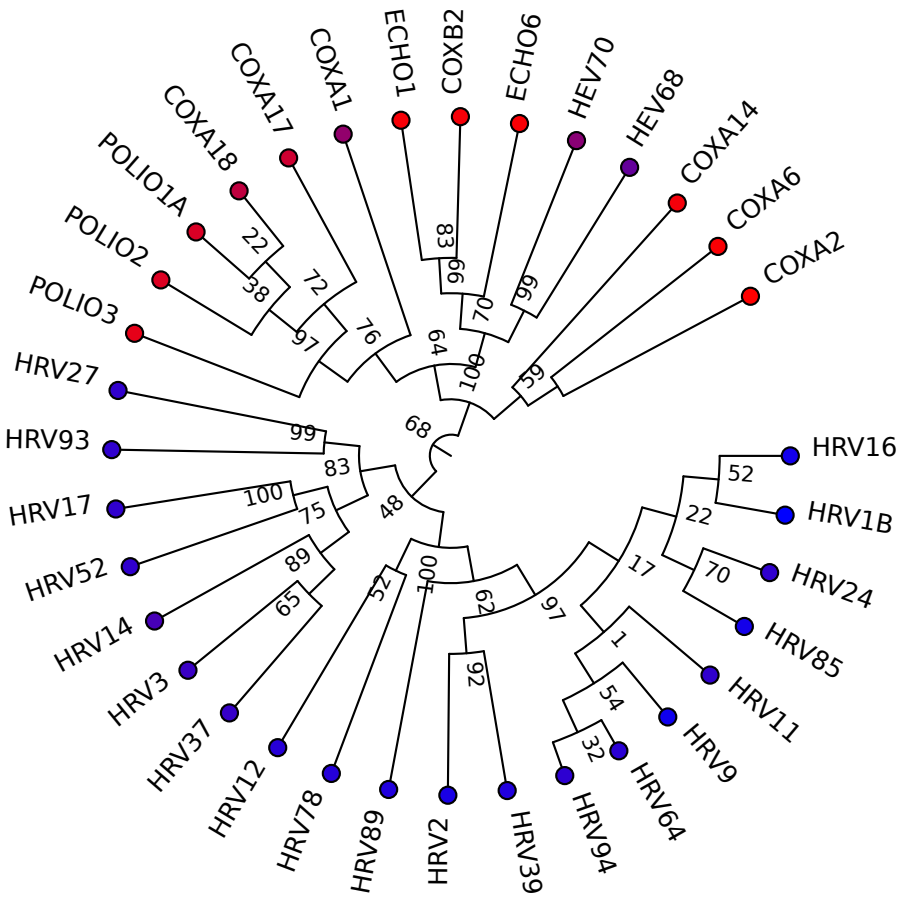
Newick Utilities Tutorial

Version 1.3.5 – February 20, 2011

Thomas Junier `thomas.junier@unige.ch`
Computational Evolutionary Genomics Group
Department of Genetic Medicine and Development
University of Geneva, Switzerland
***
Swiss Institute of Bioinformatics

`http://cegg.unige.ch/newick_utils`



Contents

Introduction

The Newick Utilities are a set of UNIX (including Mac OS X) and UNIX-like (Cygwin) shell programs for working with phylogenetic trees. Their main features are:

- they require no user interaction[1]
- they can work on any number of trees at a time
- they perform well with large trees
- they are implemented as filters[2]
- they read and write text

They are not tools for *making* phylogenies. Rather, they are for processing existing ones, by which I mean manipulating the tree or extracting information from it: rerooting, simplifying, extracting subtrees, printing branch lengths and distances, etc - see table 1; a glance at the table of contents should also give you an idea.

Each of the programs performs one task (with some variants). For example, here is how you would reroot a series of phylograms contained in file `mytrees.nw` using node `Dmelano` as the outgroup:

```
$ nw_reroot mytrees.nw Dmelano
```

Now, you might want to make cladograms from the rerooted trees. Program `nw_topology` does the job, and since the utilities are filters, you can do it in a single command:

```
$ nw_reroot mytrees.nw Dmelano | nw_topology -
```

As you can see, it is straightforward to pipe Newick Utilities together, and of course they can be mixed freely with any other shell program (see e.g. 2.14.1).

About This Document

This tutorial is organized as follows: chapter 1 discusses common features of the Newick Utilities, chapter 2 shows examples of simple tasks, and chapter 3 has examples of more advanced tasks.

It is not necessary to read this material in order: you can pretty much jump to any section in chapters 2 and 3, they do not require reading previous sections. I would suggest reading chapter 1, then section 2.1 because it explains how all the tree graphics were produced.

The files for all the examples in this tutorial can be found in subdirectory `data`.

