POY 4.0 Beta

QuickStart Guide and Program Documentation
Version 4.0.1983

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

Documentation
Ilya Tëmkin
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
Computer Science Department, The Graduate School and University Center, The City University of New York
Le Sy Vinh
Ward C. Wheeler
Ilya Tëmkin
Taran Grant
Division of Invertebrate Zoology, American Museum of Natural History
Kurt M. Pickett
Department of Biology, University of Vermont
Julián Faivovich
Departamento de Zoologia, Instituto de Biociências, Universidade Estadual Paulista
William Leo Smith
Division of Vertebrate Zoology, American Museum of Natural History

The American Museum of Natural History

Varón, A., L. S. Vinh, I. Bomash, W. C. Wheeler. 2007. `POY` 4.0.1983 Beta. New York, American Museum of Natural History. Documentation by Varón, A., L. S. Vinh, I. Bomash, W. Wheeler, I. Tëmkin, 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`.

Contents

Chapter 1

POY4 Quick Start

1.1 What is POY4

POY4 is a flexible, multi-platform program for phylogenetic analysis of molecular and other data under various optimality criteria. An essential feature of POY4 is that it implements the concept of dynamic homology allowing optimization of unaligned sequences. POY4 offers great flexibility for designing heuristic search strategies and implements an array of algorithms including swapping, tree fusing, tree drifting, and ratcheting. As output, POY4 generates a comprehensive character diagnosis, graphical representations of cladograms and their user-specified consensus, as well as 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 and take into account large-scale genomic events (translocations, inversions, and duplications).

Currently POY4 is beta software, and therefore it has some known glitches. Most of them will be worked out in the following months, and updated versions will be available on the program's webpage as we produce them. Our current schedule of work expect to have a final official version of the parsimony components of the program and a release of the beta components of the maximum likelihood components in mid may of 2007. For the list of known issues see the Section 1.12.8.

1.2 The structure of POY4 documentation

The 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 interface and its navigation, and teach how to run the program and get help. Subsequent sections (starting with *1.13 Using POY4*) build on that knowledge and give a step-by-step example on how to conduct a basic analysis and visualize the results. The *POY4 Quick Start* is not a tutorial on POY4; using POY4 assumes a knowledge of POY4 commands and their valid syntax that are detailed in the second chapter, *POY4 Commands*. More advanced operations are described in the third 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.

The Quick Start is created primarily for a typical user with limited experience using command-line applications and assumes little or no knowledge of Unix. Consequently, certain operations suggested here could be performed more efficiently by an experienced user, but in attempt to make the software as accessible as possible, we provide simple and intuitive step-by-step (platform-specific where necessary) instructions.

1.3 Requirements: software and hardware

1.3.1 Software

POY4 is a platform-independent, open-source program that is compiled for Mac OSX, Microsoft Windows, and Linux systems. POY4 *binaries* (compiled application file) is the only piece of software necessary to run POY4. Other utility programs (that are typically installed with major operation systems), can facilitate preparation of POY4 scripts (POY4 command batch files) and formatting datafiles.

Windows

Notepad is a basic text editor that can be used to create POY4 scripts and format datafiles. By default, it is located in the *Accessories* folder under *All Programs* of the *Start* menu.

Command Prompt provides a working environment for POY4 and is used to initiate a POY4 session. It can be accessed from the same

Accessories folder as Notepad.

Mac OSX

Terminal is an interface for UNIX operating systems and it provides a working environment for POY4; it is used to initiate a POY4 session. Terminal is located in the *Utilities* folder within the *Applications* folder. (The program X11, that is also provided with OS X, can be used as an alternative to Terminal.)

TextEdit is a basic text editor that can be used to create POY4 scripts and format datafiles. By default, it is located in the *Applications* folder. More flexible text editors, such as shareware applications like BBEdit or TextWrangler, are good alternatives.

Linux

A simple text editor, such as *nano*, is sufficient, though more powerful editors (such as *vim* or *emacs*) can make your life much easier writting scripts for POY4.

1.3.2 Hardware

POY4 runs on a variety of computers computers from laptops and desktops to Beowulf clusters of various sizes to symmetric multiprocessing hardware. There are no particular requirements for disk space. Processor speed and memory (and communications bandwidth and latency in parallel environments) are important. Depending on the size and complexity of the data, and the computational complexity of requested operations, a POY4 session can consume large amounts of memory. However, the flexible structure of POY4 allows for partitioning of individual tasks to prevent overwhelming the hardware. Strict guidelines cannot be provided because the performance depends on the specifics of a given dataset; however, one can estimate the the memory requirements by running a test command or script under less demanding settings.

1.4 Obtaining and installing POY4

Compressed files of POY4 binaries, source code, and documentation in PDF format are available for various Linux distributions, Microsoft Windows XP, and Mac OSX Tiger at the American Museum of Natural History Computational Sciences POY4 website:

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

The following detailed step-by-step instruction will guide you through downloading, decompressing, and installing `POY4` binaries for various platforms.

Windows

- Download the Windows compressed binary file to the desktop.
- Typically, double-clicking on the compressed file will decompress it into an executable program. If this does not work, decompress it using a the standard compression utility program (such as WinZip).
- Move the executable file to your home directory (typically the directory holding *My Documents*). This will make the program available for running under your user name.

Mac OSX

Download the disk image file to the desktop. There are versions for single processor and multiple processor machines. If you have more than one processor in your computer, download *both*; the single processor version of `POY4` provides a more user-friendly environment and allows for interruption of jobs. The multiprocessor version, on the other hand, runs as many times faster as there are processors available.
To install, simply double click on the icon. The program will appear in the *Applications* folder.

Linux

Download the gzipped copy of the program and store it in any of the folders of your `PATH` environment.

Compiling from the Source

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

1. The GNU Make tool.
2. OCaml version 3.09.3. or later.
3. The GNU Compiler Collection version 3.4 or later.
4. The ncurses library if you want the nice interactive console, thought this one is *not obligatory*.

Download, ungzip, and untar the POY4 source code; change the relevant paths and desired options in the `config` file. Then just run `make clean; make depend; make mpoy.opt`. The executable `mpoy.opt` can be found in the `src` directory.

The (most relevant) options available in the config file are:

USEINTERFACE specifies the preferred user interface. There are four supported interfaces, though 2 of them are only in early development stages and are not recommended. The valid options are:

ncurses (default) provides a nice terminal based environment for interactive experimentation with POY4. This is the most desirable environment when POY4 is running as a standalone program in a personal computer, or remotely in a server but with the user issuing commands interactively. Users can use the tab to autocomplete both commands and filenames This interface is particularly useful while learning POY4's commands. Requires an ncurses-compatible library.

flat provides an environment with minimal requirements and functionality. This is the most desirable interface for parallel runs, as there is no terminal requirements and a process output can easily be redirected from the standard error and standard output to files using regular shell facilities. The tab and arrow keys are readline functional to allow the user to autocomplete filenames and navigate the interface.

gtk2 provides a pre-alpha, extremely bad looking graphical user interface that *mimics* the ncurses interface. It is intended for use in Windows and Linux environments in the near future. It's not recommended for current use. Requires GTK+ and lablgtk.

cocoa provides a pre-alpha, extremely bad looking graphical user interface using Mac OS X's native cocoa environment. Well, it's bad looking, but how bad can it look in a Mac?. It's not recommented for current use. Requires Mac OS X.

USEPARALLEL specifies that POY4 should be compiled with support for parallel environments using the Message Passing Interface. It can be set to true or false (default). For further instructions see below.

USEGRAPHICS specifies that POY4 should be compiled with support for on-screen tree graphical output. The output is crappy right now, use it if you want to explore a dataset. The valid options are:

none (default) specifies that the program should not have on-screen graphical output support.

ocaml specifies that the program should have on-screen graphical output support. In Unix and Linux systems (including Mac OS X), requires X11.

tk specifies that the program should have on-screen graphical output support. This version drops the X11 requirement in Mac OS X (though this was an experiment and is not recommended).

USEWIN32 ensures that compilation is prepared for Microsoft Windows environments. The valid options are false (default) or true. Note that compiling for Windows is tricky! specially because the ncurses interface needs to be installed (this is easy if you have cygwin). If you want to run POY4 in windows you are better off downloading the binaries.

POY4 can be run in parallel environments using the Message Passing Interface. There are multiple implementations, and if you have a parallel environment, most likely your system administrator has already installed one. Ask him for the proper paths to set in your config file.

1.5 Starting a POY4 session

Windows

- Open the Command Prompt window:
  Start>All Programs>Accessories>Command Prompt.
- Resize the Command Prompt window to desirable dimensions. When POY4 is running, resizing the window is not possible.
- Change from the home directory to the directory containing POY4 binaries and datafiles. At the prompt enter cd ("change directory") followed by a space and the name of the directory (such as "POY4" as in the example below). Hit the Return key.
- Open POY4. At the prompt enter "poy4" and hit the Return key. This will open POY4 interface that will occupy the entire Command Prompt window.

Mac OSX

- Open the Terminal window:
  Macintosh HD>Applications>Utilities>Terminal. If a Terminal window does not open by default, go to File>New Shell.

- Change from the home directory to the directory containing the datafiles. At the Terminal prompt ($) enter `cd` ("change directory") followed by a space and drag the folder containing the datafiles in the window (see Figure 1.1). This will automatically generate the path to the folder where the datafiles are. (Alternatively, the path can be typed in.) Hit the Return key.

- Open `POY4`. At the Terminal' prompt enter `poy` and hit the Return key. This will open `POY4` interface.

- Resize the Command Prompt window to desirable dimensions. Use lower right corner of the Terminal window to resize the interface to the desired dimensions. Note that if the window is resized while `POY4` performs a task, the interface might get distorted for the rest of the process.

Linux

If you had placed the program in your PATH, then simply type `poy` in a terminal; this will start the `POY4` session.

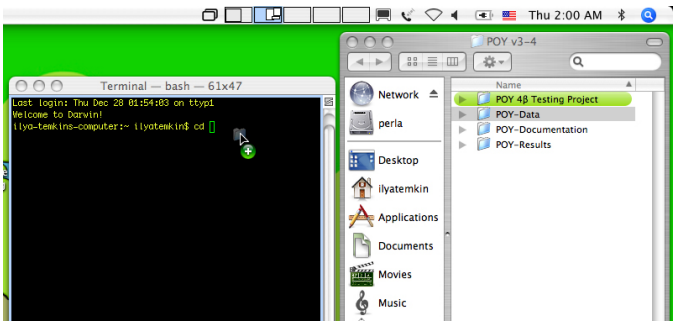

Figure 1.1: Specifying the location of datafiles. The folder `POY-Data` is dragged from the `POY v3-4` folder directly in the Terminal window.

