POY 4.0 Tutorials: general commands

Andrés Varón Megan Cevasco

July 25, 2008

1 Using the Interactive Console

In this tutorial you will learn how to move comfortably in the interactive console of POY. To begin, open POY and from the `Analyses` menu select `Run Interactive Console`.

1. A new terminal window should appear and POY should be running in it. If you are using Mac OS X or Linux, resize it to your preference. If you are in windows, unfortunately the terminal can not be resized.

2. Now click anywhere in the terminal, and type the command `help ()`, then press enter.

3. The interactive console is composed of four different frames. All the commands you type will appear in the interactive console frame.

4. Now type the command, `report (data)`. You will see new information appearing in the output frame.

5. If during an interactive session, you would like to repeat a previous command, press the combination keys Control-p (which we will from now and on represent as `<C-p>`). You will see that the last command `report (data)` appears on screen. Using the arrows and the delete keys, modify that command into `report (treestats)`. Now press enter.

6. A lot of text will have appeared in the output window. In order to scroll up, simply use the up and down arrow keys. The screen should move one line at a time, letting you see the contents of that frame.

7. Scrolling line by line can take a long time, using the page-up and the page-down keys, the program will scroll up and down, one page at a time, in the output window.

8. First use the `<C-p>` combination to go back to the `help ()` command. Then go back to `report (diagnosis)` using `<C-n>`. `<C-n>` lets you go *forward* in your command history. Once in the `report (diagnosis)` command, press enter.

9. Typing commands can take a long time, and typos will lurk in. To avoid them, you can use the Tab key let POY autocomplete things for you. For example, type the following (every occurrence of `<TAB>` means "press the tab key"):

```
tra<TAB><TAB><TAB><TAB>(static_<TAB>)
```

   You will see how POY auto-completes the commands, and cycles on all possible matches when more than one possible command is found. The program will also autocomplete filenames for you.

10. Finally, if in the following tutorials, you want to cancel a command that is being executed, simply press `<C-c>` (Control-c). That will roll back the program to the previous interactive state (i.e. the last time you pressed the `<ENTER>`).

We are ready to work with actual data. You can get it from `ftp://ftp.amnh.org/pub/group/molecular/poy/POY4/docs/data/SampleData.zip`.

2 Loading and using Sankoff characters

In this tutorial we will load some Sankoff (matrix) type characters, and introduce the commands `build`, `cd`, `pwd`, `read`, `report`, `select`, `transform`, `exit`.

1. Before we read data, we will make sure that POY is working in the directory that containing the data files. The working directory tells the application where to look for the files. In this way, whenever we tell POY to read a file, we don't need to specify where is it located in the file system, we can simply use its name. To Change Directory we use the command `cd`, as in

```
cd ("/Users/andres/Desktop/SampleData")
```

   You will have to modify the path to match your particular computer organization.

2. Every command in POY is composed of a command name in lower case (in this case `cd`), followed by its arguments in parentheses. In this example, `cd` takes only one argument, which is a string enclosed in quotes: in POY, very string enclosed in quotes represents a file or a directory. In the last example, it is the directory SampleData.

3. To verify that POY is in the correct directory, we can Print the Working Directory with the command

```
pwd()
```

The program should now print (in the output frame) the correct path to read the SampleData directory. If not, use the `cd` command again and make sure that the program does not give you any error. Remember that you can always use the `<TAB>` to autocomplete file and directory names. This will help you to avoid many mistakes!.

4. Once we are working in the desired directory, we can read input data using the command `read`. POY supports ASN.1, Clustal, FASTA, GBSeq, Genbank, Hennig86, Newick, NewSeq, NEXUS, PHYLIP, POY3, TinySeq, and XML formats directly, and performs an automatic file format recognition. In this tutorial we need to read the sample files `35.san` and `1.fasta`. Type:

```
read ("35.san", "1.fasta")
```

5. `read` also accepts wild cards. For example, to read all the files with extension .fasta, it would be enough to use the command `read ("*.fasta")` (do not run it now!).

