* menu contains the *Output* window which is subdivided into two fields: the upper *Results and Errors* and lower *Status* (Figure 2.15). These fields display, respectively, the results (including warning and error messages) and the current state of the analysis. These fields are not updated 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 and Linux.

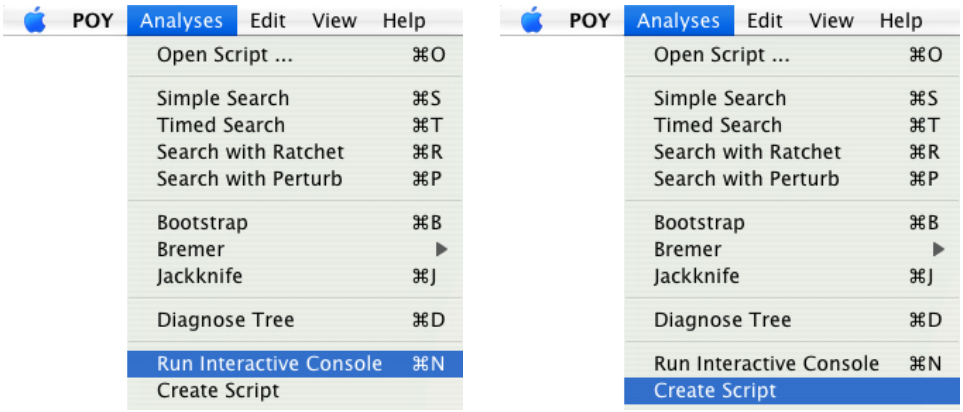


Figure 2.14: The *Run Interactive Console* selection (left) opens `POY4` interactive console in a new window. The *Create Script* selection opens the *Script Editor* window (Figure 2.4).

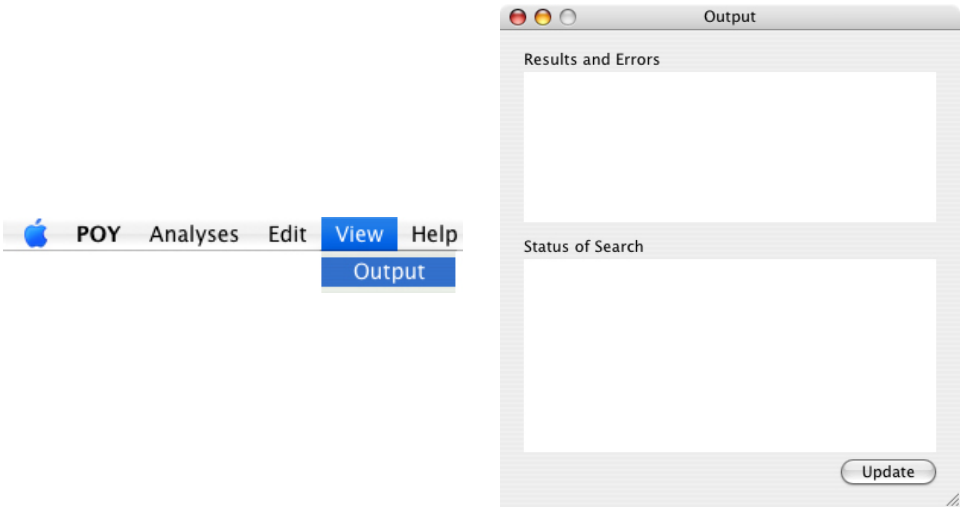


Figure 2.15: Selecting the *Output* window (left) and viewing the *Results and Errors* and *Status of Search* fields.

2.4 Using the Interactive Console

This section will help you get started using the `POY4` *Interactive Console* and will prepare you for the more extensive, technical descriptions in the next chapter, *POY4 Commands*. Now that you are acquainted with the program's interface, learned how to initiate, and exit or interrupt a `POY4` session, and how to obtain help, you are well prepared to run your first analysis. This chapter will teach you how to input datafiles, 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 strict consensus, and generate support values.

For the purpose of this exercise, three datafiles are used available at `http://research.amnh.org/scicomp/projects/poy.php`:

- `18s.fas` and `28s.fas` contain unaligned DNA sequences (partial 18S and 28S ribosomal DNA) in FASTA format. [17]
- `morpho.ss` contains a morphological data matrix in Hennig86 format. [5]

Once `POY4` has been launched and the interface (Figure 2.16) had appeared on the screen, the data can be input and the analysis can proceed. As you follow the instructions, you are encouraged to consult the help file by using the command `help` (see Section 2.6 to learn more about `POY4` commands and their arguments).

2.4.1 The interface

The *Interactive Console* provides a terminal environment with enhanced ability to display the results and the state of the analysis. It is recommended to use 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 `POY4` commands, their arguments, and the conventions of `POY4` 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 2.16):

POY Output (Figure 2.16, upper box) displays the status of the imported data, outputs the results of the phylogenetic analyses (such as trees, character diagnoses, and implied alignments), reports errors, and displays descriptions of `POY4` commands.

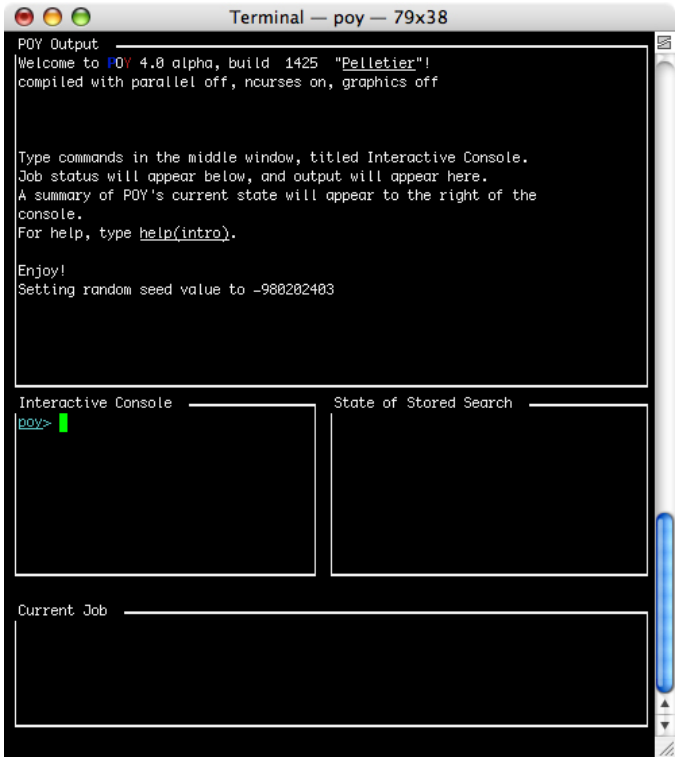


Figure 2.16: POY4 interface displayed in the Terminal window prior to analysis. Note the cursor at the POY4 prompt in the *Interactive Console* and that the *State of Stored Search* and *Current Job* windows are empty.

Interactive Console (Figure 2.16, 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 `POY4` commands.)

State of Stored Search (Figure 2.16, 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 memory and displays the range of their costs.

Current Job (Figure 2.16, lower box) describes the currently running operation. When the operation is completed, the box is blank.

2.4.2 Starting a `POY4` session using the *Interactive Console*

Windows

- Start>All Programs>POY>POY Interactive Console

Mac OSX

- Double-click `POY4` application icon to start the program.
- Select *Run Interactive Console* from the *Analyses* menu.

Linux

- Add `/opt/poy4/Resources/` to your `PATH` and run `ncurses_poy` from a terminal.

2.4.3 Entering commands

Once the `POY4` interface is called, the cursor appears in the *Interactive Console* and `POY4` is ready to accept commands. The interactive console does not support using the mouse and, as 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 of retyping commands anew during a `POY4` session, keyboard shortcuts can be used: Control-P ("previous") and Control-N ("next") will scroll through

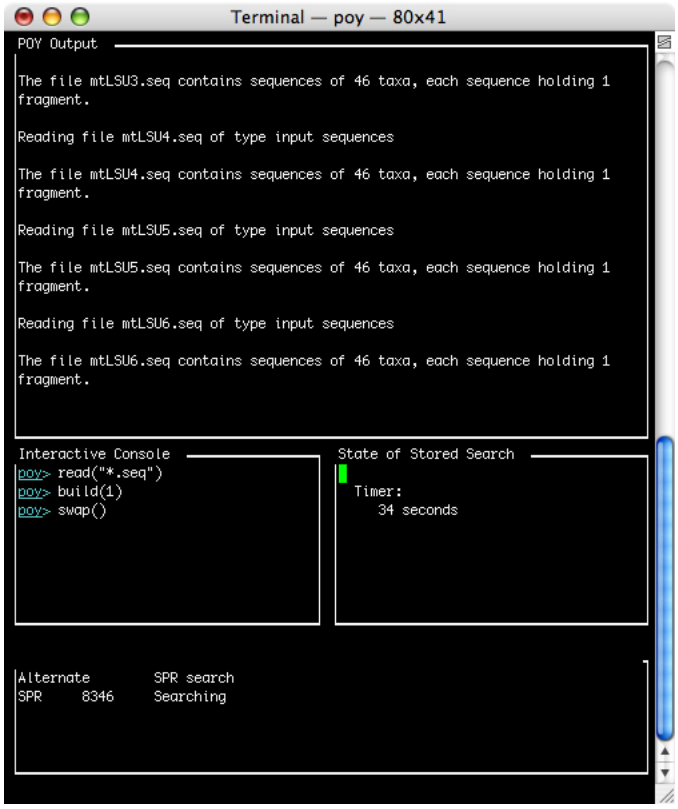


Figure 2.17: POY4 interactive console during a process. The *POY Output* window displays (by default) the information on the input datafiles. The *Interactive Console* lists the commands that have been consecutively executed. The *Current Job* window shows the state of the current operation and the current tree score. The *State of Stored Search* shows the time elapsed since the last command, `swap`, was initiated.

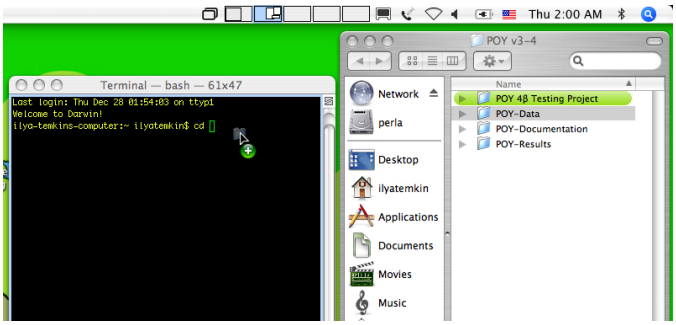


Figure 2.18: Specifying the location of datafiles. The folder `POY-Data` is dragged from the `POY v3-4` folder directly in the Terminal window.

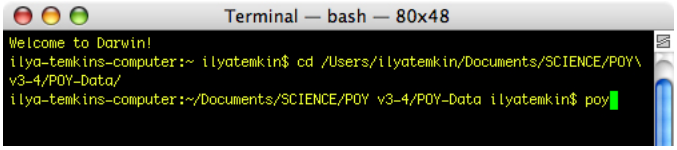


Figure 2.19: Starting `POY4`. At the folder containing datafiles, entering `poy` starts a `POY4` session.

the commands previously entered during the session. In addition, the interactive console is equipped with the autocomplete feature: it involves `POY4` 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 speeds up interaction with the program.

2.4.4 Browsing the output

As more 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)).

2.4.5 Switching between the windows

To return to the *Interactive Console*, start typing and the cursor will automatically be placed back at the `POY4` 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*.