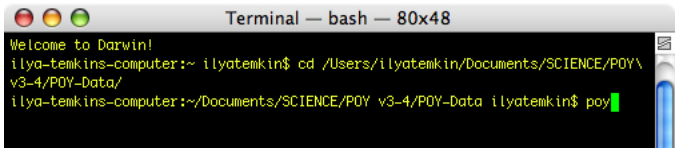

Figure 1.2: Starting POY4. At the folder containing datafiles, entering poy starts a POY4 session.

1.6 POY4 interface

The POY4 interface has the same appearance regardless of the operation system under which POY4 is run (except under parallel environment settings; see below). It has four windows: *POY Output*, *Interactive Console*, *State of Stored Search*, and *Current Job* (Figure 1.3).

POY Output window 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. By default, POY4 reports the list of imported data and generates error messages. Other outputs, however, must be requested using the `report()` command.

Interactive Console is used to instruct POY4 to import data, specify the kinds of analysis to be performed, and to request the desired output interactively by typing POY4 commands at the POY4 prompt (poy>). The commands are executed by hitting the Return key. The commands can be executed one at a time or entered sequentially until the Return key is pressed. (See Section 2.1.1 on the structure and syntax of POY4 commands.) Separating commands by spaces is optional but increases legibility. Alternatively, a file containing the list of commands (POY4 script, see below) can also be imported and executed at the prompt in the Interactive Console.

State of Stored Search 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 provides the range of their costs.

Current Job describes the currently running operation. When the operation is completed, the window is blank.

This POY4 interface is not available for parallel environments. Once the program is called, POY4 commands can be executed interactively or scripts

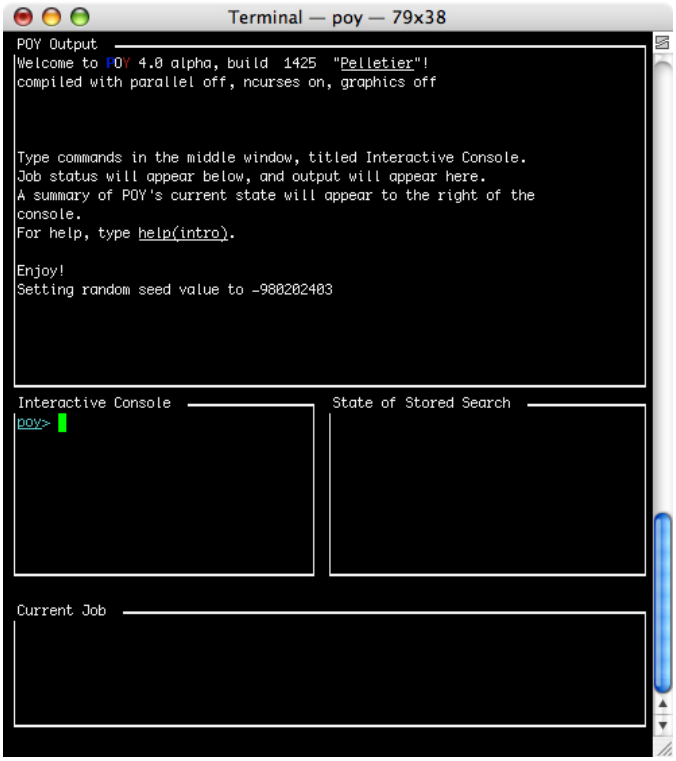

Figure 1.3: POY4 interface displayed in the Terminal window prior to analysis. Note the cursor at the POY4 prompt in the *Interactive Console* and that the *State of Stored Search* and *Current Job* windows are empty.

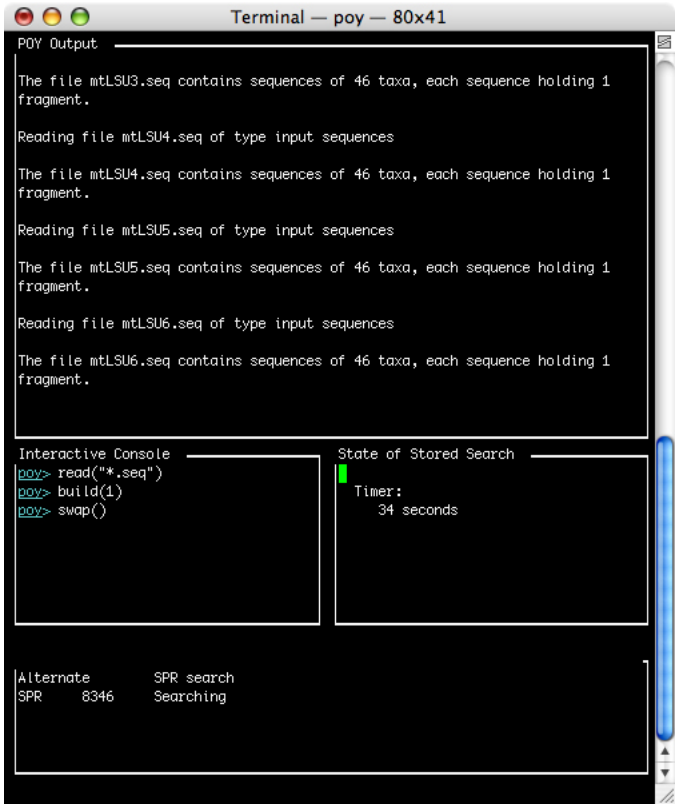

Figure 1.4: POY4 interface during a process. The *POY Output* window displays (by default) the information on the input datafiles. The *Interactive Console* lists the commands that have been consecutively executed. The *Current Job* window shows the state of the current operation and the current tree score. The *State of Stored Search* shows the time elapsed since the last command, `swap`, was initiated.

can be submitted as when using the *Interactive Console*. By default, the `POY4` will print the output on screen (the same output that is reported in *POY Output* under non-parallelized setting).

1.7 Navigating the interface

1.7.1 Entering commands

The *Interactive Console* is the only part of the interface that allows communication with `POY4`; that is where commands and scripts are executed. Once the `POY4` interface is called, the cursor appears in the *Interactive Console* and `POY4` is ready to accept commands. `POY4` interface 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 the commands entered during the session.

1.7.2 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 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. It is important to know that only 1000 lines are stored in the memory and the output that was reported before that will not be accessible by scrolling. If it is desired to keep the entire output or specific items in the output, the log can be created (using the command `set()`, see `log` (Section 2.3.24)) or specific outputs can be redirected to files (see `report` (Section 2.3.19)).

1.7.3 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 (which is 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 with the *Current Job* and *State of the State of Current Search*.

1.7.4 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. This command does not close the program, it only stops the last process that was running keeping in memory all the data and the results of the operation executed last. New commands can subsequently be entered.

1.8 Errors

`POY4` reports errors in several ways. If there is an error pertaining to wrong syntax (such as a typo in a command name), `POY4` will indicate the location of an error by underlining the problematic part of the input with "^" in the *Interactive Console* (Figure 1.5). `POY4` will also automatically display in the *POY Output* window the description of the command, its syntax, and examples of its usage. As explained above, the Up and Down keys can be used to scroll through the output and determine the source of the error. Certain kinds of errors will be reported explicitly (Figure 1.5).

1.9 Obtaining help

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

POY contains a help file that can be accessed by entering `help()` at `POY4` prompt in the *POY Output* window. This file contains descriptions and examples of all currently implemented `POY4` commands. Up and down arrows allow to scroll through the file. 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)`. Help will appear in the upper window, as shown in Figure 1.7.

POY4 Quick Start will help you to get familiar with the appearance and navigation of `POY4` interface, and will provide you with step-by-step instructions on how to run your first analysis.

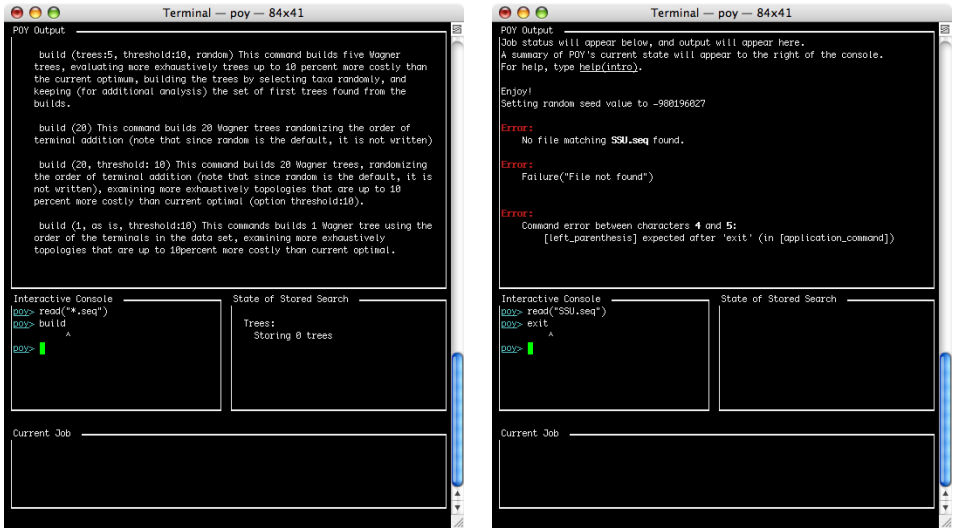

Figure 1.5: Displaying errors. `POY4` displays error messages in several ways. In the example in the left panel, the command `build` was entered without parentheses, which is required for a valid `POY4` command syntax; The exact place of the error is marked by "`^`", in this case following the `build` 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*).

POY Commands Reference contains detailed descriptions of all POY4 commands and arguments, together with comprehensive examples of their usage. This is an essential document that will provide you with necessary information to fully customize every stage of your analysis.

POY Book (Wheeler et al., 2006 *Dynamic Homology and Phylogenetic Systematics: A Unified Approach Using POY*) 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.

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`.