All the program outputs are generated on-the-fly just before the document is run through LaTeX, so they represent the actual output of the latest version of the utilities.

[1] Why this is a good thing is not the focus of this document: I shall assume that if you are reading this, you already know when a command-line interface is better than an interactive interface.

[2] In UNIX jargon, a *filter* is a program that reads input from standard input and writes output to standard output.

Program	Function
nw_clade	Extracts subtrees specified by node labels
nw_condense	Condenses (simplifies) trees
nw_display	Shows trees as graphs (ASCII graphics or SVG)
nw_duration	Convert node ages into duration
nw_distance	Prints distances between nodes, in various ways
nw_ed	Stream editor (*à la* sed or awk)
nw_gen	Random tree generator
nw_indent	Shows Newick in indented form
nw_labels	Prints node labels
nw_match	Finds matches of a tree in another one
nw_order	Orders tree (preserving topology)
nw_prune	Removes branches based on labels
nw_rename	Changes node labels according to a mapping
nw_reroot	(Re)roots the tree
nw_stats	Prints tree statistics and properties
nw_support	Computes bootstrap support of a tree given replicate trees
nw_topology	Alters branch properties, preserving topology
nw_trim	Trims a tree at a specified depth

Table 1: The Newick Utilities and their functions

Citing the Newick Utilities

If you use the Newick Utilities for published work, please cite: T. Junier and E. M. Zdobnov. The Newick Utilities: High-throughput Phylogenetic tree Processing in the UNIX Shell. *Bioinformatics*, May 2010

Chapter 1

General Remarks

The following applies to all programs in the Newick Utilities package.

1.1 Help

All programs print a help message if passed option `-h`. Here are the first 20 lines of `nw_indent`'s help:

```
$ nw_indent -h | head -20

Indents the Newick, making structure more clear.

Synopsis
--------

/home/tjunier/projects/newick_utils/src/.libs/lt-nw_indent [-cht:] <newick trees
 filename|->

Input
-----

Argument is the name of a file that contains Newick trees, or '-' (in
which case trees are read from standard input).

Output
------

By default, prints the input tree, with each parenthesis and each leaf on a
line of its own, and indented a multiple of '  ' (two spaces) to reflect
structure. The default output is valid Newick.
```

The help page describes the program's purpose, its input and output, and its options, in a format reminiscent of UNIX manpages. It also shows a few examples. All examples can be tried out using files in the `data` directory.

1.2 Input

Since the Newick Utilities are for working with trees, it should be no surprise that the main input is a file containing trees. The trees must be in Newick format, which is one of the most widely used tree formats. Its complete description can be found at `http://evolution.genetics.washington.edu/phylip/newicktree.html`.

The input file is always the first argument to the program (after any options). It may be a file stored in a filesystem, or standard input. In the latter case, the filename is replaced by a '−' (dash):

```
$ nw_display mytrees.nw
```

is the same as

```
$ cat mytrees.nw | nw_display -
```

or

```
$ nw_display - < mytrees.nw
```

Of course the second ("dashed") form is only really useful when chaining several programs into pipelines.

1.2.1 Multiple Input Trees

The input file can contain one or more trees. When there is more than one, I prefer to have one tree per line, but this is not a requirement: they can be separated by any amount of whitespace, including none at all. The task will be performed[1] on each tree in the input. So if you need to reroot 1,000 trees on the same outgroup, you can do it all in a single step (see 2.3).

1.3 Output

All output is printed on standard output (warnings and error messages go to standard error). The output is either trees or information about trees. In the first case, the trees are in Newick format, one per line. In the second case, the format depends on the program, but it is always text (ASCII graphics, SVG, numeric data, textual data, etc.).

1.4 Options

Options change program behaviour and/or allow extra arguments to be passed. They are all passed on the command line, before the mandatory argument(s), using a single letter preceded by a dash, in the usual UNIX way. There are no mandatory control files, although some tasks require additional files (*e.g.* 2.1.2). For example, we saw above that `nw_display` produces graphs. By default the graph is ASCII graphics, but with option `-s`, the program produces SVG:

```
$ nw_display -s sometree.nw
```

All options are described in the program's help page (see 1.1).

[1] well, attempted...

Chapter 2

Simple Tasks

The tasks shown in this chapter all involve a single Newick Utilities program (plus possibly `nw_display`), so they can serve as introduction to each individual program.