2.4.6 Importing data

The basic command to input data in `POY4` 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 datafiles, `morpho.ss` and `28s.fas`, respectively. In the*Interactive Console* files can also be entered by dragging them into the input window and placing them after a given command which will provide both the correct path and filename. We can issue a pair of `read()` commands (Figure 2.20):

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

The syntax of `read`, like every command in `POY4`, contains two elements: the name of the command (e.g. `read`), followed by an optional list of arguments separated by commas and enclosed in parentheses. Typically, the arguments of the command `read()` are names of datafiles, 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. `read()`) always follow the command name. An exhaustive discussion of `POY4` 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 using the command`cd`; for example

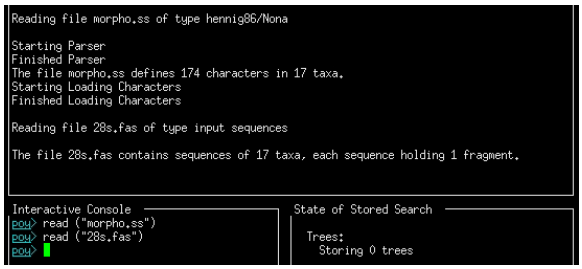


Figure 2.20: Importing datafiles using the *Interactive Console*. Two consecutive `read` commands specify both the morphological datafile in Hennig86 format (`morpho.ss`), and the molecular datafile in FASTA format (`28s.fas`). Note that `POY4` automatically reports in the *POY Output* window the names and types of files that have been imported.

`cd ("/Users/username/docs/poyfiles")`. Alternatively, the full path can be included in the argument of `read`: `read("/Users/username/docs/28s.fas")`.

Most of the time users are interested in importing multiple datafiles to analyze an entire dataset. In this case, multiple datafiles 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")`. This is equivalent to sequentially importing each file as shown above (Figures 2.20 and 2.21).

Figure 2.20 also illustrates an important feature that makes `POY4` different from many other phylogenetic analysis programs: every time a file is imported during a `POY4` session, the input data are *added* to the current 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 2.21):

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

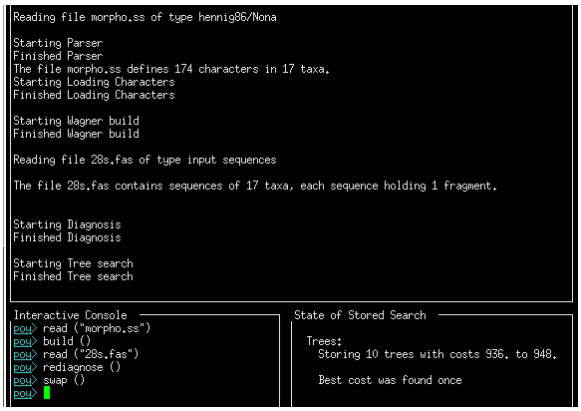


Figure 2.21: Building trees with morphological data only but continuing analysis using combined morphological and molecular data. This example shows how we can add data to the analysis incrementally by loading files at different points in the search. First, the morphological data are imported from `morpho.ss` file using `read()` the and trees are built based on these data. Then molecular data from the `28s.fas` file are loaded into memory in addition to previously imported morphological data. Finally, subsequent analyses, `rediagnose()` and `swap()`, are conducted using the data in memory, that is the trees based on morphological data, and both morphological and molecular character sets.

It must be noted that if the numbers of terminals differ among datafiles, only the data that correspond to the terminals used to generate the trees (from the morphological datafile in our example) 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 `POY4` appends trees and data in memory, it is a good practice when starting a new analysis to empty the memory using use 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 `POY4` and other types of input files see `help(read)`.)

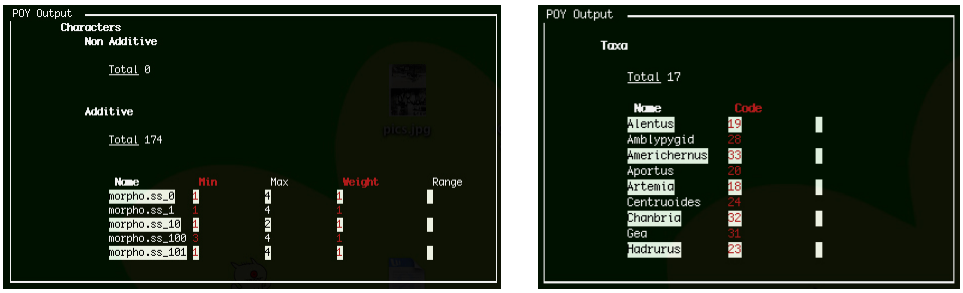


Figure 2.22: Inspecting imported data. The figure shows segments of a data report generated by `report(data)`. The left and right panels demonstrate a typical table output the character and terminal data respectively.

2.4.7 Inspecting data

Once a dataset (or multiple datasets) is imported, `POY4` automatically reports a brief description of contents for each loaded file in the *POY Output* (Figure 2.20). 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 to 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 datafiles 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 2.22. 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.

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 `select` (Section 3.3.23)).

By default, `POY4` reports the results of executed commands to the *POY Output* window. However, the same output can be redirected to a file simply

Figure 2.23: 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 datafile. In this example, the data for all the the taxa listed in the *POY Output* window are present in both datafiles, `morpho.ss` and `28s.fas`.

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`). For instance, if we would like to output into the file "data_analyzed.txt," we would enter `report("data_analyzed.txt", data)`.

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_references)
```

A typical output of `cross_references` command is shown in Figure 2.23.

2.4.8 Building the initial trees

The command to build trees is `build()` (already mentioned in Section 2.4.6). After importing `morpho.ss` and `28s.fas`, executing the command `build()`

without specifying any arguments generates 10 trees by random addition sequence (the default setting of the command).

Many `POY4` commands operate under default settings when executed without arguments. To learn what the default settings are for a particular command, use either `help()` command with the command name of interest inserted in parentheses or consult the *POY Commands* chapter (3.1.1).

If the user would like to specify a number of tree building replicates different from the default value of 10, an 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 2.24). 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 2.24).

2.4.9 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()
```

Figure 2.24: 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`), a 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`).

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 2.25). 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 of the best cost (Figure 2.25). Note 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 allows designing a large number of search strategies of 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 evalu-

Figure 2.25: 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 cost (`9`) recovered from independent tree builds. Note these trees may not necessarily have unique topologies.

ated, timeout after certain number of seconds, transform the cost regime, and other functions in conjunction with many `POY4` commands.

2.4.10 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 local-optima trees in memory. Most of the time, a user would like to select only those trees that are 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 2.26).

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

Comparing the output reported in the *State of Stored Search* before (Figure 2.25) and after (Figure 2.26) executing `select()` shows that swapping

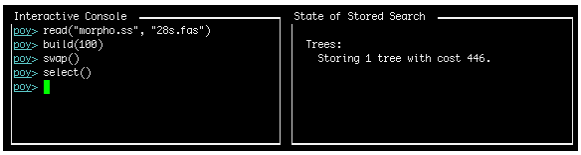


Figure 2.26: Selecting unique best trees. Executing `select()` keeps only unique trees of best cost. The *State of Stored Search* window reports that there is only one unique tree of best cost (`446`).

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()`.

2.4.11 Visualizing the results

There are several ways to visualize results. The command `report("my_first_tree", graphtrees)` outputs a cladogram in postscript format (Figure 2.27), which can be displayed, edited, and printed using graphics software (such as Adobe Illustrator or Corel Draw). `POY4` also appends the "ps" 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 2.27). The ascii tree can also be reported to a file, if the output file name is specified (in parentheses and separated from the argument `asciitrees` by a comma).

`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. Other 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)`).

2.4.12 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: (1) `read(''a'') <ENTER>` and (2) `read(''a'') <ENTER> read(''b'') read(''c'') <ENTER> <CONTROL-C>`.

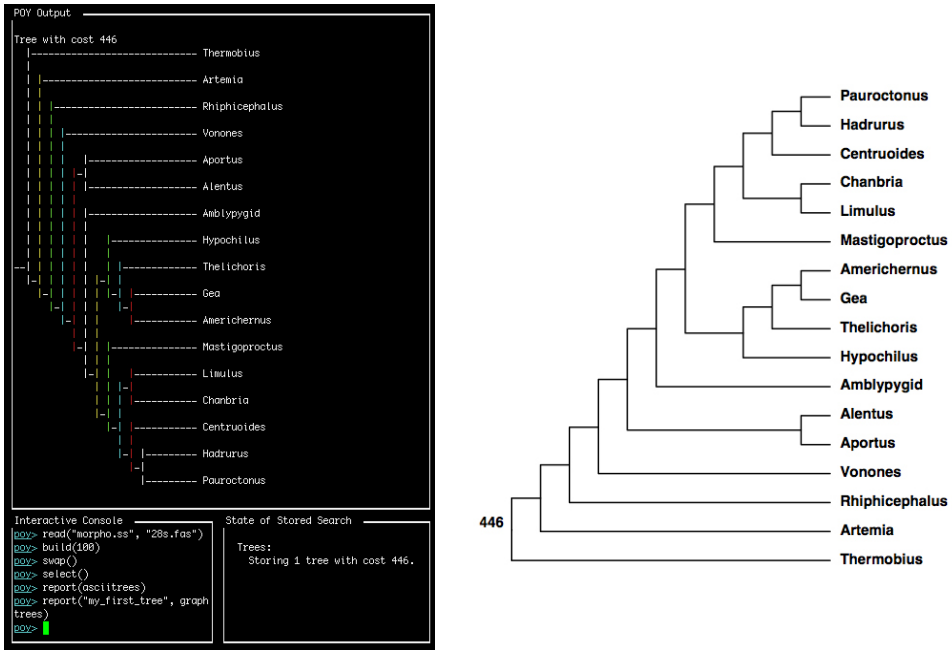


Figure 2.27: Visualizing trees. An ascii tree (left) is generated using the command `report(asciitrees)`. The same tree is reported to a file in a postscript format (right) using `report("my_first_tree", graphtrees)`. Note that both representations of trees are preceded by their costs.

2.4.13 Reporting errors

If there is an error pertaining to wrong syntax (such as a typo in a command name), POY4 will indicate the location of the error by underlining the problematic part of the input with "^" in the *Interactive Console* (Figure 2.28). 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 2.28).

2.4.14 Exiting

To finish a POY4 session, enter the command `exit()` (Figure 2.29) or `quit()`. This will close the POY4 interface and resume the Terminal window (Mac OSX) or the Command Prompt window (Windows).

2.5 Creating and running POY4 scripts

So far, we have communicated with POY4 interactively through the *Graphical User Interface* or by executing commands from the *Interactive Console*. Another way of conducting an analysis is to run a *script*, a simple-text file containing a list of commands to be performed (Figure 2.30).

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 POY4 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 POY4 but can be helpful to describe the contents of the files and provide other annotations. The comments are enclosed in parenthesis *and* asterisks. For example, `(* this is a comment *)`. Comments can be of any length and span multiple lines. Comments can also be entered interactively from the *Interactive Console*.