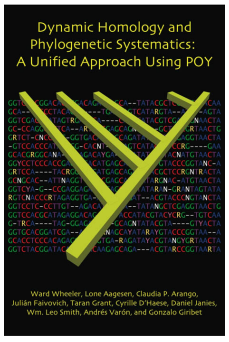

Figure 1.6: The POY4 Book.

The *POY Book* and the document containing the *POY4 QuickStart*, *POY4 Command References*, and *POY4 Tutorials* can be downloaded as PDF files from POY4 web site at

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

1.10 Exiting

To finish a POY4 session, enter command `exit()` (Figure 1.7) or `quit()`. This will close the POY4 interface and resume the Terminal window (Mac) or the Command Prompt window (Windows).

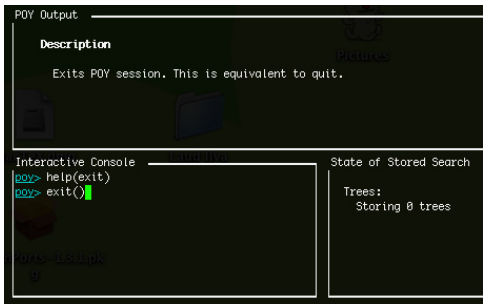

Figure 1.7: Exiting POY4

1.11 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, and, additionally, provide a forum for discussing specific issues using POY4, and present an efficient way to communicate with POY4 developers regarding any technical difficulties, reporting bugs, and obtaining help.

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 Mail Group informs registered users via email of new developments, such as new versions and updates. It also provides a way for reporting bugs and other problems with POY4 and its documentation, as well as an additional resource for obtaining help on specific issues. 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, will 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

1.12 Using POY4

This section will help you get started using POY4 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 read (import) 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 datalfiles are used. These sample files can be downloaded from
http://research.amnh.org/scicomp/projects/poy.php:

- 18s.fas contains unaligned DNA sequence data for a single locus (partial 18S ribosomal DNA) in FASTA format.
- 28s.fas contains unaligned DNA sequence data for a single locus (partial 28S ribosomal DNA) in FASTA format.
- morpho.ss contains a morphological data matrix in Hennig86 format.

Once POY4 has been launched and the interface (Figure 1.3) had appeared on the screen, the data can be imported 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 1.9 to learn more about POY4 commands and their arguments).

1.12.1 Importing data

The basic command to input data in POY4 is read(), which includes the list of files (in quatation 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. We can issue a pair of read() commands (Figure 1.8):

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

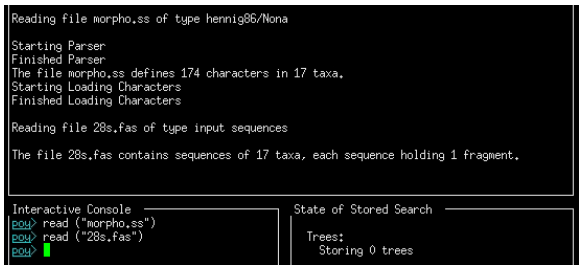

Figure 1.8: Importing datafiles. Two consecutive `read` commands import both the morphological datafile (`morpho.ss`) in Hennig86 format , and the molecular datafile (`28s.fas`) in Fasta format. Note that `POY4` automatically reports in the *POY Output* window the names and types of files that have been imported.

The syntax of `read` is not unique; in fact, every command in `POY4` contains two elements: the name of the command (e.g. `read`), followed by the 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). Eventhough arguments there might be only one argument or it might be absent or omitted 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 Reference* document.

Most of the time users are interested in importing multiple datafiles to analyze on the 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 at a time as was shown previously (Figures 1.8 and 1.9).

Figure 1.8 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 is *added* to the current data in memory, it *does not replace it*. 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 in subsequent operations, despite the fact that the trees were generated only

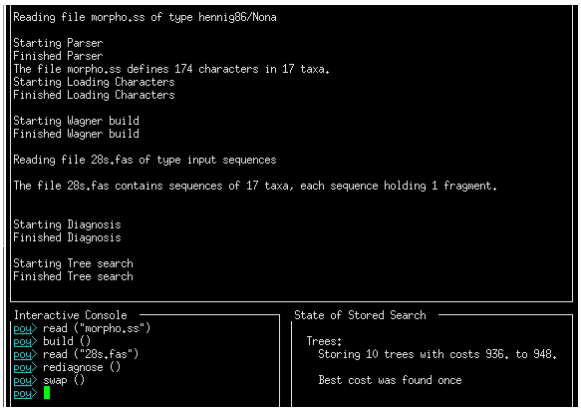

Figure 1.9: Building trees with morphological data only but continuing analysis using combined morphological and molecular data. This example shows how we can add data to the analysis incrementally by loading files at different points in the search. First, the morphological data are imported from `morpho.ss` 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.

from morphological data (Figure 1.9):

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

It must be noted that if the number of terminals differs among datafiles, only the data that corresponds 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.

Also, because `POY4` appends trees and data in memory, it is a good practice to empty the memory when starting a new analysis using use the `wipe()` command (see also `clear_memory()`).

Valid input files include nucleotide and amino acid sequence files in many formats, and morphological data in Hennig 86 format. For information on specific formats supported by POY4 and other types of input files see `help(read)`.

1.12.2 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* window as shown in Figure 1.8. However, it may be desirable to inspect the imported data in greater detail to ensure that the format and contents of the files has been interpreted correctly. This practice allows to avoid common errors, such as misspelled 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 type of characters, the list of files that have been imported so far, and the lists of terminals and characters excluded from the analysis. For example, to produce such report of the same 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 1.10. Obviously, the entire report will not be visible in the *POY Output* window. Therefore, the Up and Down arrow keys should be used to scroll through it.

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

By default, POY4 reports the results of executed commands in the *POY Output* window. However, the same output can be redirected to a file simply by adding the name of the output file in the list of argument of the command `report()` *before* the argument that specified 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 write `report("data_analyzed.txt", data)`.

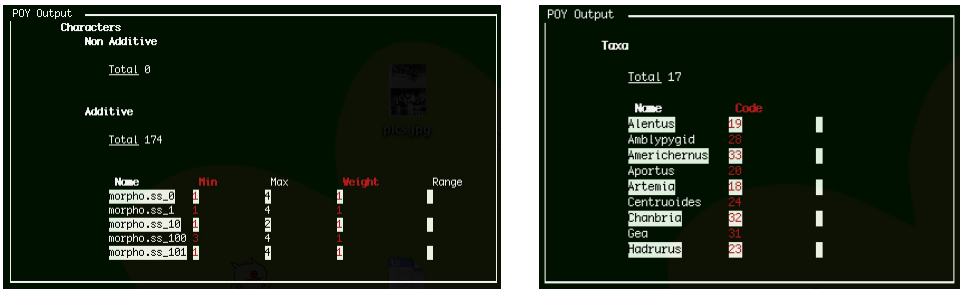

Figure 1.10: Inspecting imported data. The figure shows segments of a data report generated by `report(data)`. The left and right panels demonstrate a typical table output the character and terminal data respectively.

Another useful argument of `report` is `cross_references`. It displays which terminals are present or absent in each one of the imported files, providing a comprehensive visual overview of the 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 1.11.

1.12.3 Building initial trees

The command to build trees is `build` (already been mentioned in Section 1.12.1). After importing `morpho.ss` and `28s.fas`, executing the command `build()` without specifying any arguments will generate 10 Wagner trees by random addition sequence (the default setting of the command). Make sure that if you plan to build trees based on other data you have to purge the memory first by using `wipe()` 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 Reference*; see Section 1.9.) If you would like to build more trees, 100 for instance, 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`; therefore, `build(trees:100)`

Figure 1.11: 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 datifie. In this example, the data for all the the taxa listed in the *POY Output* window are present both datafiles, `morpho.ss` and `28s.fas`.

is equivalent to `build(100)`. As defaults, the shortcuts are fully described in the *POY Commands Reference*. 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 1.12). It 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 a current tree (recalculated after each terminal addition), and the estimated time (in seconds) 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 1.12: 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 finished, 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`).

1.12.4 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 iteratively. As with other commands, the arguments of `swap()` allow customization of the performance of the command. In case of `swap()`, additional options specify which algorithms are used in swapping and restrict swapping to certain sections of trees. In the following example, branch swapping is performed under the default settings on each of the 100 trees build in the previous step:

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

Branch swapping is performed sequentially on all trees stored in memory. During swapping, the *Current Job* window reports the number of a 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 1.13). When the process is complete, the *State of Stored Search*

Figure 1.13: Performing a local search. When searching (left panel), the *Current Job* window reports the number of the tree that is currently being analyzed (`73 of 100`), a method of branch swapping (`Alternate`), a function being currently performed (`SPR search`), and a cost of the current tree (`456`). When the searching is finished (right panel), the *State of Stored Search* window displays the best (`446`) and worst (`463`) costs, the number of trees stored in memory (`100`), and the number of trees of the best cost (`9`) recovered from independent tree builds. Note these trees may not necessarily have unique topology.

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 1.13). 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 the arguments of `swap()` allows to design 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 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:10, threshold:10)`. Other options provide the means to sample trees as they are evaluated, timeout after certain number of seconds, transform the cost regime, and other perform other functions in conjunction with other `POY4` commands.