2.1 Displaying Trees

Perhaps the simplest and most common operation on a Newick tree is just to look at it. But a Newick tree is not very intuitive for us humans, as we can quickly see by looking *e.g.* at a tree of Old World primates:

```
$ cat catarrhini
```

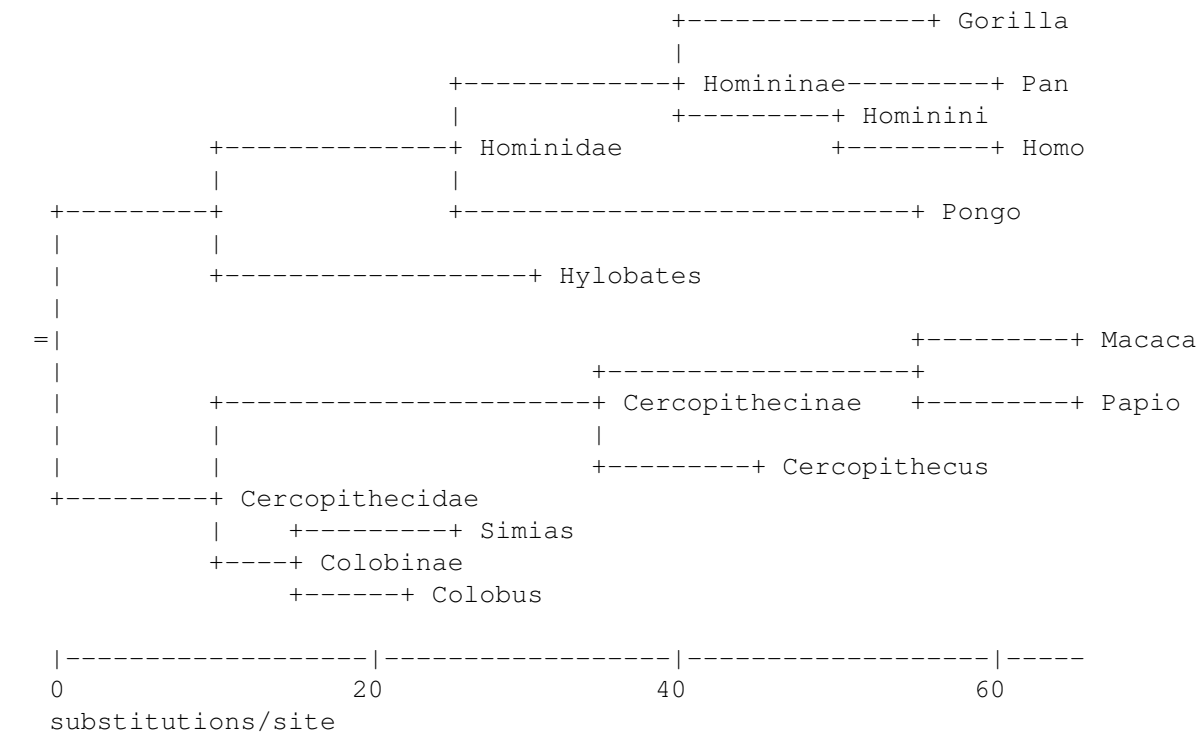
```
((((Gorilla:16,(Pan:10,Homo:10)Hominini:10)Homininae:15,Pongo:30)
Hominidae:15, Hylobates:20):10,(((Macaca:10,Papio:10):20,
Cercopithecus:10) Cercopithecinae:25,(Simias:10,Colobus:7)
Colobinae:5)Cercopithecidae:10);
```

So we want to make a graphical representation from it. This is the purpose of the `nw_display` program.

2.1.1 As Text

At its simplest, `nw_display` just outputs a text graph. Here is the primates tree, shown with `nw_display`:

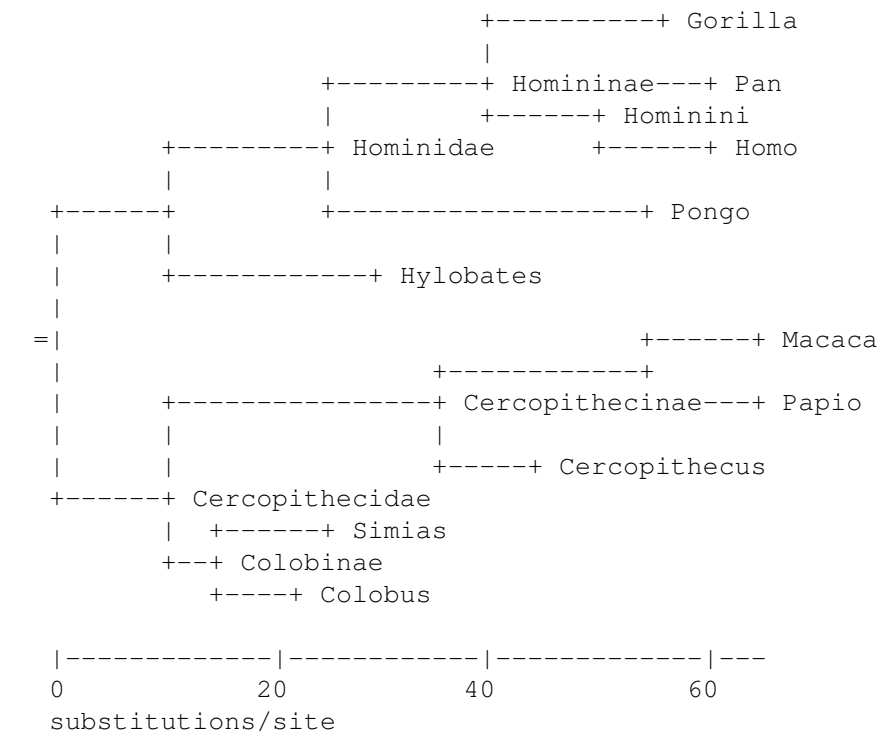
```
$ nw_display catarrhini
```



That's pretty low-tech compared to interactive, colorful graphical displays, but if you use the shell a lot (like I do), you may find it useful.

You can use option `-w` to set the number of columns available for display (the default is 80):

```
$ nw_display -w 60 catarrhini
```



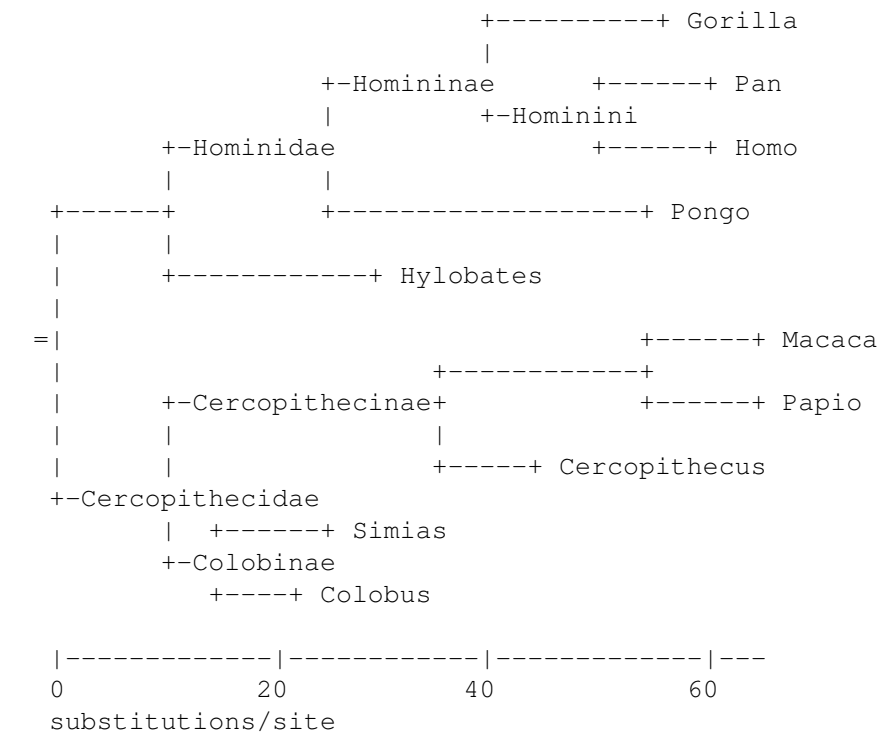
Scale Bar

If the tree is a phylogram, `nw_display` prints a scale bar. Its units can be specified with option `-u`, the default is substitutions per site. To suppress the scale bar, pass the `-S` switch. The scale bar can also "go backwards" (option `-t`), *i.e.* the scale bar's zero is aligned with the leaves and units increase towards the root. This is handy when the units are ages, *e.g.* in millions of years ago, but it only makes much sense if the leaves themselves are aligned. See 2.16 for an example.

Placement of Inner Node Labels

Option `-I` controls the placement of inner node labels. It takes an argument, which can be `l` (lowercase l – towards the leaves), `m` (in the middle), or `r` (towards the root). The default behaviour is `l`. Here is the above tree, with inner labels near the root:

```
$ nw_display -w 60 -Ir catarrhini
```



2.1.2 As Scalable Vector Graphics

First, a disclaimer: there are dozens of tools for viewing trees out there, and I'm not interested in competing with them. The reasons why I included SVG capabilities (besides automation, etc.) were:

- I wanted to be able to produce reasonably good graphics even if no other tool was at hand
- I wanted to be sure that large trees could be rendered[1]

To produce SVG, pass option `-s`:

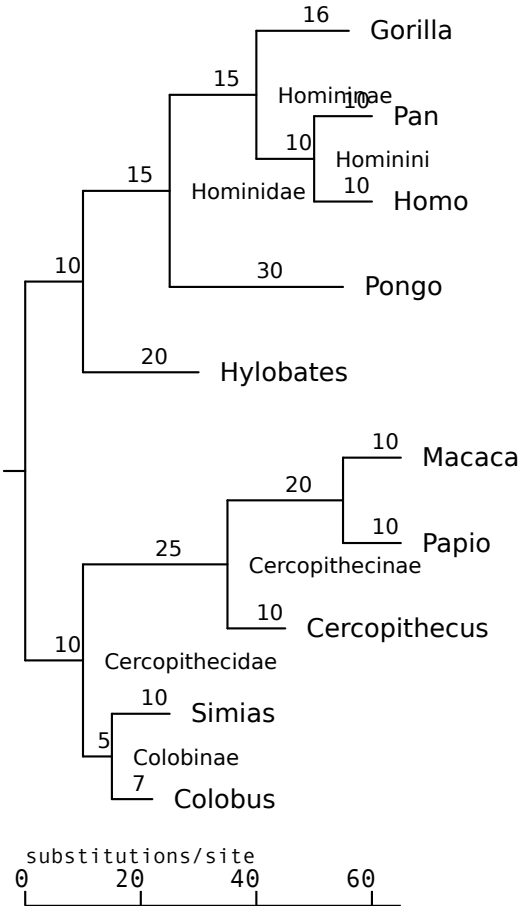
```
$ nw_display -s catarrhini > catarrhini.svg
```

Now you can visualize the result using any SVG-enabled tool (all good Web browsers can do it), or convert it to another format with, say `rsvg` or Inkscape (`http://www.inkscape.org`). The SVG produced by `nw_display` is designed to be easy to edit in an interactive editor (Inkscape, Adobe Illustrator, etc.): for example, the tree edges are in one group, and the text in another, so it is easy to change the line width of the edges, or the font family of the text (you can also do this from `nw_display` using a CSS map, see 2.1.2).

The following PDF image was produced like this:

```
$ inkscape -f catarrhini.svg -A catarrhini.pdf
```

[1] I have had serious problems visualising trees of more than 1,000 leaves using some popular software I will not name here - either it was painfully slow, or it simply crashed, or else the output was unreadable, incomplete, or otherwise unsuitable.



All SVG images shown in this tutorial were processed in the same way. In the rest of the document we will usually skip the redirection into an SVG file and omit the SVG-to-PDF conversion step.

Text-mode options

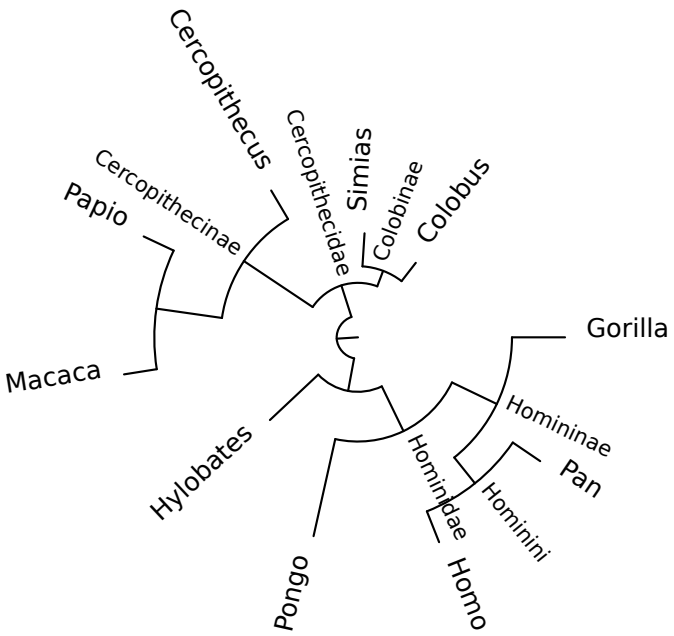
Options for ASCII trees also work for SVG: `-S` suppresses the scale bar[2], and `-u` specifies its units; `-w` governs the tree's width, except that for SVG the value is in pixels instead of columns; `-I` controls the placement of inner node labels.

Radial trees

You can make radial trees by passing the `-r` switch:

```
$ nw_display -sr -S -w 450 catarrhini
```

[2]The positioning of the scale bar is a bit crude in SVG mode, especially for radial trees. This is mainly because of the "SVG string length curse", that is, the impossibility of finding out the length of a text string in SVG. This means it is hard to ensure the scale bar will not overlap with a node label, unless one places it far away in a corner, which is what I do for now. An improvement to this is on my TODO list.