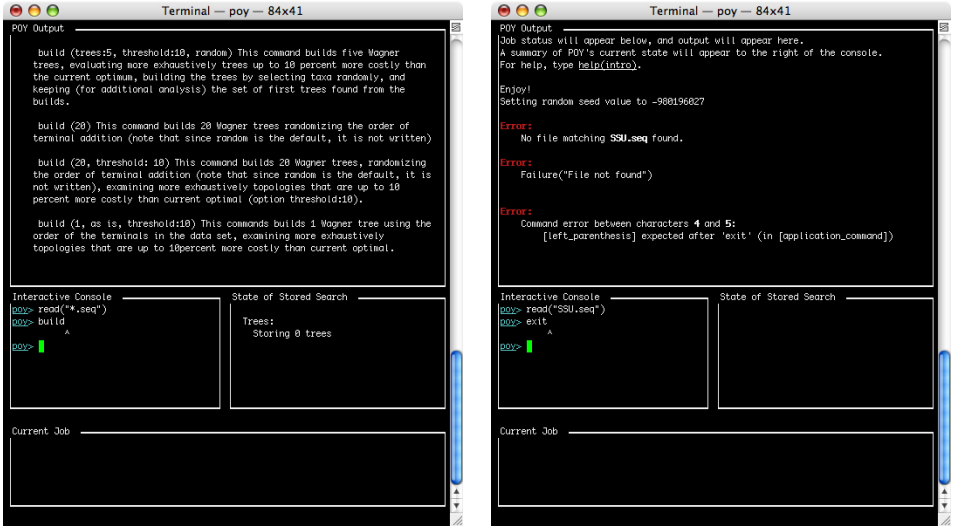
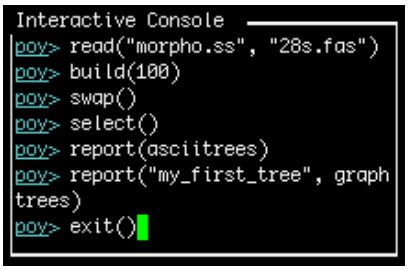


Figure 2.28: Displaying errors. POY4 displays error messages in several ways. In the example in the left panel, the command `build` was entered without parentheses, which is required for a valid POY4 command syntax; the exact place of the error is marked by "^", in this case following the `build` commands. 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 datafile `SSU.seq` is not present, which could have been caused either by a mistake in the name of the file, missing file, or the location of the file in a directory, other than the one specified prior to starting the POY4 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 2.29: Exiting POY4




```
(* Basic POY Search *)

read("morpho.ss", "28s.fas")
build(100) swap()
select()
report(asciitrees, "my_first_tree", graphtrees)

exit()
```

Figure 2.30: Using POY4 scripts. The list of commands executed interactively using the *Interactive Console* (left) and a script containing the same list of commands (right). Note, that the header of the script is a comment, enclosed in "(* *)", that is ignored by POY4. 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.)

Obviously, using scripts requires the user to design the workflow of the process prior to conducting the analysis. POY4 scripts can be created and saved using the *Script Editor* window of POY4 interface or any conventional text editor (such as TextPad, TextWrangler, BBEdit, Emacs, or NotePad).

POY4 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.

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.

It it critical to include the command `exit()` at the end of the script. Otherwise POY4 will be waiting for further instructions to be entered after executing the script's contents.

2.6 Obtaining help

Instructions to run POY4, command descriptions, and the theory behind POY4 can be obtained from a variety of sources.

POY4.0 Program Documentation (this manual) is a comprehensive and detailed manual on all the aspects of using `POY4`, from installation to output and visualization of results. Included are *Quick Start*, `POY4` command reference, practical guides and tutorials that make the program immediately accessible for beginners and provide in-depth information for experienced users. The documentation in PDF format can be accessed from the *Help* menu of the graphical user interface or downloaded separately from `POY4` web site at

`http://research.amnh.org/scicomp/projects/poy.php`

POY interactive help can be obtained by entering `help()` at the `POY4` 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 2.29.)

POY4 Mail Group is an Internet-based forum for discussing all issues related to `POY4` and provides the best way to communicate with `POY4` developers on specific issues (see *WWW resources* below). The website is located at `http://groups.google.com/group/poy4`.

POY Book (Wheeler et al., 2006 *Dynamic Homology and Phylogenetic Systematics: A Unified Approach Using POY [32]*) provides a review of the theory behind `POY4`, and contains formal descriptions of many algorithms implemented in the program and the descriptions of commands of the earlier version, `POY3`.

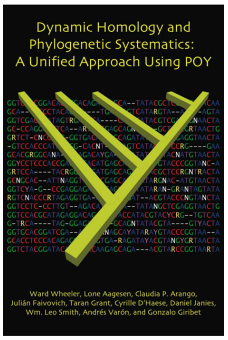


Figure 2.31: The POY Book.

2.7 WWW resources

POY4 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 and check regularly for updates. There are several Internet-based resources that offer this information.

POY4 Web Site has downloadable compressed files of POY4 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://research.amnh.org/scicomp/projects/poy.php`

POY4 source code repository contains has downloadable POY4 source code. The site is powered by Google at

`http://code.google.com/p/poy4/source`

POY4 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 POY4 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 POY4. 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 POY4 developers. The *POY4 Mail Group* is hosted by Google at

`http://groups.google.com/group/poy4`

Chapter 3

POY4 Commands

3.1 POY4 command structure

3.1.1 Brief description

POY4 interprets and executes *scripts* issued by the end user. These can come from 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 POY4 has default values.

In POY4, we recognize four types of command arguments: *primitive values*, *labeled values*, *commands*, and *lists of arguments*.

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

Labeled values are a lowercase identifier (which are referred to as *label*), and an argument, separated by the colon character. ":".

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 compositive 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, POY4 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 weighted) data as an argument of the command `perturb`. Therefore, to take the maximum advantage of POY4 functionality, it is essential to get acquainted with the grammar of POY4.