1.12.5 Selecting trees

Having performed the basic steps of importing character data, building initial trees, and conducting a local search, we obtained a set of local optimum trees in memory. Most of the time, a user would like to select only those

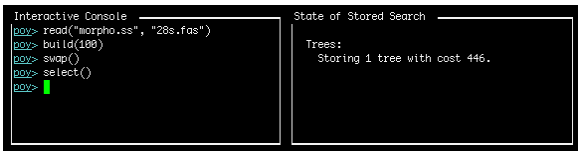

Figure 1.14: Selecting unique best trees. Executing `select()` keeps only unique tees of best cost. The *State of Stored Search* window reports that there is only one unique tree of best cost (`446`).

trees that are optimal and topologically unique and 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()
```

will select only unique trees of best cost; the remaining trees will be deleted from memory. The *State of Stored Search* window will report the number and the cost of the best tree(s) (Figure 1.14).

`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 1.13) and after (Figure 1.14) executing `select()` shows that swapping on 9 of 100 initial trees produced the trees of best cost (`446`), but these trees are identical, because only one was retained when filtered using `select()`.

1.12.6 Visualizing the results

There are several ways to visualize results. A quick way to see the tree(s) on screen is to use the command `report(asciitrees)` that will draw a cladogram in the *POY Output* window (Figure 1.15). The ascii tree can also be reported in a file, if the output file name is specified (in parentheses and separated from the argument `asciitrees` by a comma). However, for reporting trees to a file there are better options. First, the command `report("my_first_tree", graphtrees)` will output a cladogram in postscript format (Figure 1.15) which can be edited using graphics software (such as Adobe

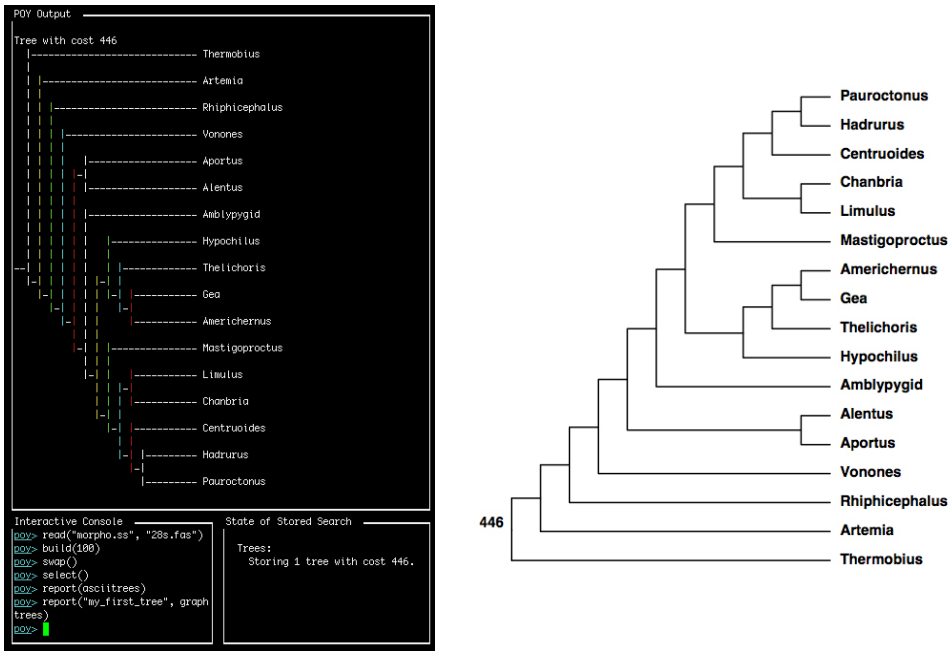

Figure 1.15: Visualizing trees. An ascii tree (left) is generated using the command `asciitrees`. The same tree is reported to a file in a postscript format (right) using `report("my_first_tree", graphtrees)`. Note that both representations of trees are preceded by their costs.

Illustrator or Corel Draw). (`POY4` will also append the "ps" extension when generating graphic output to a file.)

`report("my_first_trees", trees)` will report the trees in memory to the file `my_first_trees` that can be imported in other programs (such as TNT). Other supported tree tree formats include Newick and Hennig86. `report()` can also generate consensus trees in the graphical (postscript) format when appropriate arguments are specified (for example, `report("strict_consensus", graphconsensus)`).

1.12.7 Running scripts

So far, we have communicated with `POY4` interactively, by executing commands from the *Interactive Console* window. Another way of conducting an analysis is to run a script: a file containing the list of commands to be performed (Figure 1.16). This allows a user to design a search strategy in advance and run the entire script using a single command, `run()` or, at start

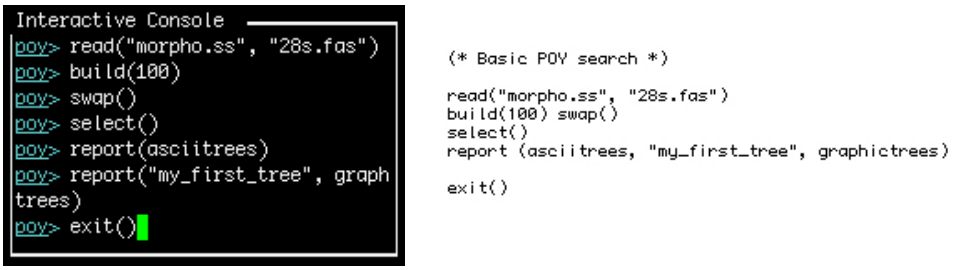

Figure 1.16: Using POY4 scripts. Executing the list of commands the *Interactive Console* (left) is equivalent to running a script containing the same list. Note, that the header of the script is a comment, inclosed in "(* *)", that is ignored by POY4. Also note, that commands can either be listed in 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.)

up, by entering the names of the script files (without quotes and parentheses). This is extremely useful in cases when operations take may take long time and you do not want to sit in front of the computer waiting for a part of your analysis to finish in order to execute the next command. Another advantage of using scripts is that it can contain comments that are ignored by POY4 but can be helpful to describe the contents of the files used and provide any other annotations. The comments are enclosed in parenthesis *and* asterisks. For example, `(*this is a comment*)`. Comments can also be entered interactively from the *Interactive Console*. Their utility in that context is, however, limited unless the comments are featured in some output files.

There are two ways to run a script:

1. From the *Interactive Console* use the command `run("script.txt")`, where `script.txt` is the name of the file containing the script.

2. From the command line you use to start POY4, include the filename of the script, as in `poy script.txt`.

In both cases, the script must include the command `exit()` at the end to finish the session, otherwise POY4 will wait for further instructions to be entered after executing the script's contents.

Enjoy POY4!

1.12.8 Known issues

The following issues are known to occur in this beta release. The core POY4 development group is presently working on fixing these issues.

1. If the window is resized during a search, it will not be updated until the job is finished.
2. The diagnosis report is rudimentary but will be greatly expanded in the next version.
3. The analysis of static homology characters is slow.
4. The output of the flat interface is reported differently than that of the ncurses interface.
5. The parallel version reports the number of jobs currently running on only *one* of the processors, not in all of them.
6. NEXUS files are not always parsed correctly.

Chapter 2

POY4 Commands

2.1 POY4 Command Structure

2.1.1 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 we will call the *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 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`.

2.1.2 Grammar Specification

The following is the formal specification of the valid grammar of a script in `POY4` 4:

```
script: | command
        | command script

command: LIDENT "(" arguments ")"

arguments:  |
            | argument
            | argument "," arguments

argument:   | primitive
            | LIDENT
            | LIDENT ":" arguments
            | command
            | "(" arguments ")"

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 show graphically a typical structure of valid POY4 commands formally defined above. The Figure 2.1 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, inclosed in parentheses and separated by a comma. Note that trees:2 is a labeled-value argument, that is it contains a label (trees) and a value (2) separated by a colon.

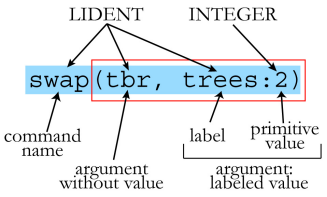

Figure 2.1: A structure of a simple POY4 command. The entire command (highlighted in blue), consists of a command name followed by a list of arguments (enclosed in red box). See text for details.

Figure 2.2 shows a compound command perturb, because the list of its arguments contains another command, swap. This means that executing perturb will perform a set of specified operations that contains a nested set of operations specified by swap. Note also that in contrast to the first labeled-values argument iterations, the second labeled-values argument ratchet has multiple values (a float and an integer), the values 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 if they exist.)

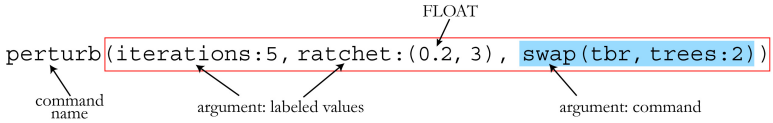

Figure 2.2: A structure of a compound POY4 command. Note that the list of arguments (enclosed in red box) includes a command (highlighted in blue). Also, note that `ratchet` accepts multiple values, a float and an integer, that are inclosed in parentheses and separated by a comma. See text for details.

2.2 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, whereas values of others are optional. In the descriptions of POY4 commands below, the elements of POY4 grammar are emphasized 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]`.
- Primitives 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.

- Some examples of its 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 is context dependent. If arguments are not logically interconnected, their order is not important. For example, commands `build(10,randomized)` and `build(randomized,10)` are equivalent. However, executing 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.

2.3 Command Reference

2.3.1 build

Syntax

build`([argument list])`

Description

Builds Wagner trees [6]. 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. The sequence of taxon addition can either be random or correspond to the order of terminals as they are listed in the first imported datafile. During tree building, POY4 reports in the *Current Job* window of the ncurses interface which of the terminal addition strategies is currently used. 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.

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 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 `report` (Section 2.3.19) that must precede the subsequent `build` command. This will produce a file containing the trees appended from all builds. 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 first imported datafile, and all others are built using a random addition sequence.

`randomized` Indicates that terminals are added in random order on every Wagner tree built.

`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 without the `trees` label. Therefore, `build(5)` is equivalent to `build(trees:5)`. Note that `trees` is also used as an argument of the command `swap` (Section 2.3.26) but with different meaning.

The value `0` generates no trees, but, unlike other values, it *retains* all trees in memory instead of eliminating them. This is useful, for example, in the `bremer` (Section 2.3.2) 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 2.3.2) support. In other contexts the command `read` (Section 2.3.14) is used.

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

`all` Turns off all preference strategies to make a join, simply try all possible join positions for all terminals added.

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 since random is the default, it is not written).

- `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, instead of simply specifying 20, we use the label trees (this might be desirable to improve a script's readability).

- `build(15, as_is)`
  Builds the first Wagner tree using tree using the order of terminals in as they are listed in the first imported datafile and generates the remaining 14 trees using a random addition sequence.

2.3.2 calculate_support

Syntax

calculate_support`([argument list])`

Description

Calculates the requested support values. POY4 implements support estimation based on resampling methods (Jackknife and Bootstrap) and Bremer support. Calculation of Bremer support presupposes that at least one tree (for which support is estimated) is in memory. All the arguments of calculate_support command are and their order is arbitrary.

NOTE

The placement of the root affects calculation of the support values. Therefore, it is critical to define the root prior to executing calculate_support. See the description of the command set (Section 2.3.24) on how to specify the root.