6. Another feature of this application is that input files *add* data to existing data. For example, we could have used two separate `read` commands, and we would have had the same effect:

```
read("35.san")
read("1.fasta")
```

7. After the data has been read, the output frame contains information displaying what type of files where loaded, and what their contents are. It is advisable to verify if those files where properly parsed by checking the characters and terminals POY has in memory. To do this, we use the command

```
report (data)
```

8. Using the arrows, PageUp, and PageDown keys, navigate the contents of the output frame. You will see that two kinds of characters are currently in memory: Sankoff and Molecular. Sankoff characters where loaded from the `35.san` file, and the one gene contained in the `1.fasta` file is the molecular character.

9. We will now run a small (and weak) analysis for just four minutes. So type the command:

```
search (max_time:0:0:4)
```

Now we have to wait four minutes for the program to run a search that includes building trees, swapping them with TBR, using a ratchet procedure to escape local optima, and tree fusing.

10. Once the search has finished, take a moment to see what the interactive console displays: the best cost found, how many times it was found, and how many trees are currently held in memory. We are only interested in the best (shortest) trees found, so we can get rid of duplicated and suboptimal trees with:

```
select ()
```

11. We now look at the trees using the command:

```
report (asciitrees)
```

Notice that POY colors the branches so that you can follow them easily when scrolling up and down in the output frame.

12. Seeing the trees on screen is somewhat useful, but it would be better if we could produce them in parenthetical notation to use in other programs like TreeView. We can do this using the command:

```
report (trees)
```

This will generate trees in newick format. To store them in a file, we simply write first the name of the output file that should contain them:

```
report ("trees.txt", trees)
```

13. How about publication quality trees directly from POY? The following command will produce a postscript file that can be read in Adobe Illustrator or any vectorial image edition program:

```
report ("graphic_tree.ps", graphtrees)
```

14. Excellent! we have finished now, time to close the application:

```
exit ()
```

3 Loading and using Non-additive (Unordered, Fitch) characters

1. We will follow a similar procedure to the one used with Sankoff characters. Follow steps 1, 2, and 3 of Section 2 to change to the exercises directory.

2. In this tutorial we will not build new trees, we just want to show that the same `read` command works to read both regular *data* and *trees*. To do this, we read from separate files, one containing the character matrix of non-additive characters, and the other containing the trees:

```
read ("1Fitch.ss", "1Fitch.tre")
```

3. After reading, this time we will report the trees in memory to one file, and the data that were analyzed together with the tree diagnosis in a second one:

```
report ("Fitch_tree.txt", trees:(total))
report ("Fitch_results.txt", data, diagnosis)
```

4. This is all we want for now, so close the application.

```
exit ()
```

Explore the contents of the two files that were reported: `Fitch_tree.txt` and `Fitch_results.txt`.

4 Loading and using Additive characters

1. Once more, we will repeat (more or less) what we did for Non-additive characters with Additive (Ordered, Wagner) characters. To do this, open the interactive console, and move to the exercises directory.

2. This time we will input the data contained in the '31' files:

```
read ("31.ss", "31.ss.tree")
```

3. Here, we will use report not only to create the contents as before, but also output the trees in human readable format to the screen. To do this, repeat these commands one by one, and observe the effect:

```
report ("Wagner_results.txt", data, diagnosis)
report (trees:(total))
report (asciitrees)
exit ()
```

Can you guess what the command would be to output the asciitrees in a file?

From now and on we will not mention the need to change to the exercises directory, but doing so will be required to perform the following exercises.