3.1.2 Grammar specification

The following is the formal specification of the valid grammar of a script in POY4:

```
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 POY4 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.

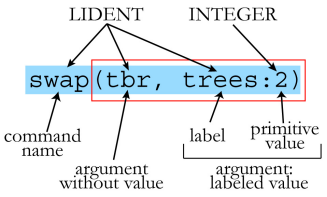


Figure 3.1: The structure of a simple POY4 command. The entire command (highlighted in blue), consists of a command name followed by a list of arguments (enclosed in red box). See text for details.

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 argu-

ment `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 3.2: A structure of a compound `POY4` 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 in 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. In the descriptions of `POY4` commands below, the elements of `POY4` grammar are defined in the text using the following conventions:

- A command that could be included in a `POY4` script (that is can be entered in the interactive console or included in an input file) is shown in `terminal type`.
- 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.
- A brief description of the command's function.
- Cross references to related commands.

- The valid syntax for the command.
- The list of descriptions of valid arguments.
- Description of default settings.
- Examples of the command's usage.

NOTE

Default syntax. The default syntax for all commands is the same: it includes the command name followed by empty parentheses. For example, `swap()`. The descriptions of default settings, however, include the entire argument list for the obvious reason of showing what is included in the omitted argument list.

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.

NOTE

Certain command arguments are mainly useful to `POY4` developers, and those arguments are preceded by an underscore.

3.3 Command reference

3.3.1 build

Syntax

build`([argument list])`

Description

Builds Wagner trees [7]. Building multiple trees with a randomized addition of terminals allows for the evaluation of many more 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 singe tree building procedure. During tree building, `POY4` reports in the *Current Job* window of the ncurses interface which of the terminal addition strategies is currently used.

By default `POY4` 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 (for example, if computer memory were limited) 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 going to be 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

`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 datafiles, and all others are built using a random addition sequence.

`branch_and_bound[:FLOAT ]` Calculates the exact solution using the Branch and Bound algorithm [11]. 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 bound (either tree cost or likelihood score).

`constraint[:STRING ]` Builds trees using the set of constraints specified by the 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.

`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.

`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.

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

`of_file:STRING` Imports 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.

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

`all` Turns off all preference strategies for adding branches and simply tries all possible addition positions for all terminals.

Defaults

`build(trees:10, randomized)` By default, `POY4` 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 that 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(15, as_is)`
  Builds the first Wagner tree using the order of terminals in the first imported datafile 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.

3.3.2 calculate_support

Syntax

calculate_support`([argument list])`

Description

Calculates the requested support values. `POY4` implements support estimation based on resampling methods (Jackknife [4] and Bootstrap [8]) and Bremer support [2, 12]. The Jackknife and Bootstrap support values are computed as frequencies of clades recovered in strict consensus trees built in each resampling iteration. The consensus trees are based on best trees recovered in each replicate with zero-length branches collapsed. All the arguments of `calculate_support` command are optional and their order is arbitrary. For examples of scripts implementing support measures see tutorials `4.3` and `4.4`.

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 the Hennig86 format can be generated based on an implied alignment using `phastwinclad` (Section 3.3.19) that can subsequently be analyzed using other programs.
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.

Arguments

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

`bremer` Calculates Bremer support [2, 12] 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

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.

`bootstrap[:INTEGER ]` Calculates Bootstrap support [8]. The integer value specifies the number of resampling iterations (pseudoreplicates). If the value is omitted, 5 pseudoreplicates are performed by default.

`jackknife[:([argument list])]` Calculates Jackknife support [4] using the sampling parameters specified by the arguments. The arguments of `jackknife` are optional and their order is arbitrary. If both values are omitted, the default values of each argument is used.

- `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, it specifies the number of Wagner tree builds per pseudoreplicate. Single best trees from all psudoreplicates 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 searching 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 `POY4` will calculate the bremer support for each

tree in memory node by node. However, if no trees 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:(resample:1000), build(), swap(tbr, trees:5))`
  Calculates Jackknife support values randomly removing 36 percent of the characters (the default of `jackknife`), building 10 Wagner trees by random addition sequence (the default of `build`), swapping these trees using `tbr`, and keeping up to 5 minimum-cost tree in the final swap per swap (totaling up to 50 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 `POY4` 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 datafiles are contained in different directories. It also eliminates the need to navigate into the working directory before beginning a POY4 session. 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 in a Mac environment (when using a PC the path will the forward slashes will be replaced with backslash characters is the path).
  /Users/username/docs/poyfiles.

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 POY4 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 `POY4` session. This command does not accept any argument. `exit` is equivalent to the command `quit`.

NOTE

To interrupt a process without quitting a `POY4` 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 entering new command 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 [9] on the trees in memory. Tree Fusing method to escape local optima by exchanging clades with identical composition of terminals between pairs of trees. Only *one* pair of trees is evaluated during a single iteration. The size of the clades being exchanged is not specified.

Arguments

`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.

`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`).

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

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

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

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 `POY4` 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 starting trees. During the 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. These trees are subjected to swapping under the default settings of `swap`. The entire procedure is repeated nine more times.

- `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 `POY4` 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 POY4 file produced by the command save (Section 3.3.21). If the description was not specified by the user, inspect reports that the description is not available. If the file is not a proper POY4 file format, a message is printed in the *POY Output* window of the ncurses interface or to the standard error of the flat interface.

POY4 files are not intended for permanent storage. They are recommended for temporary storage of a POY4 session, checkpointing the current state of the search (to avoid losing data in case the computer or the program fails), or reporting bugs. POY4 also automatically generates POY4 files in cases of terminating errors (important exceptions are out-of-memory errors).

Examples

- `inspect("initial_search.poy")`
  Prints the description of the POY4 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 `POY4` 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 `POY4` session will be replaced with the contents of the `POY4` file. If the file is not in proper `POY4` 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 `POY4` file and its usage.

`POY4` files are not intended for permanent storage: they are recommended for temporary storage of a `POY4` session, checkpointing the current state of the search (to avoid losing data in case the computer or the program fails), or reporting bugs. `POY4` also automatically generates `POY4` 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 `POY4` file `initial_search.poy`, located in the current working directory.

- `load("/Users/andres/test/initial.poy")`
  Reads and imports the contents of the `POY4` file `initial.poy` in the absolute path described by the argument.

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 a 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 for 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 [16] by reweighting characters listed in `report(data)`. For unaligned data,`ratchet` randomly selects and reweighs 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 [16].

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, LIDENT)` Resamples the data (characters or terminals) in random order with replacement. The `resample` string consists of an integer value specifying the number of items to be resampled (followed by a comma) and a lident value specifying whether characters or terminals (values `characters` and `terminals`, respectively) are to be resampled. Specifying both values is required. 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, swap (trees:1))` When no arguments specified, `POY4` performs the ratchet procedure under default settings.

Examples

- `perturb(resample:(50,terminals), iterations:10)`
  Performs 10 successive repetitions of random resampling of 50 percents of terminals 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 (*i.e.* molecular sequence data) 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 `POY4` started.

Examples

- `pwd()`
  This command generates the following message: "The current working directory is /Users/myname/datafiles/". The actual reported directory will vary depending on the directory of the shell when `POY4` 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 `POY4` session. This command does not have any arguments `quit` is equivalent to the command `exit`.

NOTE

To interrupt a process without quitting a `POY4` 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 entering 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 are ASN1, Clustal, FASTA, GBSeq, Genbank, Hennig86, Newick, NewSeq, Nexus, PHYLIP, POY3, TinySeq, and XML. Filenames should be enclosed in quotes and, if multiple filenames are specified, they must be separated by commas. All filenames read into `POY4` should include the appropriate suffix (*e.g.* .fas, .ss, .aln). `read` automatically detects the type of the input file. `read` can use wildcard expressions (such as *) to refer to multiple files in a single step. For example, `read("biv*")` imports all data files the names of which start with `biv` or `read("*.ss")` imports all files with the extension `.ss` (given that the data files are in the current directory). Specifying filename(s) is obligatory: an empty argument string, `read()`, results in no data being read by `POY4`. 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, `POY4` issues an error message but this will not interfere with subsequent file loading and execution of commands.

`POY4` 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).

NOTE

Although `POY4` 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, `POY4` 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`).
The terminal names, as well as input file names, must not contain spaces, at or percentage symbols.

NOTE

Unlike many phylogenetic programs, `POY4` 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 that appearing in it.
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 molecular file contains a space, " ", `POY4` issues a warning. This also occurs if a taxon name appears to match a nucleotide sequence.
If one of the terminal names in an imported molecular file contains an "at" or a percentage symbols, the file will not be loaded because it may cause the program to crash when reporting results.

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.

`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, `POY4` 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.

NOTE

`POY4` recognizes the characters x and n as representing any nucleotide base (a,c,t, or g). 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 recognized 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. Because additive characters are integers, such characters need to be re-scaled using the `weightfactor` of the `transform()`. 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]`. To scale the values, a transformation is applied to the character `winglength` as follows: `transform((characters,names:("winglength")), (weightfactor:0.01))`.

`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 aminoacids are *not* supported and inputing files that contain aminoacid data with ambiguities results in an error message.

`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 argument `dynamic_pam` (Section 3.3.27) of the command `transform` (Section 3.3.27). Locus homologies are determined dynamically, but based on annotated regions [22] (for a sample script using this data type see tutorial `4.7`.

`breakinv:(STRING, STRING, [orientation:BOOL, init3D:BOOL ])` An enhancement of the data file type `custom_alphabet` allowing rearrangement events specified using `dynamic_pam()`. Syntactically, breakinv data type is identical to custom_alphabet data type.

`chromosome:(STRING list)` Specifies that the data in the files listed in the string argument are chromosomal sequences without predefined locus boundaries. 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 argument `dynamic_pam` in the command `transform` (Section 3.3.27). Unlike when using `annotated` data type, both locus-level and nucleotide-level homologies are determined dynamically [21] (see tutorial `4.6`). If chromosome sequences were imported as nucleotide type data, they can be converted to chromosome type data using the argument `seq_to_chrom` of `transform` (Section 3.3.27).

`custom_alphabet:(STRING, STRING, [orientation:BOOL, init3D:BOOL ])` Reads the data in the user-defined alphabet format. The first string argument is the name of a datafile that contains custom-alphabet sequences in FASTA format. The characters can be (but are not required to be) separated by spaces.

The second string argument is the name of a custom-alphabet input file 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 impotant that the cost matrix must 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`.

An example of a corresponding input file:

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

The optional arguments of `custom_alphabet` include `orientation` and `init3D`, both of which require obligatory boolean values. The argument `orientation` allows the user to specify the orientation of custom-defined alphabet characters. The *tilde* symbol ("∼") preceding an alphabet character indicates the negative orientation. The options are `orientation:true` or `orientation:false`. The default option is `true`.

The argument `init3D` indicates that if program will calculate in advance the medians for all triplets of characters (a, b, c). The options are `init3D:true` or `init3D:false`. The default option is `true`.

`custom_alphabet` can be transformed into `breakinv` using `transform`.

`genome:(STRING list)` Specifies that the data listed in the string argument are multichromosomal nucleotide sequences with the "@" sign separating individual chromosomes. This data type allows for chromosome-level rearrangements specified by the argument `dynamic_pam` (Section 3.3.27) of the command `transform` (Section 3.3.27). Chromosome homologies are determined dynamically using distance threshold levels specified by the argument `chrom_hom` (Section 3.3.27) of `transform` (Section 3.3.27)(for a sample script using this data type see tutorial `4.9`.

`nucleotides:(STRING list)` Specifies that the data in the list of files hold nucleotide sequences in FASTA format.

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:(read argument, tcm:STRING)` Specifies that the input sequences are prealigned and should be assigned the transformation cost matrix from the input file defined by the string argument. (See the argument `tcm` (Section 3.3.27) of the command `transform`.)

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

Defaults

`read()` If no data files are specified, `POY4` does nothing. If however, data files are listed but character type is not indicated, `POY4` 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 the names of which start with `SSU`, and all the files with the extension `.txt`. The types of the datafiles 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`.

- `read(custom_alphabet:("my_data", "alphabet"))`
  Reads the first file, `my_data`, containing data in the format of a custom alphabet, which is defined in the second input file, `alphabet`. By default, the forward and reverse orientation (`orientation:true`) of custom-alphabet characters is considered and prior calculation of medians for their triplets (`init3D:true`) is performed.

- `read(annotated:("filea.txt", "fileb.txt"),`
  `chromosome:("filec.txt"))`
  Reads three files containing chromosome-type sequence data. The sequences in two files, `filea.txt` and `fileb.txt`, contain pipes (" | ") separating individual loci, whereas the sequences in the third, are without predefined boundaries.

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

- `read(prealigned:("18s.aln", tcm:(1,2)))`
  Reads the prealigned data file `18s.aln` 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 rediagnose

Syntax

rediagnose`()`

Description

Performs a re-optimization of the trees currently in memory. This function is only useful for sanity checks of the consistency of the data. Its main usage is for the `POY4` developers. This command does not have arguments.

Examples

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

3.3.16 recover

Syntax

recover`()`

Description

Recovers the best trees found during swapping, even if the swap was cancelled. This command functions only if the argument `recover` (Section 3.3.26) was included in a previously executed (in the current `POY4` 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 `POY4` 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 `POY4` 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)
- `recover` (Section 3.3.26)

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 datafiles. 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 datafiles but avoid reporting these names on the trees; (2) to provide a single name for a terminal in cases where the corresponding data is stored in different files under different terminal names; and (3) to simply 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.

NOTE

In order to change taxon names, the command `rename` must be executed *before* importing the datafiles (see command `read` (Section 3.3.14)) that contain character data the taxa 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).

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 to specify the items to be renamed either individually (by using a pair of string arguments) or in a group (by importing a *synonymy* file. The latter 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 is going to be substituted for all the subsequently listed names. Consider, for example, a two-line synonymy file below:

```
alpha beta gamma
delta 1
```

When this file is imported, the items `beta` and `gamma` will be renamed as `alpha` and the item `1` will be renamed as `delta` in all subsequently imported datafiles.

`(STRING, STRING)` Specifies the names of individual items to be renamed. The first item is renamed as the second item: specifying `("alpha", "beta")` renames the character or taxon `alpha` to `beta`. To specify multiple pairwise name substitution, several name pairs can be listed: `("alpha","beta"),("gamma","delta").`

NOTE

Note that when `rename` is applied by specifying pairs of synonyms in the command's argument, the substitute name is listed *second*. However, the substitute name appears *first* in a synonymy file, 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 datafiles.

- `rename(terminals,("Mytilus_sp","Mytilus_edulis"))`
  This command renames terminal file `Mytilus_sp` as `Mytilus_edulis` in all subsequently imported datafiles.

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.

Arguments

Reporting to files

`STRING` Specifies the name of the file to which all the specific 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.

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

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

`compare:(BOOL, identifiers, identifiers)` If the boolean argument is set to `false`, the command reports the ratios of all pairwise distances to their maximum length for the characters specified by character identifiers. If the boolean argument is set to `true`, the complement sequences for the characters specified by the second identifier are computed prior to reporting the distance.

`cross_references[:identifiers[:STRING]]` Reports a table with terminals being analyzed 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. `cross_references` is a very useful tool for visual representation of missing data.

Under default settings, cross-references are reported for all imported datafiles. To report cross-references for some of the fragments within a given file, a single character, or a subset of characters, optional arguments 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 datafiles to be cross-referenced. For example, if a command `cross_references:names:("file1")` is executed, the output is produced only for `file1`.

The argument `cross_references:all` generates a table that shows presence and absence of fragments contained within each file. If each datafile 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.

`data` Outputs a summary of the input data. More specifically, `POY4` will report the number of terminals to be analyzed, a list of included terminals with numerical identification numbers, list of synonyms (if specified), a list of excluded terminals, a 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.

`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.

`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 per cost.

`treescosts` Reports the cost of each tree separated by colons. The output contains no formatting for easy processing by any scripting language.

Trees This set of arguments outputs tree representations in parenthetical, ascii (simple text), or postscript formats. The arguments specify the types of tree outputs. They include actual trees resulting from current searches, or 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[:BOOL ]]` Draws ascii character representations of trees stored in memory. The argument `collapse` collapses the zero length branches if the boolean value is `true` (the default); if the boolean value is `false`, the zero length branches are not collapsed.

`clades` 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 [18]; if integer value is specified, a majority rule consensus is computed, collapsing nodes with occurrence frequencies less than the specified integer [15]. If a value less than `51` is specified, `POY4` reports an error.

`graphconsensus[:INTEGER ]` Same as `consensus` except for consensus trees are reported in graphical format, either in the ascii format on screen or in the postscript format if redirected to a file.

`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 postscript format. The argument values are the same as for `supports` (*i.e.* `bremer`, `jackknife`, and `bootstrap`).

`graphtrees[:collapse[:BOOL ]]` If `POY4` has been compiled with graphics support, it will display a window in which you can browse graphical representations of all the trees in memory. When working in this window, using "j" and "k" keys displays the previous or next tree respectively. If no graphical support available, the output is similar to that generated by the `asciitrees` argument. Pressing "q" key returns to the *Interactive Console* window. The argument `collapse` will collapse the zero length branches if true, otherwise not (default is `true`.)

`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.

`bremer` accepts an optional string argument (as in `report(supports: bremer:"file.txt")`, which specifies a file containing a list of trees and costs (as those generated by `visited` (Section 3.3.26)), which should be used with their annotated cost to assign the bremer support values. If no input file is given, or if `bootstrap` or `jackknife` are requested, then the necessary information must have been calculated using `calculate_support` (Section 3.3.2).

`jackknife` and `bootstrap` accept an optional argument with two possible values: `individual` or `consensus`. `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. 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:

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

`_cost` Include the cost in square brackets for every subtree in the tree. (These are *not* branch lengths.)

`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.

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

NOTE

The `hennig` and `newick` arguments are mutually exclusive.

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

`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.

`collapse[:BOOL ]` If `true`, zero length branches are collapsed (the default), but if `false`–no branches are collapsed.

Implied alignments This set of arguments outputs implied alignments [29].

`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.

`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 a name of 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`.

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

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

`phastwinclad` Produces a file in the 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 `4.2`):

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

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`.

diagnosis This set of arguments 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.

Other arguments

`ci` Calculates the ensemble consistency index (CI [6, 14]) for additive, nonadditive, and Sankoff 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; [6]) for additive, nonadditive, and Sankoff 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).

`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, `POY4` 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 `POY4` 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 `POY4` 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 POY4 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, POY4 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("my_results")`
  This commands outputs the data, trees, and diagnosis (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, file names, and other relevant data.

- `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(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`. Because 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 datafiles.

- `report(trees)`
  This command displays on screen the trees in memory in parenthetical notation with zero-length branches collapsed and terminals separated by spaces.

- `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("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 postcript format to the file `filename.ps`.

- `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` the graphical representation of these trees in postscript 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("my_results", data, diagnosis, consensus, consensus:75, "consensus", graphconsensus)`
  This command reports the requested types of outputs (*i.e.* reports on

the data, diagnosis, and strict consensus and 75 percent majority-rule consensus trees in parenthetical notation) to the file `my_results`. In also outputs a strict consensus tree to the file `consensus`.

- `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` only the tree(s) with bremer support values.

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

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

- `report("scipt1_analysis", script_analysis:"/users/datafiles/script1.poy")`
  This command produces the file `scipt1_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 `POY4` session.

- `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 POY4 script file or files. 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 take long time or many scripts need to be executed automatically, for example, when conducting sensitivity analysis[23]. There are no default settings of **run**.

NOTE

Note that if any of the scripts contain the commands **exit()** or **quit()**, POY4 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 POY4 command scripts contained in the files **script1** and **script2** in the same order as they are listed in the list of arguments of **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 POY4 state of the program to a file (POY4 file). The first, obligatory string argument specified the name of the POY4 file. The second, optional string argument specifies a string included in the POY4 file, that can be retrieved using the command `inspect` (Section 3.3.9).

POY4 files are not intended for permanent storage: they are recommended for temporary storing of a POY4 session by a user, checkpointing the current state of a search to avoid loss work in case the computer or the program itself fails, or to report bugs. POY4 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 POY4; 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", "My total evidence data")`
  This command performs the same operation as described in the example above, but, in addition, it includes the string `My total evidence data` with the file `alldata.poy`, which can later be retrieved using the command `inspect` (Section 3.3.9).

- `save("/Users/andres/test/alldata.poy", "My 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/test/` 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, swaping 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 the `search` command are included in the set of trees available for the `fuse` but are not swapped.

Arguments

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

`max_time:FLOAT:FLOAT:FLOAT` 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. `POY4` *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 `POY4`, preventing the program from overflowing the

memory. Furthermore, it is important to note that when running POY4 in parallel the maximum amount of memory specified by the user is allocated to each processor being used. Under certain circumstances, however, it might be required to use more memory to avoid program failures.

`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, POY4 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.

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. *If the user does not specify the value of `max_time`, the search will be aborted after one hour.*

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 ammong all the processes is desirable to improve the potential of tree fusing.
To guarantee that separate processes exchange trees and seed each other's fuse step, it is advisable to use few successive search commands when executing the program in parallel, to ensure proper tree exchange between different processes, increasing their probability of finding better trees during the fuse step.

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)
  search(max_time:0:6:0)
  search(max_time:0:4:0)
  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.

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

Select terminals and characters 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).

`terminals` Specifies that the subsequently listed identifiers refer to *terminals* to be selected. By default, `POY4` 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 `POY4` session, use the commands `store` (Section 3.3.25) and `use` (Section 3.3.28).

`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.

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.

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

`codes:(INTEGER list)` Specifies the codes of characters or terminals. The codes are unique numbers that are generated by `POY4` 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 effectively be used only within a current `POY4` session.

`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 sets the maximum percentage of missing data. Terminals or characters that have *more* missing data than defined by the value are included in the analysis.

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

NOTE

For dynamic homology characters, the missing data refers 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) includes 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`.

`static` Specifies the static homology characters.

`dynamic` Specifies the dynamic homology characters.

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

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

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

`optimal` Selects all trees of minimum cost.

`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).

NOTE

There is no special command in `POY4` to clear trees from memory. However, selecting zero best trees using the command `select(best:0)` effectively removes all trees currently stored in memory.

`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.

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

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

Defaults

`select(unique, optimal)` By default `POY4` selects all unique trees of optimal (best) cost. The rest of the trees are removed 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, missing:50)`
  This command excludes from subsequent analyses all the terminals that have more than 50 percent of characters missing. The list of included and excluded terminals is automatically reported on screen.

- `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 `POY4`. This command performs diverse auxiliary functions, from setting the seed of the random number generator to selecting a terminal for rooting output trees.

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

Arguments

Application settings Some generic application settings. Have no effect in the analyses themselves.

`history:INTEGER` Sets the size of the `POY4` 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, `POY4` will append the text to it; if the log file does not exist, then `POY4` 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`.

`nolog` Stops outputting the log to any previously selected file. See the description of the argument `log` above.

`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 `POY4` at the beginning of each `POY4` session (for example, `set(root:45)`. The codes can be obtained using the command `report(data)`). The terminal codes, however, are unique only within a current session.

`timer:INTEGER` Specifies the lapse of time in seconds that should have 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 routines and are applied to all character types. The arguments are mutually exclusive: only the argument of `set` specified last is used.

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

`exhaustive_do` Applies a standard Direct Optimization algorithm for the tree cost estimation [24, 28]. The difference with `normal_do` is that the calculation of the tree costs during a search is much more intense, always looking for the best possible alignment for every single topology (instead of a lazy and greedy strategy used by the `normal_do`).

`iterative` Applies the Iterative Pass optimization for the tree cost calculations [30]. This improves the tree cost estimation but at the expense of a tremendous execution time. The iterative argument can be applied to all the dynamic homology character types including chromosome, genome, and custom_alphabet characters. When implementing the iterative pass option for these complex character types the parameter `max_3d_len` can be used within the command `transform` ((all, `dynamic_pam`:(`max_3d_len: INTEGER`))) to reduce execution time. Another hueristic strategy is to implement `iterative` at the very end of an analysis to polish the final set of trees and perform a final search.

NOTE

Due to the complexity of heuristics of the iterative pass optimization, there is no guarantee that the tree cost recovered from the search would 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 the same tree (see `rediagnose` (Section 3.3.15)), but will recover the same tree cost in subsequent rediagnoses.

Randomized routines

`seed:INTEGER` Sets the seed for the random number generator using the integer value. If unspecified, `POY4` uses the system's time as seed.

NOTE

It is critical to set a seed value to insure reproducibility of the results of the analyses that require randomization routines (such as tree building).

Defaults

`set(history:1000, normal_do)` Under default settings the size of the history buffer is limited to 1,000 lines, the Direct Optimization is used for tree cost calculation, and the current time is used to specify the seed.

Examples

- `set(history:1500, seed:45, log:"mylog.txt")`
  This command increases the size of the history in the ncurses interface to 1,500 lines, sets 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 terminal `Mytilus_edulis` as a root for the output trees.

See also

- `report` (Section 3.3.19)

3.3.25 store

Syntax

store`(STRING )`

Description

Stores the current state of `POY4` 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 `POY4` session.

Arguments

`STRING` Specifies the name of the stored search state of the current `POY4` 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 `POY4`. 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 if `POY4` 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 tune the search heuristics, but the arguments within each group of are mutually exclusive. (If more than one arguments of one group is listed, only the last one is executed.)

Arguments

Neighborhood A neighborhood is a subset of topologies reachable from a given one by a given search method. The basic standard procedures for local search in phylogenetic analysis are SPR and TBR [20]. The nearest-neighbor interchanges (NNI) swapping strategy is implemented by combin-

ing the arguments `spr` and `sectorial` (see *Join method* group of arguments): `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.

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 `POY4` is to choose the first one available that shows a better cost than the current best cost.

`annealing:(FLOAT, FLOAT)` Uses simmulated annealing [13]. If the argument's value is (a, b), `POY4` 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 `POY4` drifting function [9]. If the argument's value is (a, b), then `POY4` 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.

Branch break order During the local search, a branch is broken and local branch swapping is performed (see *Neighborhood* group of arguments), the precise choice of which branches should be 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 `POY4`.

`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.

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[:INTEGER | (depth:INTEGER, file:STRING)]` Sets constraints on 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 to consider 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 and perform swapping with the value of `depth` set to `0` (no maximum distance is specified).

`all[:INTEGER ]` Turns off all preference strategies to make a join, simply try all possible join positions for each pair of clades generated after a break, in a randomized order.

`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.

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

`bfs[:INTEGER ]` Reroots using breath first search [3] 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 samples During the search, POY4 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 3.3.16). 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, POY4 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).

Character transformation

`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.

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, TBR, threshold:0, bfs)` By default, current trees are submitted to a round of 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 distance 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"))`.

- `swap(constraint:(file:"bleh", depth:4))`
  Uses the tree in the file `bleh` as a constraint tree and joins at distance at most 4 from the breaking branch during the swap.

- `swap(recover, timedprint:(5, "timedprint.txt"))`
  Saves the current best tree to 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 type. This includes changing in costs for indels and substitution, modifying character weights, converting dynamic into static homology characters, and transforming nucleotide into chromosomal 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 specified by identifiers as defined in the description of the command `select` (Section 3.3.23). Identifiers and methods are included in parentheses and separated by a comma. It is important to remember that only identifiers of *characters* (such as `names`, `codes`, among others) can be used. The parentheses separate these essential arguments from all other optional arguments that might be included in the list. Thus, if only identifiers and methods are specified, the argument list of `transform` is included

in double parentheses. For example, the command `transform((all, gap_-opening:1))` contains only an identifier (`all`) and a method (`gap_opening`). Minimally, only methods can be specified; in that case, the transformation is applied to all characters to which the transformation method can be applied and only a single set of parentheses is used. For instance, `transform(gap_-opening:1)`, where `gap_opening` defines the transformation method.

There are no default values for `transform`, that is if no methods 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.

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).

`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 [29]. This method greatly accelerates searching and is applicable only to nucleotide sequences under dynamic homology analysis.

`auto_sequence_partition` Evaluates each fragment 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.

`direct_optimization` Transforms the characters specified so that the initial assignment of sequences to the internal vertices of a tree use direct optimization [24]. This method is recommended for small alphabets

(less than 7 elements). Otherwise fixed states is recommended. It is only applicable to dynamic homology characters.

`fixed_states` Transforms the characters specified in fixed state characters [25] 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. This method is recommended for large alphabets (more than or equal to 7 elements). It is only applicable to dynamic homology characters.

`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 `POY4` 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.

`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 [31]. This option is intended only for heuristic search purposes and is applicable only to dynamic homology characters.

`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.

`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 [29]. 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 value `remove` excludes all uninformative characters information (except autapomorphies), whereas the value `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 reverted. 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.

`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.

`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.

`tcm:(INTEGER, INTEGER)` Defines transformation cost matrix. The first integer value specifies substitution cost, the second integer value defines indel cost. By default, the cost of substitution is `1`, and the cost of an indel is `2` (`tcm:(1,2)`).

`tcm:STRING` 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
```

`weight:argument` Changes the cost of specified characters by a constant value (weight) which is specified by either a float or an integer value. This method is applicable to any character type.