POY4 does not report support values by default; the output of support values must be requested using report (Section 2.3.19). This is particularly important for Jackknife and Bootstrap support values, as these sampling techniques do not require the existence of trees in memory. Therefore, it is possible to perform the sampling for support values *before* the tree of interest has been found.

NOTE

It is critical to understand that 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 desirable 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 2.3.11) prior to executing `calculate_support`. Alternatively, an output file in the Hennig86 format can be generated based on an implied alignment using `phastwinclad` (Section 2.3.19) that can subsequently be analyzed using other programs such as NONA or TNT.
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, in calculating support values by sampling individual nucleotides based on the static homology data (obtained by `static_approx`) can produce a discrepancy in tree costs when compared to the tree cost based on the original dynamic homology data. Therefore, it is recommended to perform an extra round of swapping on the transformed data to insure that the local maximum is reached for the static homology characters prior to calculating support values.

Arguments

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

`bremer` Calculates Bremer support [2, 10] for each tree in memory by performing independent searches for each node. The parameters for the searches can be modified using arguments described under *Search strategy*. The argument `bremer` takes no values.

`bootstrap[:INTEGER ]` Calculates Bootstrap support [7]. The integer value specifies the number of resampling iterations (pseudoreplicates). If the value is omitted, `POY4` performs 5 pseudoreplicates 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, POY4 uses the default values of the arguments.

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. Alternatively, the initial trees for calculating Bremer support can be imported using the argument of_file of the command build (Section 2.3.1).

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, only one of them is selected at random. If build is omitted from the argument list of calculate_support, POY4 a single random addition Wagner tree per pseudoreplicate by default. This is equivalent to build(trees:1, randomized). See build (Section 2.3.1) for a detailed discussion of arguments of the command build.

swap Specifies the method and parameters for local tree search. If the argument swap is omitted, the search is performed under the default settings of the command swap (Section 2.3.26).

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 loaded 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. 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 1 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 1 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. The procedure is repeated 1000 times.

See also

- `report` (Section 2.3.19)

- `supports` (Section 2.3.19)
- `graphsupports` (Section 2.3.19)

2.3.3 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 2.3.30)

2.3.4 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 enter to the working directory before beginning a `POY4` session. To display the path of the current directory, use the command `pwd` (Section 2.3.12).

Arguments

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

Examples

- `cd ("/Users/username/docs/poyfiles")`
  Changes the current directory to the directory `/Users/username/docs/poyfiles`.

See also

- `pwd` (Section 2.3.12)

2.3.5 echo

Syntax

echo`(STRING, output class)`

Description

Prints a 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 commands does not generate any output.

Arguments

Output class

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

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

`output[:STRING ]` Reports a specified string on screen or to a 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 in the flat interface.

See also

- `report` (Section 2.3.19)

2.3.6 exit

Syntax

exit`()`

Description

Exits a `POY4` session. This command does not have any arguments. `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 2.3.13)

2.3.7 fuse

Syntax

fuse`([argument list])`

Description

Performs Tree Fusing on the trees in memory. Tree Fusing is a genetic algorithm technique that allows the escaping the local optimum 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 determined.

Arguments

`keep:INTEGER` Specifies the maximum number of trees to be kept 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 values 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 command `swap` (Section 2.3.26).

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 swapping (see the argument `constraint` (Section 2.3.26) of the command `swap`).

See also

- `swap` (Section 2.3.7)

2.3.8 help

Syntax

help`([argument])`

Description

Reports the requested contents of the help file on screen.

Arguments

`LIDENT` Specifies a command name, the help for which is requested.

`STRING` Specified the expression (treated as an emacs regular expression), every occurrence of which in the help file is reported on screen.

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.

2.3.9 inspect

Syntax

inspect`(STRING)`

Description

Retrieves the description of a `POY4` file. 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 out in the *POY Output* window of the ncurses interface or 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 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 2.3.21)
- `load` (Section 2.3.10)
- `cd` (Section 2.3.4)
- `pwd` (Section 2.3.12)

2.3.10 load

Syntax

load`(STRING)`

Description

Reads and imports a POY4 file, the name of which (or its path if the file is not located in the current working directory) 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 2.3.21) 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 (important exceptions are out-of-memory errors).

Examples

- `load("initial_search.poy")`
  Reads and imports the contents of the `POY4` file `initial_search.poy`, located in the current working directory (as printed by the `pwd`) command.
- `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 2.3.21)
- `inspect` (Section 2.3.9)
- `cd` (Section 2.3.4)
- `pwd` (Section 2.3.12)

2.3.11 perturb

Syntax

perturb`([argument list])`

Description

Performs a search using a temporarily modified ("perturbed") characters starting with the trees currently in memory. Once a local optimum is found for the perturbed characters, a new round of search 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 the escape from a local 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 [14]. 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). For static matrices, such as those obtained using the command `transform` (Section 2.3.27), `ratchet` randomly selects and reweights individual nucleotide positions (column vectors), as in Nixon's original implementation.

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 2.3.26).

`transform` Specifies a type of character transformation to be performed *before* executing a `perturb` procedure. See the command `transform`

(Section 2.3.27) for the description of the methods of character type transformations and character selection.

Defaults

`perturb(ratchet, swap (trees:1))` By default, `POY4` performs the ratchet procedure under default settings.

Examples

- `perturb(resample:(50,terminals), iterations:10)`
  Performs 10 successive repetitions of random resampling of 50 terminals with replacement. Branch swapping is performed using alternating SPR and TBR, and and keeping 1 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 1 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 1 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 2.3.26)
- `transform` (Section 2.3.27)

2.3.12 pwd

Syntax

pwd`()`

Description

Prints the current working directory in the *POY Output* window of the ncurses interface and the standard error 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 will generate the following message: "The current working directory is /Users/myname/datafiles/". The reported actual 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 2.3.4)

2.3.13 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 command and continuing the session.

Examples

- quit()
  Quits the program.

See also

- exit (Section 2.3.6)

2.3.14 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. 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 2.3.19).

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.

NOTE

Unlike many phylogenetic programs, POY4 does not clear the memory buffer 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 will be assigned missing data for all previously loaded characters. If a taxon does *not* appear in a file, it will be assigned missing data for the characters appearing on it.
If the user wants to eliminate any data read, and then read a new file to be analyzed alone, the `wipe()` command must be issued first.

NOTE

If one of the taxon names in an imported molecular file contains a space, " ", POY4 will issue a warning. This will also happen if a taxon name appears to match a nucleotide sequence.

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 that explicitly denotes the data type, is included; it is followed by a colon (":") and then enclosed in parentheses, the list of data file names, 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.

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

`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 2.3.27) of the command `transform` (Section 2.3.27). Locus homologies are determined dynamically, but based on annotated regions.

`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 2.3.27). Unlike when using `annotated` data type, both locus-level and nucleotide-level homologies are determined dynamically. 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 2.3.27).