5 Loading a Direct Optimization Sequence

1. Again change to the exercises directory (Section 2), and read the files `3.fasta` and `5.fasta`. Both contain sequences in FASTA format:

```
read ("3.fasta", "5.fasta")
```

2. The `cross_references` argument of the `report` command is quite useful in checking the completeness of data files relative to one another. In addition to giving us an idea of our data completeness (to a level we may not want to know!), producing a presence/absence table of terminals versus files:

```
report (cross_references)
```

3. We will now construct 10 Wagner trees with the command `build` (the default), then select the best unique trees resulting from the Wagner builds and report the trees in parenthetical notation:

```
build ()
select ()
report (trees:(total))
```

4. Another useful way to view the data is to `report` the *implied alignment* of the molecular data currently loaded. Implied alignments can be used to discover problems in your data, and unexpected results before running the complete analysis:

```
report (ia)
```

5. Implied alignments also show us where we have issues with variability in sequence lengths as is the case with t16 (5.fasta). However, note that sequence length is not problematic for t18 (5.fasta). In POY the characters N and X are symbols used to represent any nucleotide base (as the IUPAC code specifies), while a question mark '?' represents any base or a gap. However, for missing sequences, the implied alignment always show them as gap-only sequences. This way your files will remain readable by other programs.

6. We are done for now with this tutorial. Close the interactive console:

```
exit ()
```

6 Searching the local neighborhood

1. In this section, we will use the command `swap`. This command performs a local search using either SPR or TBR as the tree neighborhood. However, as we learned in the previous sections, we must first read in data and build a set of initial trees:

```
read ("course.fasta")
build ()
```

2. The importance of understanding the commands used to create an initial set of trees for an analysis cannot be overemphasized. A search strategy usually proceeds by building an initial set of trees, each of which is then improved through a local heuristic search. Keeping this in mind it is easy to imagine how a search on one initial tree is drastically inferior to a search on 200 initial trees.

3. Making 10 trees seems like a rather limited number of replicates. We can request the program to build a larger number of trees by specifying an integer as the build argument. So let us now build 20 trees:

```
build (20)
```

4. Observe that a new build command eliminates the previous trees in memory. Every build will simply eliminate previous trees.

5. Now we can store those trees in parenthetical notation for later use, using the `report` command:

```
report ("built_trees.tree", trees)
```

6. To verify that we have successfully accomplished this task, we can read the tree file back in POY:

```
read ("built_trees.tree")
```

7. Notice that reading an input tree file does not delete previous trees in memory. This way it is possible to add new results to a set of current results. This is *always* the behavior of the `read` command: *adding* either data, or trees, to the current characters and trees available for analysis.

8. We should now verify that the input data were correctly loaded, as well as the trees:

```
report (data)
report (treestats)
```

9. In this exercise, we want to compare the effect of different swap options. In order to do this, we must always start from the very same trees and original data. So, at this point, we will introduce the command `wipe` that allow us to eliminate all the contents of the program's memory. Let us use that function, and verify that indeed there are neither data nor trees currently loaded:

```
wipe ()
report (data)
report (treestats)
```

   Observe that no data are now held in memory.

10. Now, we read the data again, the trees, and run the default `swap ()` command. Then we check the cost of the trees found:

```
read ("course.fasta", "built_trees.tree")
swap ()
report (treestats)
```

11. Write down the tree cost for later comparison. Question: what kind of neighborhood did POY use in this search? (TBR or SPR).

12. If we wanted to use a SPR or a TBR search, we can select it by passing the corresponding argument to `swap`. Let's give SPR a try now:

```
read ("course.fasta", "built_trees.tree")
swap (spr)
report (treestats)
```

13. Compare the results of the default neighborhood. What do you observe? Does this match your expectations? Which worked better? which took longer time?

7 Searching the local neighborhood (again)

1. Although `wipe ()` seems like a handy command, in interactive tests it would be desirable to have other means to store the state of the program and continue from there later on. The `store ()` and `use ()` commands serve this purpose.