`weightfactor:argument` Changes the cost of specified characters by a multiplicative factor (weight factor) which is specified by either a float or an integer value. This method is applicable to any characters.

Chromosomal transformation methods For chromosome and genome character types, `POY4` optimizes nucleotide-, locus-, and chromosome-level variation simultaneously. The arguments in this group transform nucleotide characters into chromosomal character to 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 GRAPPA, Genome Rearrangements Analysis under Parsimony and other Phylogenetic Algorithms, available at `http://www.cs.unm.edu/~moret/GRAPPA/`.

`breakinv_to_custom` Transforms `breakinv` character type into `custom_alphabet` characters. This transformation prevents the use of rearrangement operations.

`custom_to_breakinv:([argument list])` Transforms `custom_alphabet` characters into the `breakinv` character type to allow for rearrangement operations (translocations and inversions; duplications are not currently supported). This argument is useful, for example, when `custom_alphabet` characters are used to define a sequence of individual genes and one is interested in detecting potential change in their order within a chromosome. See the command `read` (Section 3.3.14) for the description on how to load the `custom_ alphabet` and `breakinv` character types. The optional list of arguments includes the arguments of the `dynamic_pam` that can also be specified subsequently, as a separate step (for a sample script using this character transformation see tutorial `4.8`).

`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 `dynamic_pam` If no `dynamic_pam` values are specified, its default values are applied.

`dynamic_pam:([argument list])` Specifies parameters for creating chromosome- and genome-level HTUs (medians). The arguments of `dynamic_pam` define homologous blocks within unannotated chromosome sequences using (`min_seed_length`, `min_loci_len`, and `min_rearranged_len`); specify the cost of locus-level transformation events: (i.e. `locus_inversion` or `locus_breakpoint`, and `locus_indel`); specify the cost of chromosome-level transformation events: (i.e. `chrom_breakpoint`, and `chrom_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.

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 3.3: Examples of gene rearrangements: inversions and translocations.

$$\begin{array}{ccccc} g_1 & g_2 & g_3 & g_4 & g_5 \\ g_1 & g_2 & -g_5 & -g_4 & -g_3 \end{array}$$

Figure 3.4: 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_1, g_2, g_3).

`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`.

`locus_breakpoint:INTEGER` Calculates the breakpoint distance [1] 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 gene appears exactly once and its orientation is either positve or negative. Gene orders are altered by gene rearrangement operations: gene inversion, gene translocation, gene inversion and translocation (see Figure 3.3). 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 consectively in G' [19]. The breakpoint distance between G and G' is the number of breakpoints between them. Figure 3.4 shows two breakpoints between G and G'. The breakpoint can be calculated easily in linear time. This argument *cannot* be used in conjunction with `inversion`. The default value of `locus_breakpoint` is `10`.

`locus_inversion:INTEGER` Calculates the inversion distance [10] 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' [10]. Figure 3.4 shows one inversion between G and G'. The inversion can be calculated in linear time. The breakpoint distance is normally larger than inversion distance. This argument *cannot* be used in conjunction with `breakpoint`.

`locus_indel:(INTEGER, FLOAT)` Specifies the cost for insertion/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 $o = 10, e = 1.0$.

`min_seed_length:INTEGER` Specifies the minimum length of identical (invariant, completely conserved) contiguous sequence fragments during comparison between two chromosomes. The integer value of `min_seed_length` is the number of nucleotides. Correct identification of such fragments facilitates detecting chro-

mosome rearrangement events and accelerates other operations (such as tree building and swapping). However, if `min_seed_length` value is set too low (detecting many short fragments) or too high (such that no identical fragments are detected) the time required for subsequent searching procedures may significantly increase. The optimal `min_seed_length` value depends on the specifics of a given dataset (see Figure 3.5). The default value of `min_seed_length` is `9`.

`min_loci_len:INTEGER` Creates a pairwise alignment between two chromosomes to detect conserved areas ("blocks"). However, only blocks of lengths (in number of nucleotides) greater or equal to the specified `min_loci_len` value are considered as hypothetically homologous blocks and used as anchors to divide chromosomes into fragments. Thus, increasing the value of `min_loci_len` decreases the chance of inferring small-size rearrangements (see Figure 3.5). The default value is `100`.

`min_rearrangement_len:INTEGER` Two seeds are said to be *non-rearranged*, if their distance is less than the predefined threshold value set for `min_rearrangement_len`. In other words, it is unlikely that rearrangement operations can occur between two non-rearranged seeds if they are connected (see Figure 3.5). The default value is `100`

`chrom_breakpoint:INTEGER` Calculates the breakpoint distance [1] between two sequences of multiple chromosomes given the cost for rearrangement specified by an integer value. The breakpoint distance takes into account locus rearrangements between non-homologous chromosomes (translocations) but not inversions. For further discussion on how breakpoint distance is calculated see the argument `locus_breakpoint`. The default value of `chrom_breakpoint` is `100`.

NOTE

Note that the arguments `locus_breakpoint` and `chrom_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 arguments of `dynamic_pam` is arbitrary.

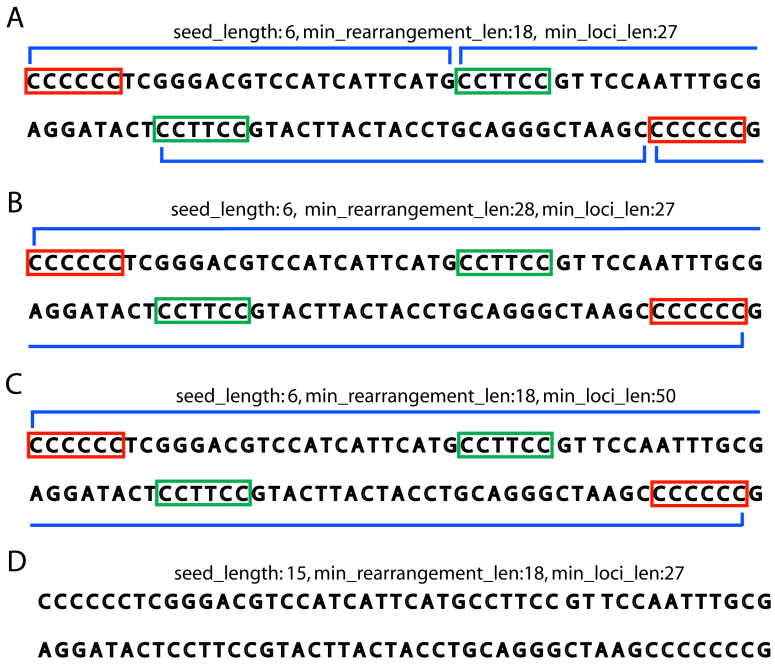


Figure 3.5: The effect of dynamic parameter arguments (`seed_length`, `min_rearrangement_len`, and `min_loci_len`) on determining homologous blocks in unannotated chromosomal sequences. *Case A* allows for rearrangements between the short homologous blocks within the blue brackets constructed upon six nucleotide seeds (red and green), *Case B* restricts rearrangement between seeds that are less than 28 nucleotides apart. *Case C* restricts rearrangement by requiring homologous blocks at least 50 nucleotides in length. *Case D* restricts recognition of homology between sequences by setting seed length of conserved seeds at 15.

`chrom_hom:(FLOAT)` Specifies the lower limit of distance between two chromosomes beyond which 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 if chromosome is circular (boolean value `true`) or linear (boolean value `false`). The default value of `circular` is `false` (linear chromosome).

`median:INTEGER` Specifies the number alternative locus and chromosome rearrangements of the best cost selected (randomly) for each HTU (hypothetical taxon unit) or median. Limiting the number of rearrangements stored in memory (smaller value of `median`) is 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.

`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`.

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((all, tcm:(1,1)))`
  This command is equivalent to `transform((dynamic, tcm:(1,1)))`.

- `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((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((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`, and for 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 actual 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(fixed_states)`
  Transformed all sequence characters into fixed states characters.

- `transform((names:("gen1", "gen4"), fixed_states))`
  Transformed only specified sequence characters (`gen1` and `gen4`) into fixed states characters.

- `transform(custom_to_breakinv:(circular:true))`
  In this example all custom_alphabet data is transformed into the breakinv data type and is treated as a circular chromosome.

- `transform(seq_to_chrom:(locus_indel:(50, 1.0), min_seed_length:15))`
  All applicable (*i.e.* sequence) data are transformed into chromosome data with the minimum length of identical contiguous sequence fragments which form the seeds of homologous blocks set at `15` nucleotides and the locus-level gap opening cost is set at `50` with a gap extension cost at `1.0`.

- `transform((all, dynamic_pam:(locus_breakpoint:10, min_rearrangement_len:60, median:1, circular:false)))`
  This example shows the transformation of chromosomal data using the argument `dynamic_pam` to set the locus rearrangement (breakpoint) cost at `10`, and only strings of `60` or more nucleotides are considered in determining possible rearrangements between identified seeds. The chromosome data are treated as linear and only a single set of median rearrangements are stored.

3.3.28 use

Syntax

use`(STRING)`

Description

Restores from memory the state of a `POY4` 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 very useful for exploring alternative cost regimes and terminal sets within a single `POY4` 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 `POY4` 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

Eliminates the data stored in memory (all character data, trees, *etc.*).

Examples

- `wipe ()`

Chapter 4

POY4 Tutorials

These tutorials are intended to provide guidance for more sophisticated applications of POY4 that involve multiple steps and a combination of different commands. Each tutorial contains a POY4 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 Interphace*, the most practical way to do this is to use POY4 scripts (see *POY4 Quick Start* for more information on POY4 scripts).

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 as optimal parameters.

The tutorials use sample datasets that are provided with POY4 installation but can also be downloaded from the POY4 site at

`http://research.amnh.org/scicomp/projects/poy.php`

The minimal required items to run the tutorial analyses are the POY4 application and the sample datafiles. Running these analyses requires some familiarity with POY4 interface and command structure that can be found in the preceding chapters.

4.1 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 thoroughly exploring local tree space and escape the effect of composite optima by effectively 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", aminoacids:("41.aa"))
set(seed:1,log:"all_data_search.log",root:"t1")
report(timer:"search start")
transform(tcm:(1,2),gap_opening:1)
build(250)
swap(threshold:5.0)
select()
perturb(transform(static_approx),iterations:15,ratchet:(0.2,3))
select()
fuse(iterations:200,swap())
select()
report("all_trees",trees:(total),"constree",graphconsensus,
"diagnosis",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 provide are useful for housekeeping purposes.

- `read("9.fas","31.ss", aminoacids:("41.aa"))` This command imports all the nucleotide sequence datafiles (all files with the extension `.seq`), a morphological datafile `morph.ss` in Hennig86 format, and an aminoacid datafile `myosin.aa`.

- `set(seed:1,log:"all_data_search.log",root:"t1")` The `set` command specifies conditions prior to tree searching. The `seed` is used to ensure that the subsequent randomization procedures (such as tree building and selecting) are reproducible. Specifying the log produces a file, `all_data_search.log` that provides an additional means to monitor the process of the search. The outgroup (`taxon1`) is designated by 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.

- `transform(tcm:(1,2),gap_opening:1)` This command sets the transformation cost matrix for molecular data to be used in calculating the cost of the tree. Note, that in addition to the substitution and indel costs, the `transform` specifies the cost for gap opening.

- `build(250)` This commands begins tree-building step of the search that generates 250 random-addition trees. A large number of independent starting point insures that a large portion of tree space have been examined.