`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. Aa extra rightmost column and lowermost row correspond to a gap. It is impotant that the cost matrix must be symmetrical. Here is an example of a valid custom alphabet input file:

```
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 2.3.27) of the command `transform` (Section 2.3.27). Chromosome homologies are determined dynamically using distance threshold levels specified by the argument `chrom_hom` (Section 2.3.27) of `transform` (Section 2.3.27).

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

`prealigned:(read argument, tcm:STRING)` Specifies that the input sequences are prealigned and should be assigned the transformation cost matrix specified in the input file. (See `tcm`.)

`prealigned:(read argument, tcm:(INTEGER, INTEGER))` Specifies that the input sequences are prealigned and should be assigned substitution and indel costs as defined in the tcm argument. (See `tcm`.)

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 (the default), 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 of 1 for substitutions and 2 for indels.

See also

- `report` (Section 2.3.19)

2.3.15 rediagnose

Syntax

rediagnose`()`

Description

Performs a reoptimization 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.

2.3.16 recover

Syntax

recover`()`

Description

Recovers the best trees found during swapping, even if the swap was cancelled. This command functions only if a previously executed (in the current `POY4` session) command `swap` included the argument `recover` (Section 2.3.26). 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 2.3.26)
- `recover` (Section 2.3.26)

2.3.17 redraw

Syntax

redraw`()`

Description

Redraws the screen of the terminal. This command is only used in the ncurses interface, other interfaces will 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.

2.3.18 rename

Syntax

rename`(argument)`

Description

Changes the name of specified items from one string to another. Valid `rename` arguments can be `character`, `terminals`, a filename, or a pair of strings. Specifying `("a", "b")` would rename the character or taxon `a` to `b`.

2.3.19 report

Syntax

report`([argument])`

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()`. See the examples for the file outputting 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)` Given an argument value `(a, b, c)`, report the ratio between the all pairs distance and their maximum length, between the characters listed in `b` and `c`. If `a` is

true, then compute the complement sequence of all the characters in `c` before 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 2.3.23)) 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, *etc.*) with corresponding transformation cost matrix (if specified), 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 for each input file. For information on including and excluding terminals use the command `select` (Section 2.3.23).

`treestats` Reports the number of hits per cost found by `POY4` in a table format.

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

The root can be specified using the command `set` (Section 2.3.24).

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 2.3.23) 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), calculate the cost of the $n-1$ rooted trees as implied by a root located in the subdivision vertex each edge in the unrooted tree in memory.

`asciitrees[:collapse[:BOOL ]]` Draws textual 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 [15]; if integer value is specified, a majority rule consensus is computed, collapsing nodes with occurrence frequencies less than the specified integer [13]. If a value less than `51` is specified, `POY4` will report 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 2.3.2) 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. 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 parenthetical representation of a tree with the support values has previously been calculated using the command `calculate_support` (Section 2.3.2) (however see `bremer`), 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 2.3.26)), 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 needed, then the necessary information must have been calculated using `calculate_support` (Section 2.3.2).

`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. The valid optional arguments are:

- `total` Includes the total cost of a tree in square brackets after each tree
- `_cost` Include the cost of every subtree in the tree in square brackets.
- `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.

`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` then no branches are collapsed.

Implied alignments This set of arguments will output implied alignment [18].

`implied_alignments[:STRING ]` 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 2.3.23). 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[:STRING ]` 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:

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

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 to screen or to a file, if specified.

Other arguments

`ci` Calculates the ensemble consistency index (CI; [5, 12, **?**]) for additive and nonadditive characters. Dynamic homology characters are ignored in calculating the CI, therefore, the dynamic homology characters must be converted to static homology characters using the argument `static_approx` of the command `transform` (Section 2.3.27).

`memory` Reports on screen (or to file, if specified), the statistics of the garbage collector. These statistics are only estimates in the native code version but exact in the bytecode version of `POY4`. Note that the values consider only the OCaml-handled memory, the C data structures (sequences, additive, and non-additive characters), are not included in the reported values. For a precise description of each memory parameter, see the Objective Caml documentation.

`ri` Calculates the ensemble retention index (RI; [5]) for additive and nonadditive characters. Dynamic homology characters are ignored in calculating the RI, therefore, the dynamic homology characters must be converted to static homology characters using the argument `static_approx` of the command `transform` (Section 2.3.27).

`script_analysis:STRING` Reports the order in which commands listed of the imported script (listed in the string argument) are going to be executed. Unlike executing commands sequentially (when entering commands interactively through the *Interactive Console* of the ncurses interface or the flat interface), 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 the memory usage and improves parallelization.

The colored output in the *POY Output* window of ncurses interface facilitates reading the output of `script_analysis`: red lines mark hard constraints that allow neither parallelization nor memory optimizations, blue lines mark nice 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 on screen (or redirected to a file, if specified). 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.

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 (by 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))`
  Output 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 commands reports the requested types of outputs (emphi.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 commands 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 alignment of 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 2.3.4) 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.

See also

- calculate_support (Section 2.3.2)

2.3.20 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. There are no default settings of run.

NOTE

Note that if any of the scripts contain 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 2.3.6)
- quit (Section 2.3.13)

2.3.21 save

Syntax

save(STRING [, STRING])

Description

Saves the current POY4 state 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.

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).

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` to the file `alldata.poy`, which can later be retrieved using the command `inspect` (Section 2.3.9).

- `save("/Users/andres/test/alldata.poy", "My total evidence data")`
  This command performs the same operation is 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 2.3.9)
- `load` (Section 2.3.10)

2.3.22 search

Syntax

search`([argument])`

Description

The command `search` implements a default strategy for a driven search. The command integrates `build`, `transform`, and `swap` commands for an efficient initial search. Tree building and swapping are executed under default settings; the `transform` provides an option of making sequential transformations of characters that substantially speeds up the search, however, at the expense of accuracy in calculating tree cost. The arguments are optional and their order is arbitrary. Even though the entire sequence of the commands can also be specified by setting `build`, `transform`, and `swap` individually to corresponding values, the advantage of using `search` command is that these steps are already predefined.

Arguments

`build[:BOOL ]` Specifies either to build new trees (boolean value `true`) or to use the trees stored in memory (that is do not build new trees; boolean value `false`). The default for `build` is `false`. Therefore, executing `search` under default settings will produce no result if there are no trees in memory.

`transform[:BOOL ]` Specifies to either transform characters as part of the search by sequential execution of commands `auto_sequence_partition` and `auto_static_approx` (both arguments of the command `transform` (Section 2.3.27)) (boolean value `true`) or not (boolean value `false`). This combination of character transformations substantially accelerates the search but at the expense of accuracy in calculating the exact tree cost. The default is `false` (characters are not transformed).

Defaults

`search(build:false, transform:false)` Under default settings, all the trees currently in memory (either from prior builds or imported as tree files) are subjected to branch swapping using alternating SPR and TBR, keeping one tree per swap (the default of `swap`), and without transforming characters.

Examples

- `search(build:true, transform:false)`
  This command builds 10 Wagner trees by random addition sequence (the default of `build`), performs alternating SPR and TBR branch

swapping and keeping one tree per swap (the default of `swap`), and does not transform characters. Because the default of the argument `transform` is `false`, it can be omitted from the list of argument. Therefore this command is equivalent to `search(build:true)`. Note that if currently there are trees in memory, the new trees generated by `search` will replace them.

- `search(transform:true)`
  This command performs branch swapping on the existing trees in memory under default parameters of `swap` as shown in the examples above. The searches are performed on characters transformed using sequential application of `transform` arguments `auto_sequence_partition` and `auto_static_approx` to speed up the swapping procedure.

See also

- `build` (Section 2.3.1)
- `swap` (Section 2.3.26)
- `transform` (Section 2.3.27)

2.3.23 select

Syntax

select`([argument])`

Description

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

Arguments

Select terminals and characters Specifies terminals and/or characters to be used in subsequent analysis. The selection is based on terminal and character names and the naming conventions are shared between both classes. The arguments in this group specify whether terminals or characters are being selected. *Identifiers* are used to specify which specific characters or terminals are being selected, either by listing their names or importing a

file containing a list of terminals or characters (see the *Character and terminal identifiers* argument group below for the description of methods for selecting specific terminals or characters).

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.

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 2.3.25) and use (Section 2.3.28).

terminals Specifies that subsequently listed identifier(s) refer to *terminals* to be selected.

characters Specifies that subsequently listed identifier(s) refer to *characters* to be selected.

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

Character and terminal identifiers *Identifiers* specify which characters or terminals are processed by a command. 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 2.3.19) and transform (Section 2.3.27).

all Specifies all characters or terminals.

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

codes:(STRING 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 2.3.19) 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 excluded from the analysis based on the level of missing data. The integer value sets the minimum percentage of missing data. Terminals or characters that have *more* missing data than defined by the value are excluded from the analysis.

NOTE

For dynamic homology characters, the missing data refers to sequence fragments, whereas for static characters it refers to individual nucleotide positions. Therefore, when excluding terminals with missing data, the resulting set of selected terminals depends on the character type 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.

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

Select trees Selects 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 within 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 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 commands selects only on 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 commands excludes from subsequent analysis all the terminals that have more than 50 percent of characters missing. The lists 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 2.3.23)
- `transform` (Section 2.3.27)

2.3.24 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 file already exists, POY4 will append the text to it; if the file does not exist, then POY4 creates a new file. If the user would like to delete the contents of a preexisting 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 `log`.

`root:LIDENT` Specifies the terminal with with the the output trees are rooted. 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.

Cost Calculation Intensity of the tree cost estimation routines. All these arguments are mutually exclusive: only the last to appear in a `set` command will be used.

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

`exhaustive_do` Use the starndard Direct Optimization algorithm for the tree cost estimation. The difference with `normal_do` is that the calculation of the tree costs during a search are 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`).

Randomized routines

`seed:INTEGER` Sets the seed of the random number generator to the argument's value. If unspecified, `POY4` uses the system's time as seed. It is reported when the program starts.

Defaults

`set()` If no argument are given, the command does nothing.

Examples

- `set(history:1500, seed:45, log:"mylog.txt")`
  This command increases the size of the history in the ncurses interface to 1500 lines, sets the random number generator to `45`, and initiates a log file `mylog.txt`, located in the current working directory, as printed by the command `pwd()`.

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

See also

- `report` (Section 2.3.19)

2.3.25 store

Syntax

store`(STRING )`

Description

Stores 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 2.3.28)
- transform (Section 2.3.11)

2.3.26 swap

Syntax

swap([argument list])

Description

swap is the basic local search function in POY4. This command implements a family of algorithms collectively known as branch swapping in systematics and as hill climbing in combinatorial optimization. They proceed by clipping parts of a given tree and attaching them in different positions of the same tree. It can be used to perform a local search in the set of trees loaded in memory.

Swapping is performed on all trees in memory. During search, swap collects 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 2.3.2).

All arguments of swap are optional and their order is arbitrary.

Arguments

Neighborhood The basic standard procedures for local search in phylogenetic analysis are SPR and TBR [16]. The arguments in this group define the parameters of these methods.

The nearest-neighbor interchanges (NNI) swapping strategy is implemented by combining 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` will stop 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` Similar to `current_neighborhood`, this argument changes the trajectory of a search, by completely exploring the neighborhood of the current tree in memory, and choosing the best swap position among in this neighborhood first before continuing. The default in `POY4` is to choose the first one available that shows a better cost than the current best.

`annealing:(FLOAT, FLOAT)` Uses simmulated annealing [11]. 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 [8]. If the argument's value is (a, b), then `POY4` always accepts a tree with better cost than the current best, with probability a a tree with equal cost, and

with probability $1/b + d$ a tree with cost d greater than the current best.

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 commands select among the different strategies available in `POY4`.

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

`randomized` Chooses edges uniformly at random for breakages.

`distance` Gives higher priority to those edges with biggest 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)]` The constraint argument for the `swap` command sets constraints on the join locations during the search using an input tree 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 will be attempted to produce during swapping. An integer value of `depth` specifies the maximum distance from the break branch to attempt joins. A string value of `file` specifies an input file containing a singe 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 ]` Turn off all preference strategies to make a join, simply try all possible join positions for each pair of clades generated after a break.

`sectorial[:INTEGER ]` Do not join in edges at distance greater 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; it is possible to ask the program to collect information about those trees, to be used later: either to provide backups of the state of a search (in the improbable case that `POY4` crashes), or simply to analyze the characteristics of the alignments. A difference with other groups of arguments in `swap`, is that the user can choose any combination of trajectory samples, and they will all be used during the search. None of the trajectory samples is used by default.

`recover` Store the current best tree in memory to recover it in case of failure (default is off); If it is necessary to recover such trees after an aborted command, use the command `recover` (Section 2.3.16). If the program terminates normally, the stored trees are exactly those produced at the end of the `swap`. Using `recover` will require twice as much memory as the `swap` would take without it; however, it will not affect the application's performance.

`timeout:INTEGER` is the number of seconds after which the swap will be cancelled. Use this argument in association with `recover` to keep the best trees found up to n seconds after starting the search.

`timedprint:(INTEGER, STRING) timedprint:(n, "trees.txt")` will print 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.

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

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

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

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

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

Defaults

`swap(trees:1, alternate, threshold:0, bfs)` By default, current trees are submitted to a round of SPR followed by TBR using breath first search under default setting, and keeping one best tree per each starting tree.

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

- `swap(constraint:(file:"bleh"))`
  Read 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))`
  Use the tree in file "bleh" as constraint tree, and join at distance at most 4 from the breaking branch during the swap.

See also

- `transform` (Section 2.3.27)

2.3.27 transform

Syntax

transform`([argument list])`

Description

Transforms a character or a list of 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 (and vise versa) 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 2.3.23). 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 2.3.23) 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 2.2).

`auto_static_approx` Evaluates each loaded 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 [18]. If no characters are specified (using identifiers), all sequence fragments are evaluated. This method greatly accelerates searching.

`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 accelerate searching.

`fixedstates` Transforms the characters specified in fixed state characters [17] with distances equal to the edition distance between their observed values. By default, the application of `fixedstates` transforms all

molecular characters. To specify a subset of characters, an identifier must be used in conjunction with `fixedstates`. For example, `((names:("s_[0-5]"), fixedstates))`.

`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 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 $a + tcm(s[0]) + tcm(s[1]) + ... + tcm(s[n-1]) + tcm(s[n])$.

`multi_static_approx` Calculate 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 [19]. This option is intended only for heuristic search purposes.

`prealigned` Assume that the sequences are prealigned and should use the cost regime specified in their assigned transformation cost matrix. All other cost parameters are ignored, including affine gap costs.

`static_approx[:LIDENT ]` Transforms the sequences to the static homology characters corresponding to their implied alignments and their transformation cost matrix [18]. 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`.

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 to calculate jackknife or bootstrap support values based on sampling of individual nucleotides, all commands executed subsequently will be applied to the transform data.

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`.

`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`.

`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 transformation cost matrix by importing a file (specified by the string value) that contains a user defined nucleotide transformation cost matrix. 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. 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:FLOAT` Changes the cost of specified characters by a constant value (weight) specified by the float value.

`weightfactor:FLOAT` Changes the cost of specified characters by a multiplicative factor (weight factor) specified by the float value.

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 at the locus-level in a chromosome and chromosome level in a genome.

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

`seq_to_breakinv:([argument list])` Transforms custom_alphabet character type into 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 once is interested in detecting potential change in their order on a chromosome. See the command `read` (Section 2.3.14) for the description on how to load a 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, using the argument `dynamic_pam`.

`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.* breakpoint, locus–insertion, and locus-deletion costs) can be specified by the argument `dynamic_pam`. If no `dynamic_pam` values are specified, its default values are applied. The optional list of arguments includes the arguments of the `dynamic_pam` that can also be specified subsequently, as a separate step, using the argument `dynamic_pam`.

`dynamic_pam:([argument list])` Specifies parameters for creating chromosome- and genome-level HTUs (medians). The argument values of `dynamic_pam` specify the method if calculating distance between pairs of chromosomes (`inversion` and `breakpoint`), costs of locus-level events (`inversion`, `breakpoint`, `locus_indel`), take into account whether the chromosome is linear or circular (`circular`), and implement a number of heuristic procedures to accelerate computations when working with chromosome data type (`seed_length`, `median`, `swap_med`, `rearranged_len`, `approx`). Under default settings, the distance between two chromosomes is calculated using `breakpoint` and the rest of the arguments are executed under their default settings.

NOTE

Note that the arguments `breakpoint` and `inversion` are *alternative* methods of calculating distance between two chromosomes. Therefore, they cannot be used simultaneously. If both arguments are specified, the latter will be executed. The order of other arguments of `dynamic_pam` is arbitrary.

`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 takes into account rearrangements but not inversions. Note, that this argument *cannot* be used in conjunction with `inversion`. The default value of `breakpoint` is `50`.

`circular:BOOL` Specifies if chromosome is circular (boolean value `true`) or linear (boolean value `false`). The default value of `circular` is `false` (linear chromosome).

`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. The default value of `chrom_breakpoint` is `10`.

`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/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: $o + (l - 1) \times e$. The default values are $o = 10, e = 1.0$.

`inversion:INTEGER` Calculates the inversion distance [9] between two chromosomes given the cost for inversion specified by the integer value. The inversion distance takes in consideration rearrangements and inversions. Note, that 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 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: $o + (l - 1) \times e$. The default values are $o = 10, e = 1.0$.

`median:INTEGER` Specifies the number alternative locus- and chromosome-level rearrangements of the best cost selected (randomly) for each HTU (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.

`seed_length:INTEGER` Specifies the minimum length of identical (invariant, completely conserved) contiguous sequence fragments during comparison between two chromosomes. The integer value of `seed_length` is the number of nucleotides. Correct identification of such fragments facilitates detecting chromosome rearrangement events and accelerates other operations (such as tree building and swapping). However, if `seed_length` value is set too low (allowing for detection of short, multiple fragments that are likely to occur frequently in a genome) or if it is set too high (that might result in no identical fragments detected), the speed of subsequent procedures can potentially decrease. The optimal parameters depend on specifics of a given dataset. The default value of `seed_length` is `9`.

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

`rearranged_len:INTEGER` Two seeds are said to be *non-rearranged*, if their distance is not greater than a predefined threshold *rearranged_len*. In other words, it is unlikely that rearrangement operations can occur between seeds if they are connected. The default value is `1000`

`swap_med:INTEGER` Specifies the maximum number of swapping iterations per each HTU (median) to search for best pairwise alignment of two genomes taking into account locus-level rearrangement events. Limiting the number of swapping iterations acceler-

ates 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)))`
  is equivalent to transform ((dynamic, tcm:(1,1)))

- `transform(tcm:(1,1), gap_opening:1)`

- `transform(tcm:(2,2), ti:(1,1,1,1,0), td:(1,1,1,1,0))`
  will assign 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 will reweight 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))`
  If all the weights are intended to be the same (4), for all the static homology characters.

- `transform((dynamic, weight:4))`
  If all the weights are intended to be the same (4) for all the dynamic homology characters.

- `transform((all, tcm:(1,1)), (names:("gen1", "gen2"), static_approx), (names:("gen3"), tcm:"molmatrix"))`
  Applies tcm (1,1) to all characters, then applies static approx 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:("gen 1:3", "gen2:10", "gen3:1", "gen4:5"), static_approx), (names:("gen5", "gen6"), tcm: "Molmatrix1"))`
  Applies tcm (1,1) to all characters, then applies static approx using that tcm to sequence fragments in files gen1, gen2, gen3, and gen4, and for files gen5 and gen6, it invokes a different transformation cost matrix, contained in the file molmatrix.

- `transform(fixedstates)`

- `transform((names:("s_[0-5]"), fixedstates))`

- `transform((all, seq_to_breakinv:()))`
  In this example all sequence data is transformed into breakinv data type under default settings of `dynamic_pam`.

- `transform(seq_to_chrom:(circular:true, locus_indel:(50, 1.0)))`
  All applicable (*i.e.* sequence) data is transformed into chromosome data, which is treated as a circular chromosome, and settings locus-level gap opening cost at `50` and gap extension cost at `1.0`.

- `read (chromosome:("mito"))`
  `transform((all, dynamic_pam:(breakpoint:10, rearranged_len:60, median:1, circular:false)))`
  This example shows a file read ("mito") containing mitochondrial chromosome sequences that is transformed to set the breakpoint cost at 10, 60 or more nucleotides are necessary to allow rearrangement between 2 identified seeds, the number of median swap passes at 1, and the chromosomes are linear.

2.3.28 use

Syntax

use(STRING)

Description

Loads a saved POY4 state from memory. It replaces the current one. See the usage in store.

See also

- store (Section 2.3.25)

2.3.29 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 ()

2.3.30 wipe

Syntax

wipe()

Description

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

Examples

- `wipe ()`

Chapter 3

POY4 Tutorials

These tutorials are intended for new POY4 users as well as for users who have previously used POY3 and are now upgrading to POY4. The command structure has been overhauled in POY4 to allow greater user control. Although POY3 users will find the changes challenging initially (as did we), the new structure is more intuitive, flexible, and powerful. For details, see the POY4 Commands Reference; for a quick overview see the POY4 Quick Guide. The following tutorials are intended to facilitate the transition to the new command structure by providing examples of analyses that are loosely based on the commands covered in the tutorials found in Chapter 14 in Wheeler et al. (2006). For a fuller discussion of analytical strategies, refer to:

http://homepage.mac.com/wmleosmith/homepage/datafiles/joy/joy.html

and Chapter 9 in Wheeler et al. (2006). In addition to the POY4 executable, the only other application software you need for these tutorials is a text editor, such as Textpad, BBEDIT, EMACS, Word, WordPad, or NotePad. Additionally, a postscript (.ps) viewer/converter (e.g., online at http://pdf.sesse.net/ , Adobe Acrobat Distiller, Apple Preview) will be required to view publication-quality postscript cladograms.

The commands in the following tutorials can all be written and executed separately from the command line, saved and run as a script file, or written into a text file and copied and pasted into POY4's interactive console or flat interface. Initially, we recommend writing and executing them separately, as presented below, as this will allow you to see how POY4 reports progress as you start to use POY4. We also recommend running these tutorials using the graphical interface (ncurses version) of POY4, although they can also be run using the flat interface.

Preliminary tasks:

1. Make sure you have downloaded all of the POY4 tutorial datafiles.
2. Create a folder named tutorial. Although you can place this folder anywhere, we suggest you make it a subfolder to the POY4 folder you created during installation. In addition, nothing depends operationally on this new folder being named tutorial.
3. Review the POY4 Quick Guide for instructions to install and run POY4 using Windows, Mac OSX, and Linux operating systems.
4. Either copy the POY4 executable from the POY4 folder to the new tutorial folder or make sure the executable is in /usr/local/bin/ or another directory included in your path (Linux and Mac OSX only).
5. Copy all of the files from the POY4 tutorial folder to the new tutorial folder.

3.1 Basic Search

This tutorial illustrates the basics of running POY4 using single input datafile.

1. Navigate to the tutorial folder.
2. Launch POY4 (review POY4 Quick Guide for instructions for your platform).
3. Type:

```
read ("mol1.txt") [enter]
build (100) [enter]
select (unique) [enter]
swap (threshold:10) [enter]
select () [enter]
report (asciitrees) [enter]
report ("tutorial1_trees.txt", trees) [enter]
report ("tutorial1_stats.txt", treestats) [enter]
```

   The above commands will perform the following tasks using default parameters:

   - Import the DNA sequence datafile mol1.txt.

- Generate 100 random addition sequence Wagner trees.
- Discard duplicate trees.
- Alternate SPR and TBR branch swapping of all current trees as well as all new trees found that are up to 10% longer than the current tree being swapped. Note: Threshold during swapping is equivalent to slop in `POY3`. Given the faster algorithms and better heuristics of `POY4` it is likely to be less important than it was in `POY3`.
- Discard suboptimal and duplicate trees (i.e., retain only optimal trees).
- Draw optimal trees in `POY4` Output window (ncurses version only) or output file (non-ncurses version), reporting the cost of each tree.
- Output all current trees to file tutorial1_trees.txt.
- Output basic tree statistics to file tutorial1_stats.txt.

4. View cladogram(s) in parenthetical format by opening tutorial1_trees.txt in your chosen text editor.

5. View basic tree statistics by opening tutorial1_stats.txt in your chosen text editor.

3.2 Advanced Search I: Multiple Datasets and Data Types, Equal Weighting, Rooting, and Publication Quality Trees

This tutorial builds on Tutorial **??** to include multiple input datafiles and multiple data types (i.e., genotypic and phenotypic) using equal weighting, designate the root, and automatically generate publication quality trees.

1. Type:

```
read ("mol2.txt", "morph.txt") [enter]
set (root: "Hagfish") [enter]
transform ((all, tcm:(1,1))) [enter]
build (100) [enter]
select (unique) [enter]
swap (threshold:10) [enter]
```

```
select () [enter]
report ("tutorial2", graphtrees) [enter]
report ("tutorial2_trees.txt", trees) [enter]
report ("tutorial2_stats.txt", treestats) [enter]
```

The above commands will perform the following tasks using default parameters:

- Import the DNA sequence datafile mol2.txt and the phenotypic datafile morph.ss.
- Designate the Hagfish as the root.
- Set indel and substitution costs to 1 (equal weighting). The additivity (ordering) of the phenotypic characters is determined in the ccode of the NONA file morph.ss. By default, the transformation cost for phenotypic characters is 1.
- Generate 100 random addition sequence Wagner trees.
- Discard duplicate trees.
- For each tree in memory, alternate SPR and TBR branch swapping of the tree and all trees found within 10% of the cost of the tree.
- Discard suboptimal and duplicate trees (*i.e.*, retain only optimal trees).
- Output publication-quality cladogram(s) of optimal tree(s) to a postscript file tutorial2.ps. (Note: `POY4` automatically adds the `.ps` extension.)
- Output all current trees to file tutorial2_trees.txt.
- Output basic tree statistics to file tutorial2_stats.txt.

2. View publication-quality cladogram(s) by opening tutorial2.ps in your chosen postscript viewer.

3. View cladogram(s) in parenthetical format by opening tutorial2_trees.txt in your chosen text editor.

4. View basic tree statistics by opening tutorial2_stats.txt in your chosen text editor.

3.3 Advanced Search II: Tree Fusing and Fragment Ratcheting

This tutorial builds on Tutorial 3.2 to use tree fusing and fragment ratcheting that help to escape suboptimal islands. This tutorial also introduces matrix and tree output in NONA format.

1. Type:

```
build (100) [enter]
select (unique) [enter]
fuse () [enter]
select () [enter]
perturb (iterations:10, ratchet:(0.2,5)) [enter]
select () [enter]
swap (threshold:10) [enter]
select () [enter]
report ("tutorial3", graphtrees) [enter]
report ("alignment3.ss", phastwinclad) [enter]
```

   The above commands will perform the following tasks using default parameters:

   - Generate 100 random addition sequence Wagner trees.
   - Discard duplicate trees.
   - Perform cladogram searching by fusing one pair of trees in memory. Note: Tree fusing requires at least two trees; if there was only one tree in memory, an error message will be generated.
   - Discard suboptimal and duplicate trees (*i.e.*, retain only optimal trees).
   - Perform 10 successive repetitions of a fragment-based ratchet by randomly selecting 20% of the fragments and upweighting them by a factor of five.
   - Discard suboptimal and duplicate trees (i.e., retain only optimal trees).
   - For each tree in memory, alternate SPR and TBR branch swapping of the tree and all trees found within 10% of the cost of the tree.

- Discard suboptimal and duplicate trees (*i.e.*, retain only optimal trees).
- Output publication-quality cladogram(s) of optimal tree(s) to a postscript file tutorial3.ps. (Note: `POY4` automatically adds the `.ps` extension.)
- Output NONA file with the implied alignment for one (if multiple) optimal trees and all optimal trees in the file alignment3.ss.

2. View publication-quality cladogram(s) by opening tutorial3.ps in your chosen postscript viewer.

3. View implied alignment, optimal tree(s), and manipulate data by opening the file alignment3.ss in WinClada or MacClade.

3.4 Advanced Search III: Input Trees and Step Matrices

This tutorial builds on Tutorials 3.2 and 3.3, and introduces the use of input trees and complex differential cost step matrices.

1. Type:

```
read ("trees.txt") [enter]
transform ((all, tcm:"g4ts1tv2.txt"),
(static, weightfactor:4)) [enter]
swap (threshold:10) [enter]
select () [enter]
report ("tutorial5", graphtrees) [enter]
```

   The above commands will perform the following tasks using default parameters:

   - Read cladograms from the input file trees.txt.
   - Apply specified transformation costs to data. The transformation cost matrix in the file g4ts1tv2.txt is applied to the dynamic homology characters (i.e., unaligned DNA sequences) to assign indel events a cost of 4, transitions 1, and transversions 2. All static homology characters (i.e., the phenotypic characters in the file morph.txt) are upweighted by a factor of 4, with additivities

and previous relative weights unchanged (e.g., a character with transformations costs of 2 would be now have a transformation cost of 8).

- For each tree in memory, alternate SPR and TBR branch swapping of the tree and all trees found within 10% of the cost of the tree.
- Discard suboptimal and duplicate trees (i.e., retain all most optimal trees).
- Output publication-quality cladogram(s) of optimal tree(s) to a postscript file tutorial5.ps. (Note: `POY4` automatically adds the .ps extension.)

2. View publication-quality cladogram(s) by opening tutorial4.ps in your chosen postscript viewer.

3. View cladogram(s) in parenthetical format by opening tutorial4_trees.txt in your chosen text editor.

4. View basic tree statistics by opening tutorial4_stats.txt in your chosen text editor.

3.5 Support I: Bremer Support

This tutorial builds on the previous tutorials to illustrate Bremer support calculation.

1. Type:

```
transform ((all, tcm:(1,1)), (static, weightfactor:1)) [enter]
swap () [enter]
calculate_support (bremer, build (trees:5), swap (trees:2)) [enter]
report (supports) [enter]
```

The above commands will perform the following tasks using default parameters:

2. Apply a weight of one to all transformations.

3. For each tree in memory, alternate SPR and TBR branch swapping of the optimal tree(s).

4. Estimate Bremer support by using inverse constraints, doing five independent searches for every group, holding a maximum of two trees.
5. Output support values for each group in parenthetical notation to POY4 output window.

3.6 Support II: Bootstrap Support Using Dynamic Homology

This tutorial builds on the previous tutorials to illustrate the calculation of bootstrap frequencies. As discussed in the POY4 Commands Reference, the characters sampled during pseudoreplicates are entire fragments of DNA sequences, not individual nucleotide characters. Tutorial 7 shows how to estimate bootstrap frequencies using static homology, which allows nucleotide-level characters to be sampled.

1. Type:

```
calculate_support (bootstrap: 100, build(trees:2),
swap(trees:1)) [enter]
report (supports) [enter]
```

   The above commands will perform the following tasks using default parameters:

   - Perform 100 pseudoreplicates by sampling characters with replacement, doing two independent searches for each pseudoreplicate and holding a maximum of one tree.
   - Output bootstrap frequencies for each group in parenthetical notation to POY4 output window.

3.7 Support III: Bootstrap Support Using Static Homology

This tutorial builds on the previous tutorials to illustrate the calculation of bootstrap frequencies. Here, bootstrap frequencies are obtained from analysis of the implied alignment of static homologies, which permits individual nucleotide- level characters to be sampled instead of whole fragments, as is done using dynamic homology.

1. Type:

```
transform ((all, static_approx)) [enter]
calculate_support (bootstrap: 100, build(trees:2),
swap(trees:1)) [enter]
report (supports) [enter]
```

The above commands will perform the following tasks using default parameters:

- Generate the alignment implied by the optimal tree and given the assumed transformation costs.
- Perform 100 pseudoreplicates by sampling characters with replacement, doing two independent searches for each pseudoreplicate and holding a maximum of one tree.
- Output bootstrap frequencies for each group in parenthetical notation to `POY4` output window.

3.8 Chromosome Analysis I: 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.

1. Type

```
wipe () [enter]
read (chromosome:("mit5.txt"))  [enter]
transform ((all, dynamic_pam:(inversion:15, locus_indel:(10, 1.5),
median:  3, swap_med:5, circular:true, approx:true))) [enter]
build (5) [enter]
swap ()[enter]
select () [enter]
report (asciitrees, diagnosis) [enter]
transform ((all, dynamic_pam:(inversion:15, locus_indel:(10, 1.5),
median:3, swap_med:5, circular:true, approx:false))) [enter]
swap ()[enter]
select ()[enter]
report (asciitrees, diagnosis) [enter]
```

The above commands will perform the following tasks using default parameters:

- Clear all data and trees from memory.
- Import datafile mit5.txt.
- Treat all data as pertaining to unannotated chromosome data, setting the following parameters: inversion distance and cost 15, locus indel 10 + 1.5 times the length (number of nucleotides) of the locus, keep three candidate medians, swap on medians for 5 rounds, treat as circular chromosome, and use fixed-states optimization to approximate chromosome medians.
- Generate 5 random addition sequence Wagner trees.
- Alternate SPR and TBR branch swapping of each tree in memory.
- Discard suboptimal and duplicate trees (i.e., retain only optimal trees).
- Draw optimal trees in POY4 Output window (ncurses version only) or output file (non-ncurses version), reporting the cost of each tree.
- Output the optimal median states and edge costs.
- Treat all data as pertaining to unannotated chromosome data with parameters as above but using optimization alignment (not fixed-states) to approximate chromosome medians.
- Alternate SPR and TBR branch swapping of each tree in memory.
- Draw optimal trees in POY4 Output window (ncurses version only) or output file (non-ncurses version), reporting the cost of each tree.
- Output optimal median states and edge costs.

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. The retention index and the rescaled consistency index. *Cladistics*, 5:417–419, 1989.

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

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

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

[9] 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.

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

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

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

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

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

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

[16] 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.

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

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

[19] 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.

General Index

POY 3.0 Command Line Index