2. `store ("name")` will store the current state of the program (that is all the data, trees, and support values) under the *name* label. Conversely, the command `use ("name")` sets the current state of the program to that which was stored under the label *name*. For example:

```
(* These are comments in a POY script *)
wipe ()
read ("course.fasta", "built_trees.tree")
report (data, treestats)
(* At this point you can see that we
have loaded some molecular data as well as
initial trees. We will store it under the
label initial_data. *)

store ("initial_data")
select ()

(* We will now modify the data, using a
command that will be introduced later. *)
transform (static_approximation)
report (data, treestats)
(* Notice that now we are only analyzing
static homology characters. *)

store ("static_data")
use ("initial_data")
report (data, treestats)
(* See the characters are the same as the
originals? *)
```

```
use ("static_data")
report (data, treestats)
(* Is it what you expected? *)
```

3. We can now do things more efficiently. Lets compare SPR and TBR again:

```
wipe ()
read ("course.fasta", "built_trees.tree")
store ("initial")
swap (tbr)
report (treestats)
use ("initial")
swap (spr)
report (treestats)
```

4. Finally make sure that the best trees that we found are saved for future use (we will need them later).

```
report ("swap_course.tree")
exit ()
```

8 Trees argument

1. By default, from every starting tree, POY will only store one resulting tree. That is, if you do:

```
build (1)
swap ()
```

You are guaranteed to end with one tree in memory. What about other trees of the same cost that could be found during the search? In order to increase the number of trees held in memory, after `swap`, you can use the `trees` argument:

```
use ("initial")
select (best:1)
report (treestats)
swap (trees:10)
```

Did you get more than one tree at the end?

2. We can also keep suboptimal trees in memory by using the `threshold` argument. This allow us to find trees that are at most some number of steps longer than the current optimum. For example, try the following:

```
use ("initial")
select (best:1)
swap (threshold:20, trees:10)
report (treestats)
```

   How many trees did you get? What are the minimum and maximum lengths?

3. Now we will use the argument `timeout`. We are preparing a talk that starts in 5 minutes, and only have 3 minutes to run the analysis, and then only 2 to prepare the slides! So we decide to build 3 trees, and then swap for 60 seconds each, here is how we would do it:

```
use ("initial")
select (best:3)
swap (timeout:60)
report (treestats)
select ()
report ("graphic_trees.ps", graphtrees)
```

   The argument timeout will stop the search on each tree after 60 seconds, and the result will be the best tree found in that short amount of search. Finally we have the best tree in a file to prepare the slide.

4. Let's continue with the same scenario. This time you decide to let it run for as long as it can, and stop it manually. For this case, the `recover` argument is useful:

```
use ("initial")
select (best:3)
report (treestats)
swap (recover)

(* Now let it swap for a little bit so
that POY finds some better trees. Then
interrupt it with <C-c>. *)

recover ()
report (treestats)
select ()
```

5. Same scenario again. This time, you find that your initial trees are not that bad, and you would like to swap only on those trees that resolve the strict consensus produced by the best three trees. To do this we use the `constraint` argument:

```
use ("initial")
select (best:3)
swap (constraint)
```

6. You can also provide a constraint file from an input file. Let's try that setup. First create the consensus file:

```
use ("initial")
select (best:3)
report ("constraint.txt", consensus)
```

   Now open the `constraint.txt` file and delete the title at the beginning. We are ready to proceed with the swap:

```
swap (constraint:("constraint.txt"), timeout:10)
```

   Notice that this time we have also added a timeout of 10 minutes. The main difference between using the constraint file in this way with the automated constraint in the previous item is that the reported consensus tree collapse zero length branches.

7. Same scenario, but this time ... ehm ... POY is crashing. You don't know what could be the problem, and Andrés has not been able to fix it soon enough. There is a nice argument called `trajectory` which allows `swap` to print every better tree found during the search. You can now give it a try:

```
use ("initial")
swap (trajectory)
```

8. OK, printing on the screen is useful, but the program is crashing! So I still loose the results! No problem, just output the trees in a file:

```
use ("initial")
swap (trajectory:"trees_of_swap.txt")

(* Let it run for few minutes and then
cancel with <C-c> *)
```

```
read ("trees_of_swap.txt")
report (treestats)
(* We have all the trees of the trajectory
plus the initial trees before the swap *)

select ()
```