- `swap(threshold:5.0)` `swap` specifies that each of the 250 trees is subjected to alternating SPR and TBR branch swapping routine (the default of `POY4`). In addition to the most optimal trees, all the suboptimal trees found within 5% of the best cost are thoroughly evaluated. This step ensures that the local searches settled on the local optima.

- `select()` Upon completion of branch swapping, this command retains only optimal and topologically unique trees; all other trees are discarded from memory.

- `perturb(transform(static_approx),iterations:15,ratchet:(0.2,3))` This command subjects the resulting trees to 15 rounds of ratchet, re-weighting 20% of characters by a factor of 2. During ratcheting, the dynamic homology characters are transformed into static homology characters, so that the fraction of nucleotides (rather than of sequence fragments) is being re-weighted. This step, that begins at multiple local maxima, is intended to further traverse the tree space in search of a global optimum.

- `fuse(iterations:200,swap())` In this step, up to 200 swappings 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`.

- `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",trees:(total),"constree", graphconsensus,"diagnosis",diagnosis)` This command produces a series of outputs of the results of the search. It includes a file containing best trees in parenthetical notation and their costs (`all_trees`), a graphical representation (in PostScript format) of the strict consensus (`constree`), and the diagnoses for all best trees (`diagnosis`).

- `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 any output to the log file, `all_data_search.log`.

- `exit()` This commands ends the `POY4` session.

4.2 Searching under iterative pass

The following script implements a strategy for a thorough search under iterative pass optimization. The 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",aminoacids:("41.aa"))
read("inter_tree.tre")
transform(tcm:(1,2),gap_opening:1)
set(iterative)
```

```
swap(around)
select()
report("all_trees",trees:(total),"constree", graphconsensus,
"diagnosis",diagnosis)
transform ((all, static_approx))
report ("phastwinclad.ss", phastwinclad)
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 provide are useful for housekeeping purposes.
- `read("9.fas","31.ss",aminoacids:("41.aa"))` This command imports all the nucleotide sequence datafiles (all files with the extension `.seq`), a morphological datafile `morph.ss` in Hennig86 format, and an aminoacid datafile `myosin.aa`.
- `read("inter_tree.tre")` This command imports a tree file, `inter_tree.tre`, that contains the most optimal tree from prior analyses.
- `transform(tcm:(1,2),gap_opening:1)` This command sets the transformation cost matrix for molecular data to be used in calculating the cost of the tree. Note, that in addition to the substitution and indel costs, the `transform` specifies the cost for gap opening.
- `set(iterative)` This command sets the optimization procedure to iterative pass.
- `swap(around)` This commands specifies that the the imported tree is subjected to alternating SPR and TBR branch swapping routine (the default of `POY4`) 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",trees:(total),"constree", graphconsensus,"diagnosis",diagnosis)` This command produces a series of outputs of the results of the search. It includes a file containing best trees in parenthetical notation and their costs (`all_trees`), a graphical representation (in PostScript format) of the strict consensus (`constree`), and the diagnoses for all best trees (`diagnosis`).

- `transform ((all, "static_approx"))` This command transforming all data into static homology characters corresponding to their implied alignments is necessary before reporting the data in the Hennig86 format.

- `report ("phastwinclad.ss", phastwinclad)` This command produces a file in the Hennig86 format which can be imported into other programs, such as WinClada.

- `exit()` This commands ends the `POY4` session.

4.3 Bremer support

This tutorial builds on the previous tutorials to illustrate Bremer support calculation on trees constructed using dynamic homology characters

```
(* Bremer support part 1: generating trees *)
read("18s.fas","28s.fas")
set(root:"Americhernus")
build(200)
swap(all,visited:"tmp.trees", timeout:3600)
select()
report("my.tree",trees)
exit()

(* Bremer support part 2: Bremer calculations *)
read("18s.fas","28s.fas","my.tree")
report("support_tree.ps",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 provide are useful for housekeeping purposes.

- `read("18s.fas","28s.fas")` This command imports the nucleotide sequence files `18s.fas, 28s.fas`.

- `set(root:"Americhernus")` The `set` command specifies conditions prior to tree searching. The outgroup (`Americhernus`) is designated by the `root`, so that all the reported trees have the desired polarity.

- `build(200)` This commands initializes tree-building and generates 200 random-addition trees.

- `swap(all,visited:"tmp.trees")` The `swap` command specifies that each of the trees be subjected to an alternating SPR and TBR branch swapping routine (the default of `POY4`). The `all` argument turns offf all swap heuristics. The `visited:"tmp.trees"` argument stores every visited tree in the file specified. Due to the large number of trees that the specified file will hold the argument `timeout` can be used to specify the number of seconds allowed for swapping. Alternately the `swap` command can be performed as a separate analysis and terminated at the users discretion to maximize the number of trees generated.

- `select()` Upon completion of branch swapping, this command retains only optimal and topologically unique trees; all other trees are discarded from memory.

- `report("my.tree",trees)` This command will save the swapped tree, `my.tree` to a file.

- `exit()` This commands ends the `POY4` session.

- `(* Bremer support part 2:  Bremer calculations *)` A comment indicating the intent of the commands which follow.

- `read("18s.fas","28s.fas","my.tree")` This command imports the nucleotide sequence files `18s.fas, 28s.fas` and the tree file, `my.tree` for which the support values will be generated. It is important to only read the selected `"my.tree"` file rather than the expansive `"tmp.trees"` file which will be used in bremer calculations.

- `report("support_tree.ps",graphsupports:bremer:"tmp.trees")` The `report` command in combination with a file name and the `graphsupports` generates a postscript file designated by the name `support_tree.ps` with bremer values for the selected trees held in `tmp.trees`. It is strongly recommended that this more exhaustive approach is used for calculating Bremer supports rather than simply using the
`graphsupports` default s.

- `exit()` This commands ends the `POY4` session.

4.4 Jackknife support

This tutorial illustrates calculating Jackknife support values for trees constructed with static homology characters. Although it is possible to calculate both Jackknife and Bootstrap support values for trees constructed using dynamic homology characters, it is not recommended because resampling of dynamic characters occurs at the fragment (rather than nucleotide) level. Alternately dynamic homology characters can be converted to static characters using the transform argument `static_approx`, as it is common for biologists to want to compute support values by resampling the characters from a fixed alignment. In jackknife support a specified number of pseudo-repicates are performed independently such that in each one a percentage of characters is selected at random, without replacement. The frequency of clade occurrence is its jackknife value.

```
(* Jackknife support using static nucleotide characters *)
read ("28s.fas")
search (max_time:0:0:5)
select ()
report ("tree_for_supports.tre", trees)
transform(static_approx)
calculate_support (jackknife:(remove:0.50, resample:1000))
report (supports:jackknife)
report ("jacktree", graphsupports:jackknife)
exit()
```

- `(* Jackknife support using static nucleotide characters *)` This first line of the script is a comment. While comments are optional and do not affect the analyses, they provide are useful for housekeeping purposes.

- `read("28s.fas")` This command imports nucleotide sequence file `28s.fas`.

- `search(max_time:0:0:5)` This command performs a search (i.e. build, swap, perturb, fuse) of the data `28s.fas` for a maximum of 5 min (note that the short search time was selected for demonstration purposes to expedite the tutorial and not as a general time recommendation for actual data analyses).

- `select()` This command retains only optimal and topologically unique trees; all other trees are discarded from memory.

- `report("tree_for_supports.tre", trees)` This command outputs a parenthetical representation of the tree `"tree_for_supports.tre"`.
- `transform(static_approx)` This command transforms the data (i.e. build, swap, perturb, fuse) of the data `28s.fas` for a maximum of 2 hours.
- `calculate_support(jackknife,(remove:0.50,resample:1000), 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 `remove:0.50`. The total number of of pseudoreplicates to be performed is designated by the `resample:1000` argument. The `build(5)` argument indicates the number of Wagner trees to be built in each pseudoreplicate, and the `swap(tbr,trees:3)` argument subjects each Wagner build to tbr swapping keeping up to three best trees per search round.
- `report(supports:jackknife)` This command outputs a parenthetical representation of a tree with the support values previously calculated with the `calculate_support` command.
- `report("jacktree",graphsupports:jackknife)` The `report` command in combination with a file name and the `graphsupports` generates a postscript file with jackknife values designated by the name specified (*i.e.* `jacktree`).
- `exit()` This commands ends the `POY4` session.

4.5 Sensitivity analysis

This tutorial demonstrates how data for parameter sensitivity analysis is generated. Sensitivity analysis [23] 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.

POY4 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 POY4 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:"111.txt")
build(100)
swap(timeout:3600)
select()
report("111.tre",trees:(total) ,"111con.tre",consensus,
"111con.ps",graphconsensus)
use("original_data")
transform(tcm:"112.txt")
build(100)
swap(timeout:3600)
select()
report("112.tre",trees:(total),"112con.tre",consensus,
"112con.ps",graphconsensus)
exit()
```

- `(* sensitivity analysis *)` This first line of the script is a comment. While comments are optional and do not affect the analyses, they provide are useful for housekeeping purposes.

- `read("9.fas")` This command imports all dynamic homology nucleotide data.

- `set(root:"t1")` The outgroup (`taxon1`) is designated by 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:"111.txt")` This command applies a transformation cost matrix from the file `111.txt` to for subsequent tree searching.

- `build(100)` This commands begins tree-building step of the search that generates 250 random-addition trees. A large number of independent starting point insures that thee large portion of tree space have been examined.

- `swap(timeout:3600) swap` specifies that each of the 100 trees build in the previous step is subjected to alternating SPR and TBR branch swapping routine (the default of `POY4`). 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 a tremendous amount of 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 and the search is trapped in its neighborhood.

- `select()` Upon completion of branch swapping, this command retains only optimal and topologically unique trees; all other trees are discarded from memory.

- `report("111.tre",trees:(total) ,"111con.tre",consensus, "111con.ps", graphconsensus)` This command produces a file containing best tree(s) in parenthetical notation and their costs (`111.tre`), a a file containing the strict consensus in parenthetical notation (`111con.tre`), and a graphical representation (in PostScript format) of the strict consensus (`111con.ps`).

- `use("original_data")` This command restored the original (non-transformed) data from the temporary file `original_data` generated by `store`.

- `transform(tcm:"112.txt")` This command applies a different transformation cost matrix from the file `112.txt` to for another round of tree searching under this new cost regime.

- `build(100)` This commands begins tree-building step of the search that generates 100 random-addition trees. A large number of independent starting point insures that thee large portion of tree space have been examined.
- `swap(timeout:3600) swap` specifies that each of the 100 trees build in the previous step is subjected to alternating SPR and TBR branch swapping routine (the default of `POY4`) 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("112.tre",trees:(total),"112con.tre",consensus, "112con.ps", 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 commands ends the `POY4` session.