9. Trajectory only reports a tree if its better than the current best. How about all the trees that have been evaluated? This is a useful command for Bremer support calculations (which we will learn later). Let's give that argument a try. We will ask POY to print the trees in the `visited_trees.txt` file:

```
use ("initial")
swap (visited)

(* Wow! many trees on screen!
Cancel it with <C-c> *)
swap (visited:"visited_trees.txt")

(* Let it run for few _seconds_ and then
cancel it with <C-c> *)
read ("visited_trees.txt")
report (treestats)
```

   Observe that we have lots of trees in memory now. `visited` collect *every* tested tree during the search. This is a lot! If you let it run long enough, `visited` will produce file of several Gigabytes. So beware, don't try to read one of those files directly into POY, either it will take a long time to finish, or it will just run out of memory and it is not really the program's fault.

10. To finalize this tutorial, complete the TBR swap on your initial 20 trees and store the resulting trees in the file `tbr_trees.tree`.

9 Modifying your characters with transform

1. Your input characters can be modified in many ways, for example to use a particular cost or weighting scheme, as well as to modify the type of character being analyzed. To begin this series of exercises, let's start with our typical data set:

```
read ("course.fasta")
```

2. Now we will report the cost matrix being used in the loaded characters:

```
report (data)
```

3. You can see that by default POY will give cost 2 to each indel, and 1 to every substitution (that's what the `tcm:(1,2)` means). We can modify all the characters with the command `transform`, as follows:

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

   This will change will the characters for which a transformation cost matrix for an alignment is applicable. `tcm:(1,1)` will assign cost 1 to every substitution and 1 to every indel. Let's verify its effect:

```
report (data)
```

4. We can also assign a particular cost to opening a gap block. Not surprisingly the argument is `gap_opening`:

```
transform (tcm:(3,1), gap_opening:3)
report (data)
```

   Can you see the effect in the `data` report? It is time now to see the effect of the different parameters in the implied alignment.

5. First read again your input data and build a tree:

```
wipe ()
read ("course.fasta")
build (1)
```

6. Now we write down the cost of the tree, and output the implied alignment in a file:

```
report (treestats, "1_2_ia.txt", implied_alignments)
```

7. Next we modify the cost regime to substitutions 1, indels 1, and report the new cost as well as the implied alignment:

```
transform (tcm:(1,1))
report (treestats, "1_1_ia.txt", implied_alignments)
```

8. Finally we will do the same operations using a cost of 3 for substitutions, 1 for an individual gap, and 3 for gap opening:

```
transform (tcm:(3,1), gap_opening:3)
report (treestats, "3_1_3_ia.txt", implied_alignments)
wipe ()
```

9. Compare the costs and the implied alignments. What do you expect? what do you observe? Are the transformation cost matrices metric? Are your characters metric?

10. You can fix a particular scheme of indels using the command `transform (static_approx)`, which stands for "static approximation". A static approximation fixes a particular implied alignment for the best tree in memory, and creates a set of characters that match that particular alignment and resembles as much as possible the cost regime of choice. Here is example of this:

```
read ("course.fasta")
build (1)
transform (tcm:(1,1))
report (data)
```

11. We see that there are 8 molecular characters currently in memory. Before we continue, as we will play around with this initial set of characters and tree, we should store this initial state of the program:

```
store ("initial")
```

12. We can now check the implied alignment:

```
report (ia)
```

    Yes, `ia` and `implied_alignment` are equivalent.

13. This alignment can now be fixed to use the resulting matrix as the characters:

```
transform (static_approx)
report (data)
```

14. Observe that after the transform there are no molecular characters left. Instead, there are a number of non-additive characters.

15. What happens if we have the default cost regime? Let's roll back to the characters stored in "initial" and give this a try:

```
use ("initial")
transform (tcm:(1,2))
transform (static_approx)
report (data)
```

What can you observe?

16. Finally, let's check how the static approximation behaves if you have a gap opening parameter:

```
use ("initial")
transform (tcm:(3,1), gap_opening:3)
transform (static_approx)
report (data)
```

What is the main difference the you observe? How are indel blocks being treated?

17. Now we will learn how to `transform` specific characters. Suppose that we would like to assign `tcm:(2,1)` to the first fragment in course.fasta. We first check the name of the fragment:

```
use ("initial")
report (data)
```

You can see that the name of the first fragment is `course.fasta:0` (the precise name may vary slightly in your computer). We can specify in the transform command which characters should be transformed in which way:

```
transform ((names:("couse.fasta:0"), tcm:(2,1)))
```

18. try to visually match the parenthesis and understand their effect. Here is another example, aimed at up-weighting static homology characters only:

```
transform ((static, weight:2))
```

In this case instead of specifying characters by name, we do it by type. This command probably makes the syntax easier to understand. If you had troubles with the first one, try to understand the `weight` example and go back to the `tcm:(2,1)` case again.

19. To finish this section, we leave you a task: fix the alignment of the third and fourth fragments of the file course.fasta using cost 1 for substitutions and cost 1 for indels. Every other character should have the default cost regime of substitutions 1 and indels 2.

10 Executing scripts

1. POY optimizes the execution of a script to reduce memory consumption. To illustrate this, let's look at the following very simple search:

```
read ("course.fasta")
build (3)
swap ()
select ()
```

   If we run it interactively, line by line, then POY would:

   - build 3 trees using a random addition sequence.
   - swap those three trees using TBR.
   - select the best, unique trees from those stored in memory at the end of the swap.

   At the peak of memory consumption, the program should have 3 trees in memory. Imagine that the script now looks as follows:

```
read ("course.fasta")
build (500)
swap ()
select ()
```

   Then we would need to have enough memory capacity to hold 500 trees! This might not be possible.

2. However, this *script* can be executed differently:

   - 500 times:
     - build 1 tree
     - swap it
     - compare with previous trees and select the best unique trees.

   This would require, with high probability, only two trees in memory at the peak. This is exactly how POY will execute it. To test it, run the first script of item 1 in a single line. See how the program behaves slightly differently.

3. Notice that POY reports pipelines instead of simple builds, swaps, and search statements, and the estimated time does not correspond to each of these steps, but complete pipelines. A pipeline here includes both the `build` and `swap` steps.

4. Now test the following two scripts and see what effect they have:

   - ```
     wipe ()
     read ("course.fasta")
     build (3)
     swap ()
     report (treestats)
     select ()
     ```

   - ```
     wipe ()
     read ("course.fasta")
     build (3)
     swap (constraint)
     select ()
     ```

   - ```
     wipe ()
     read ("course.fasta")
     build (3)
     swap ()
     perturb (transform (static_approx))
     select ()
     ```

   - ```
     wipe ()
     read ("course.fasta")
     build (3)
     swap ()
     transform (static_approx)
     perturb ()
     select ()
     ```

5. In order to execute a script in the interactive console, write the commands in a file using notepad (or text edit), save it as a plain text file, and then use the command:

   ```
   run ("script.txt")
   ```

   You can include the run command in other scripts to call, for example, routines for searches or for support value calculations. Write a simple script and try to execute it in this way.

11 Tree Fusing

1. The `fuse` command implements tree fusing, in which pairs of trees and compared, and all pairs of subtrees that have different resolution but the same terminals are exchanged, in an attempt to find better combinations. We will first run a default `fuse ()`:

```
wipe ()
read ("course.fasta", "swap_course.tree")
report (treestats)
fuse ()
report (treestats)
```

   How many iterations happened?

2. During an iteration, the program picks two trees at random, a source, and a destination, to compare. When a better tree than the destination is found, fuse replaces it with the better tree.

3. Note that by default the `fuse` command does not `swap`. However, like many other commands in POY, `fuse` accepts a `swap` argument. Let's now run a `fuse` with a `swap`:

```
wipe ()
read ("course.fasta", "swap_course.tree")
report (treestats)
fuse (swap (tbr))
report (treestats)
```

4. This time save the trees found for later use:

```
report ("fuse_course.tree")
```

5. Now try to construct a fuse step which lasts for at most 20 seconds. (Use the command documentation section about `swap` for help.)