4.6 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. Prior to attempting an analysis of unannotated chromosomes it is necessary to enable the `"long sequences"` option when compiling the `POY4` program.

```
(* Chromosome analysis of unannotated sequences *)
read(chromosome:("ua15.fas"))
transform((all,dynamic_pam:(locus_breakpoint:20,locus_indel:
(10,1.5),circular:true,median:2, min_seed_length:15,
min_rearrangement_len:45, min_loci_len:50,median:2,swap_med:1)))
build()
swap()
select()
report("chrom",diagnosis)
report("consensustree",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 provide are useful for housekeeping purposes.

- `read(chromosome:("ua15.fas"))` This command imports the unannotated chromosomal sequence file `ua15.fas`. The argument `chromosome` specifies the characters as unannotated chromosomes.

- `transform((all,dynamic_pam:(locus_breakpoint:20,locus_indel:(10,1.5),circular:true,seed_length:15,rearranged_len:50,sig_-block_len:50,median:2,swap_med:1)))` The `transform` followed by the argument `dynamic_pam` specifies the conditions to be applied when calculating chromosome-level HTUs (medians). The argument `locus_-breakpoint:20` applies a breakpoint distance between chromosome loci with the integer value determining the rearrangement cost. The argument `locus_indel:10,1.5` specifies the indel costs for the chromosomal segments, whereby the integer 10 sets the gap opening cost and the float 1.5 sets the gap extension cost. As the type of chromosomal sequences being analyzed are of mitochondrial origin, the argument `circular:true` treats each chromosome sequence as a continuous rather than linear. The argument `min_seed_length:15` sets the minimum length of identical continuous fragments (seeds) at 15. As seeds are the foundation for larger homologous blocks setting the seed length to an integer appropriate for the data is critical to optimizing the efficiency with which the program correctly identifies chromosomal fragments and detects rearrangements. The `min_rearrangement_len` argument sets the lower limit for number of nucleotides between two seeds such that each is considered independent of the other. Independent seeds belong to separate homologous blocks such that rearrangement events between blocks can be detected. The argument `min_loci_len` provides the integer value determining the minimum number of nucleotides constituting an homologous block. In this example, because the data are mitochondrial containing relatively short homologous tRNA sequences, both the `min_rearrangement_len` and the `min_loci_len` were set to values below the defaults for these arguments. The `median` specifies the number of best cost locus-rearrangements which will be considered for each HTU (median), while the `swap_med` argument specifies the maximum number of swapping iterations performed in searching for the best pairwise alignment between two chro-

mosomes. Because values for the `median` and `swap_med` arguments set above the default (1) will significantly increase the calculation time, the default values are recommended for larger chromosomal data sets.

- `build()` This commands begins the tree-building step of the search that generates by default 10 random-addition trees. 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 `POY4`).
- `select()` Upon completion of branch swapping, this command retains only optimal and topologically unique trees; all other trees are discarded from memory.
- `report("chrom",diagnosis)` The `report` command in combination with a file name and the `diagnosis` outputs the optimal median states and edge values to a specified file (`chrom`).
- `report("consensustree",graphconsensus)` The `report` command in combination with a file name and the `graphconsensus` generates a postscript strict consensus file of the trees generated (`consensustree`).
- `exit()` This commands ends the `POY4` session.

4.7 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"))
transform((all,dynamic_pam:(locus_inversion:20,locus_indel:(10,
1.5),circular:false,median:1,swap_med:1)))
build()
swap()
select()
report("Annotated",diagnosis)
report("consensustree",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 provide are useful for housekeeping purposes.
- `read(annotated:("aninv2"))` This command imports the annotated chromosomal sequence file `aninv2`. The argument `annotated` specifies the characters.
- `transform((all,dynamic_pam:(inversion:20,locus_indel:(10,1.5),median:1,swap_med:1)))` The `transform` followed by the argument `dynamic_pam` specifies the conditions to be applied when calculating chromosome-level HTUs (medians). The argument `locus_inversion:20` applies an inversion distance between chromosome loci with the integer value determining the rearrangement cost. The argument `locus_indel:10,1.5` specifies the indel costs for the chromosomal segments, whereby 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 arguments `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 designated for the `median` and `swap_med` arguments.
- `build()` This commands begins the tree-building step of the search that generates by default 10 random-addition trees. 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 `POY4`).
- `select()` Upon completion of branch swapping, this command retains only optimal and topologically unique trees; all other trees are discarded from memory.
- `report("Annotated",diagnosis)` The `report` command in combination with a file name and the `diagnosis` outputs the optimal median states and edge values to a specified file (`Annotated`).
- `exit()` This commands ends the `POY4` session.

4.8 Custom alphabet break inversion characters

This tutorial illustrates the analysis of the break inversion character type. Break inversion characters are generated by transforming user-defined `custom_alphabet` characters. For example, observations of developmental stages could be represented in a corresponding array such that for each terminal taxon there is a sequence of observed developmental stages which are represented by a user-defined alphabet. To allow rearrangement as well as indel events to be considered among alphabet elements, requires either reading in the data with the `breakinv` argument or transforming the `custom_alphabet` sequences read to `breakinv` characters.

```
(* Custom Alphabet to Breakinv characters *)
read(custom_alphabet:("ca1.fas","m1.fas"))
transform((all,custom_to_breakinv:()))
transform((all,dynamic_pam:(locus_breakpoint:20,
locus_indel:(10,1.5),median:1,swap_med:1)))
build()
swap()
select()
report("breakinv",diagnosis)
report("consensustree",graphconsensus)
exit()
```

- `(* Custom Alphabet to Breakinv characters *)` This first line of the script is a comment. While comments are optional and do not affect the analyses, they provide are useful for housekeeping purposes.

- `read(custom_alphabet:("ca1.fas","m1.fas"))` This command imports the user-defined `custom_alphabet` character file `ca1.fas` and the accompanying transformation matrix `m1.fas`.

- `transform((all,custom_to_breakinv:()))` This command transforms `custom_alphabet` characters to `breakinv` characters which allow for rearrangement operations.

- `transform((all,dynamic_pam:(locus_breakpoint:20,locus_indel:(10,1.5),median:1,swap_med:1)))` The `transform` followed by the argument `dynamic_pam` specifies the conditions to be applied when calculating medians. The argument `locus_breakpoint:20` applies a breakpoint distance calculation where the integer value specifies

the rearrangement cost of `breakinv` elements. The argument `locus_indel:10,1.5` specifies the indel costs for each `breakinv` element, whereby 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 `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()` This commands begins the tree-building step of the search that generates by default 10 random-addition trees. 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 `POY4`).

- `select()` Upon completion of branch swapping, this command retains only optimal and topologically unique trees; all other trees are discarded from memory.

- `report ("breakinv",diagnosis)` The `report` command in combination with a file name and the `diagnosis` outputs the optimal median states and edge values to a specified file (`breakinv`).

- `exit()` This commands ends the `POY4` session.

4.9 Genome analysis: multiple chromosomes

This tutorial illustrates the analysis of genome-level transformations using data from multiple chromosomes. Prior to attempting an analysis of unannotated chromosomes it is necessary to enable the `"long sequences"` option when compiling the `POY4` program.

```
(* Genome analysis of multiple chromosomes *)
read (genome:("gen5bp"))
transform((all,dynamic_pam:(chrom_breakpoint:80, chrom_indel:
(15,2.5),locus_inversion:20,locus_indel:(10,1.5), median:1,
swap_med:1)))
```

```
build()
swap()
select()
report("genome",diagnosis)
report("genconsensus",graphconsensus)
exit()
```

- `(* Genome analysis of multiple chromosomes*)` This first line of the script is a comment. While comments are optional and do not affect the analyses, they provide are useful for housekeeping purposes.

- `read(genome:("gen5bp"))` This command imports the genomic sequence file `mit5.txt`. The argument `genome` specifies the characters as data consisting of multiple chromomsomes.

- `transform((all,dynamic_pam:(chrom_breakpoint:80,chrom_indel:(15, 2.5),locus_breakpoint:20,locus_indel:(10,1.5),median:1,swap_-med:1)))` The `transform` followed by the argument `dynamic_pam` specifies the conditions to be applied when calculating genome-level HTUs (medians). The argument `chrom_breakpoint:80` applies a breakpoint distance between chromosomes with the integer value determining the rearrangement cost. The argument `chrom_indel:15,1.5` specifies the indel costs for each entire chromosome, whereby the integer sets the gap opening cost and the float sets the gap extension cost. The argument `locus_inversion:20` applies an inversion distance between loci with the integer value determining the rearrangement cost. The argument `locus_indel:10,1.5` specifies the indel costs for the chromosomal segments, whereby 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 `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()` This commands begins the tree-building step of the search that generates by default 10 random-addition trees. 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 `POY4`).

- `select()` Upon completion of branch swapping, this command retains only optimal and topologically unique trees; all other trees are discarded from memory.

- `report("genome",diagnosis)` The `report` command in combination with a file name and the `diagnosis` outputs the optimal median states and edge values to a specified file (`genome`).

- `report("genconsens",graphconsensus)` The `report` command in combination with a file name and the `graphconsensus` generates a postscript strict consensus file of the trees generated (`genconsensus`).

- `exit()` This commands ends the `POY4` session.

Bibliography

[1] 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.

[2] K. Bremer. The limits of amino acid sequence data in angiosperm phylogenetic reconstruction. *Evolution*, 42:795–803, 1988.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

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

[5] James S. Farris. Hennig86, 1988.

[6] James S. Farris. The retention index and the rescaled consistency index. *Cladistics*, 5:417–419, 1989.

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

[8] Joseph Felsenstein. Confidence limits on phylogenies: An approach using the bootstrap. *Evolution*, 39(4):783–791, 1985.

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

[10] 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.

[11] M. D. Hendy and D. Penny. Branch and bound algorithms to determine minimal evolutionary trees. *Mathematical Biosciences*, 60:133–142, 1982.

[12] Mari Källersjö, James S. Farris, A. G. Kluge, and C. Bult. Skewness and permutation. *Cladistics*, 8:275–287, 1992.

[13] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. 220(4598):671–680, May 1983.

[14] A. G. Kluge and J. S. Farris. Quantitative phyletics and the evolution of anurans. *Systematic Zoology*, 30:1–32, 1969.

[15] T. Margush and F. R. McMorris. Consensus n-trees. *Bulletin of Mathematical Biology*, 43:239–244, 1981.

[16] Kevin C. Nixon. The parsimony ratchet, a new method for rapid parsimony analysis. *Cladistics*, 15(4):407–414, 1999.

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

[18] F. J. Rohlf. Consensus indices for comparing classifications. *Mathematical Biosciences*, 59:131–144, 1982.

[19] D. Sankoff and M. Blanchette. Multiple genome rearrangement and breakpoint phylogeny. *J. Comput. Biol.*, 5:555–570, 1998.

[20] 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.

[21] Le Sy Vinh, Andres Varon, Daniel Janies, and Ward 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.

[22] Le Sy Vinh, Andres Varon, and Ward C. Wheeler. Pairwise alignment with rearrangements. *Genome informatics*, 17(2):141–151, 2006.

[23] W. C. Wheeler. Sequence alignment, parameter sensitivity, and the phylogenetic analysis of molecular data. *Systematic Biology*, 44(3):321–331, 1995.

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

[25] W. C. Wheeler. Fixed character states and the optimization of molecular sequence data. *Cladistics*, 15(4):379–385, 1999.

[26] 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.

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

[28] W. C. Wheeler. *Optimization Alignment: down, up, error, and improvements*. Techniques in Molecular Systematics and Evolution. Birkhäuser, Basel, Boston, Berlin, 2002.

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

[30] W. C. Wheeler. Iterative pass optimization. *Cladistics*, 19:254–260, 2003.

[31] W. C. Wheeler, John Gatesy, and Rob DeSalle. Elision: A method for accommodating multiple molecular sequence alignments with alignment-ambiguous sites. *Molecular Phylogenetics and Evolution*, 4(1):1–9, 1995.

[32] Ward C. Wheeler, Lona Aagesen, Claudia P. Arango, Julian Faivoich, Taran Grant, Cyrille D'Haese, Daneil Janies, William Leo Smith, Andrés Varón, and Gonzalo Giribet. *Dynamic Homology and Systematics: A Unified Approach*. American Museum of Natural History, 2006.

General Index

POY 3.0 Commands Index