12 Perturbing trees for improved searches

1. Perturbing the search space is a powerful search strategy. The fundamental concept is to modify the cost of the trees in such a way that we can find even better trees from already good ones, that is, from trees that are local optima. The `perturb` command in POY implements this kind of search strategy.

2. `perturb` works in the following way: for n iterations, POY perturbs the characters using the arguments and options that we specify, searches for optimal trees in the altered character landscape, and finally searches the current best tree using the original characters.

3. The parsimony Ratchet is a classic powerful perturbation strategy in phylogenetic research. In POY, the `ratchet` argument within the `perturb` command works by up-weighting a percentage of characters. The default settings of `perturb ()` performs a ratchet in which 25% of the characters are up-weighted by a factor of 2.

4. Let us now perturb our data with the ratchet. First, we will read the data, and the trees stored in the fuse tutorial, to perturb them:

```
read ("course.fasta", "fuse_course.tree")
perturb ()
```

5. How many iterations are performed by default? How many characters do you have in your analysis?

6. As you saw in the previous command, one problem we have is that the ratchet works on characters, and this data set has few of them: only 8. Our experience is that an excellent strategy is to apply the ratchet on the characters produced by the implied alignment, that is, on the static approximation.

7. To do this, we use the `transform` command as an argument of the `perturb` command:

```
perturb (transform (static_approx))
```

   which executes the following algorithm:

   - For 5 iterations
     - Run the parsimony ratchet
     - Transform back to the original dynamic homology characters
     - Run a new search in the resulting tree
     - If the new tree is better, replace the original.

8. Now lets perform a ratchet with SPR and static approximation:

```
perturb (transform (static_approx), swap (spr))
```

9. Alternatively, we can try to escape the local optima by perturbing the cost of the matrix employed by the dynamic homology characters:

```
perturb (transform (tcm:(1,1)))
```

Can you describe what this command does? An important observation is that running five iterations of this command does not help at all. Can you see why?

13 Using Search

1. The `search` command runs a mixture of:
   - Randomized wagner builds
   - TBR swapping
   - Nixon's ratchet
   - Exhaustive DO
   - Tree fusing
2. `search` should be the first analysis option for both new and expert users. Let's start with a very small data set, so that we can get some meaningful results with a very small amount of time:

   ```
   wipe ()
   read ("18s.fasta")
   search (max_time:0:0:1)
   (* This will run a search for exactly 1
   minute *)
   ```

3. After this command finishes, you should see a message on screen telling you how many trees where built, how many fusing generations where performed, how many times the best tree was found, and what its score is.
4. We can constraint the search some more. Suppose that we have, from previous searches, the impression that the best tree that we could find has cost 385, hen we can tell POY that this should be the target cost for an expected number of hits:

   ```
   wipe ()
   read ("18s.fasta")
   search (max_time:0:1:0, hits:5, target_cost:385)
   ```

   This command will now run for *one hour* or until it has found 5 trees with cost 385 or less, whichever happens first.

5. If we have limited memory resources, we can now execute this search with a memory constraint, so that POY will only store as many trees as it can fit in 256 MB or RAM, not more.

```
read ("18s.fasta")
search (max_time:0:1:0, hits:5, target_cost:385,
    memory:mb:256)
```

6. So lets run a regular search for half an hour on the 18s data set, and see what results you get. We will store the resulting trees in the file 18s.tree:

```
read ("18s.fasta")
search (max_time:0:0:30, memory:mb:256)
select ()
report ("search_18s.tree", trees)
```

14 Calculating Supports: Jackknife

1. The command `calculate_support (jackknife)` performs n pseudoreplicates independently, in each once a percentage of characters in selected at random, without replacement. The frequency of occurrence of a clade is its jackknife support value.

2. Open the program documentation (in the help menu of the Graphic User Interface Window). Go to the command reference chapter and find the `calculate_support` command. Find the jackknife argument, and read its description and default values, and how to specify the number of pseudoreplicates inside calculate_support. Then write a command to do just 50 pseudorplicates, using the default values of jackknife.

3. An important detail is that jackknife resamples the characters. What you define as a characters is a biological question that you must resolve before running your analyses. POY does not calculate jackknife supports in any special way, it is just the case that the characters that you are using are not necessarily just the bases of your sequences, but the sequences themselves!

4. A common procedure used by many biologists is to fix an alignment and compute support values resampling the bases of that alignment. Let us use this strategy and compare the support values for a pair of different alignments on the same tree. To do this we will use a new report argument:

```
read ("course.fasta")
transform (tcm:(1,1))
store ("initial")
```

```
search (max_time:0:0:1)
select ()
```

5. We store the tree in a file so that then we can report support values for it:

```
report ("tree_for_support_values.tre", trees)
transform (static_approx)
report ("good_alignment.ss", phastwinclad)
(* This commands output a NONA file that can
be read in many programs, most notably POY *)
```

6. Now we will produce a different alignment and store it

```
use ("initial")
build (1)
transform (static_approx)
report ("regular_alignment.ss", phastwinclad)
```

7. We have just produced our two data sets that we will use to calculate supports. The second data set is currently held in memory.

```
calculate_support (jackknife: (resample:1000))
read ("tree_for_support_values.tre")
calculate_support (jackknife: (resample:1000))
read ("tree_for_support_values.tre")
(*We just stored the tree in a file to
compare the results later *)
```

8. Now we will compute the support values using the first alignment

```
wipe ()
read ("good_alignment.ss")
calculate_support (jackknife: (resample:1000)
read ("tree_for_support_values.tre")
report ("good_alignment_supports.ps",
    graphsupports:jackknife)
```

Compare the two trees (using an image edition program, nothing special is needed for Mac OS X, Linux, or Unix, but Windows computers may need Ghostview or Adobe Illustrator):

- good_alignment_supports.ps.
- regular_alignment_supports.ps.

What did you notice? Where do you get better supports? Which would be a reasonable alignment from which to compute your support values?

9. To calculate bootstrap support values, find the `bootstrap` argument in the `calculate_support` command documentation, and repeat the previous (Calculating Supports: Jackknife) excercise using good_alignment.ss and regular_alignment.ss files to generate bootstrap supports for the tree in tree_for_support_values.tre, in eactly the same way.

15 Calculating Supports: Bremer

1. One way to do a search for Bremer supports is to constrain the search so that a particular clade is never allowed in the tree that is being visited. This is exactly what `calculate_supports (bremer)` does.

   To test this command, first read the best tree that you could find from all the searches (e.g. after fusing and ratcheting). We will first generate a good tree for the bremer supports:

```
read ("course.fasta")
search (max_time:0:0:3)
select ()
(*Store the tree in a file for the
next tutorial *)
report ("tree_for_bremer.tre", trees)
(*We have the tree from which bremer
is to be calculated*)
select (best:1)
calculate_support (bremer)
```

   Notice that this command is slow, as it performs a very intense search. Ten trees are built and swapped on each branch. If your original search was not powerful enough, this command may find shorter trees than your initial search.

   When this command is finished, we are ready to report the support values for this tree.

```
report (supports:bremer)
```

   Another way to calculate Bremer supports is to not constrain the search at all, and from the trees visited, collect the information for the clades not present. To do this we can use the `visited` argument within swap.

```
build (100)
swap (cisited:"for_bremer.txt")
select (best:1)
read ("tree_for_bremer.tre")
report (supports:bremer:"for_bremer.txt")
```

This strategy has several advantages. First and foremost, it tends to yield tighter support values (lower). Secondly, it is more efficient in the sense that the default technique has to repeat the search for every clade to collect the necessary information, while this strategy collects all the clades from the same strategy. Finally, you can calculate Bremer support with your search strategy of choice. You may cancel this Bremer search at any time and the necessary information will remain stored in the file and the best tree found in your overall search, which means that you will never recover negative Bremer support values.

Index