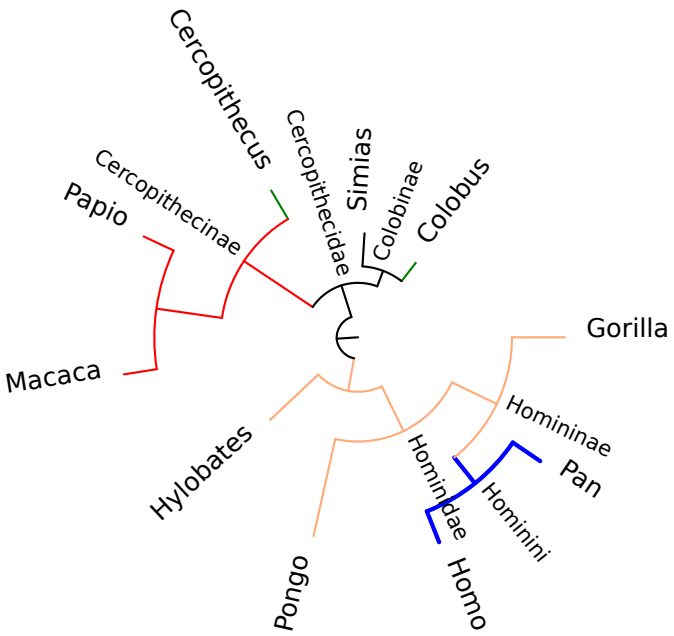
Using CSS

You can modify node style using CSS. This is done by specifying a CSS *map*, which is just a text file that says which style should be applied to which node. If file `css.map` contains the following

```
# Cercopithecidae in red
stroke:red  Clade Macaca Cercopithecus
# Apes (Hominoidea) in pinkish
stroke:#fa7  C  Homo Hylobates
# Colobus and Cercopithecus (individually) in green
stroke:green Individual Colobus Cercopithecus
# Hominines in thick blue
"stroke-width:2; stroke:blue"  Clade Homo Pan
```

we can apply the style map to the tree above by passing `-c`, which takes the name of the CSS file as argument:

```
$ nw_display -sr -S -w 450 -c css.map catarrhini
```



The syntax of the CSS map file is as follows:

- A line that starts with a # (hash) is a comment, and will be ignored, as will be any line that contains only whitespace (space and TAB), as well as empty lines.
- Each line describes one style and the set of nodes to which it applies. A line contains elements separated by whitespace (whitespace between quotes does not count).
- The first element of the line is the style, and it is a snippet of CSS code.
- The second element states whether the following nodes are to be treated individually or as a clade. It is either `Clade` or `Individual` (which can be abbreviated to `C` or `I`, respectively).
- The remaining element(s) are node labels and specify the nodes to which the style must be applied: if the second element was `Clade`, the program finds the last common ancestor of the nodes and applies the style to that node and all its descendants. If the second element was `Individual`, then the style is only applied to the nodes themselves.

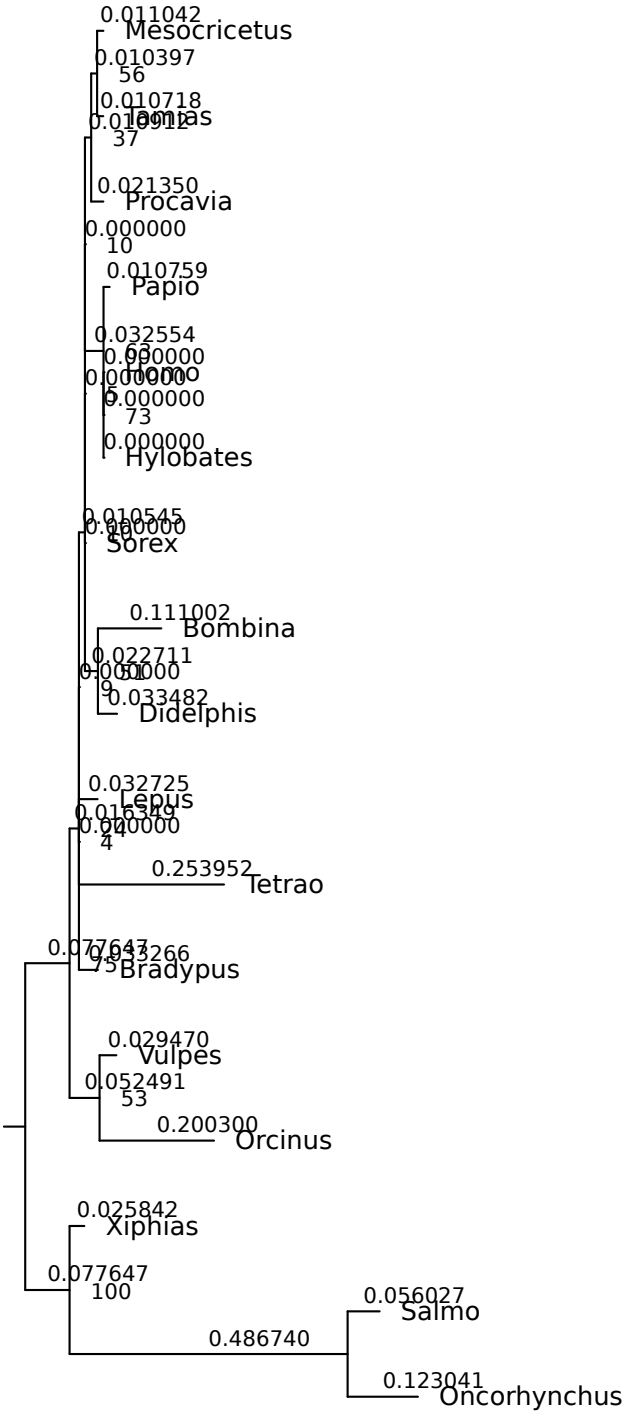
In our example, `css.map`:

- the first line states that the style `stroke:red` must be applied to the `Clade` defined by `Macaca` and `Cercopithecus`, which consists of these two nodes, their ancestor `Cercopithecinae`, and `Papio`.
- Line 2 prescribes that style `stroke:#fa7` (an SVG hexadecimal color specification) must be applied to the clade defined by `Homo` and `Hylobates`, which consists of these two nodes, their last common ancestor (unlabeled), and all its descendants (that is, `Homo`, `Pan`, `Gorilla`, `Pongo`, and `Hylobates`, as well as the inner nodes `Hominini`, `Homininae` and `Hominidae`).
- Line 3 instructs that the style `stroke:green` be applied individually to nodes `Colobus` and `Cercopithecus`, and only these nodes - not to the clade that they define.
- Line 4 says that style `stroke-width:2; stroke:blue` should be applied to the clade defined by `Homo` and `Pan` - note that the quotes have been removed: they are not part of the style, rather they allow us to improve readability by adding some whitespace.

The style of an inner clade overrides that of an outer clade, *e.g.*, although the `Homo - Pan` clade is nested inside the `Homo - Hylobates` clade, it has its own style (blue, wide lines) which overrides the containing clade's style (pinkish with normal width). Likewise, `Individual` overrides `Clade`, which is why `Cercopithecus` is green even though it belongs to a "red" clade.

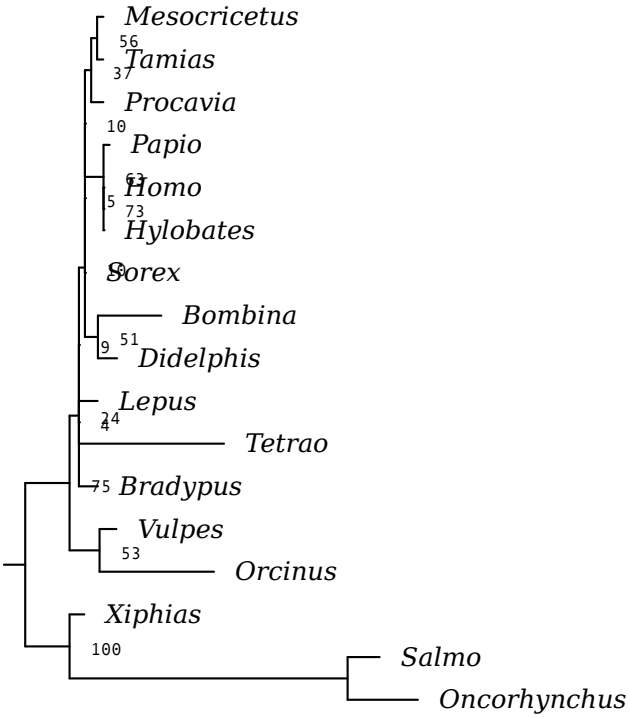
Styles can also be applied to labels. Option `-l` (lowercase l) specifies the leaf label style, option `-i` the inner node label style, and option `-b` the branch length style. For example, the following tree, which was produced using defaults, could be improved a bit:

```
$ nw_display -sS vertebrates.nw
```



Let's remove the branch length labels, reduce the vertical spacing, reduce the size of inner node labels (bootstrap values), and write the leaf labels in italics, using a font with serifs:

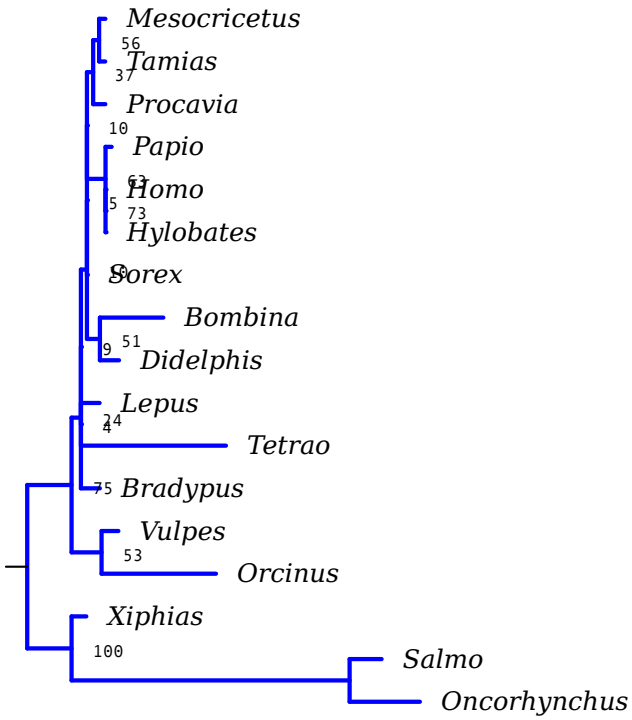
```
$ nw_display -s -S -v 20 -b 'opacity:0' -i 'font-size:8' \
-l 'font-family:serif;font-style:italic' vertebrates.nw
```



Still not perfect, but much better. Option `-v` specifies the vertical spacing, in pixels, between two successive leaves (the default is 40). Option `-b` sets the style of branch labels, option `-l` sets the style of leaf labels, and option `-i` sets the style of inner node labels. Note that we did not *discard* the branch lengths (we could do this with `nw_topology`), because doing so would reduce the tree to a cladogram. Instead, we set their CSS style to `opacity:0` (`visibility:hidden` also works).

What if we want to change the default style? Say we want the branches in blue, and two pixels wide? That's option `-d`:

```
$ nw_display -s -S -v 20 -b 'opacity:0' -i 'font-size:8' \
-l 'font-family:serif;font-style:italic' \
-d 'stroke-width:2;stroke:blue' vertebrates.nw
```



2.1.3 Ornaments

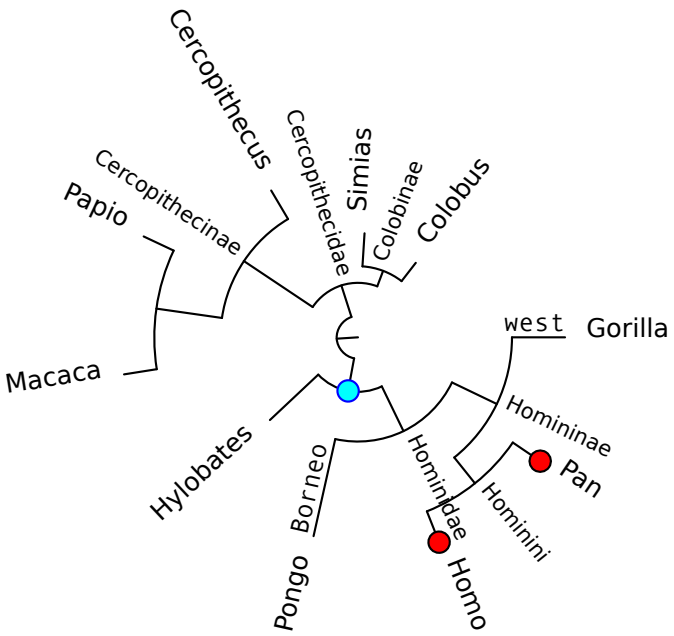
Ornaments are arbitrary snippets of SVG code that are displayed at specified node positions. Like CSS, this is done with a map. The ornament map has the same syntax as the CSS map, except that you specify SVG elements rather than CSS styles. The `Individual` keyword means that all nodes named on a given line sport the corresponding ornament, while `Clade` means that only the clade's LCA must be adorned. The ornament is translated in such a way that its (0,0) coordinate corresponds to the position of the node. In radial graphs, text ornaments are rotated like node labels.

The following file, `ornament.map`, instructs to draw a red circle with a black border on `Homo` and `Pan`, and a cyan circle with a blue border on the root of the `Homo - Hylobates` clade. `Gorilla` node will be annotated with the word "plains", and `Pongo` with "Borneo" in italics[3]. The SVG is enclosed in double quotes because it contains spaces - note that single quotes are used for the values of XML attributes. The ornament map is specified with option `-o`:

```
"<circle style='fill:red;stroke:black' r='5'/>" I Homo Pan
"<circle style='fill:cyan;stroke:blue' r='5'/>" C Homo Hylobates
"<text style='font-style:italic'>Borneo</text>" I Pongo
"<text>west</text>" I Gorilla
```

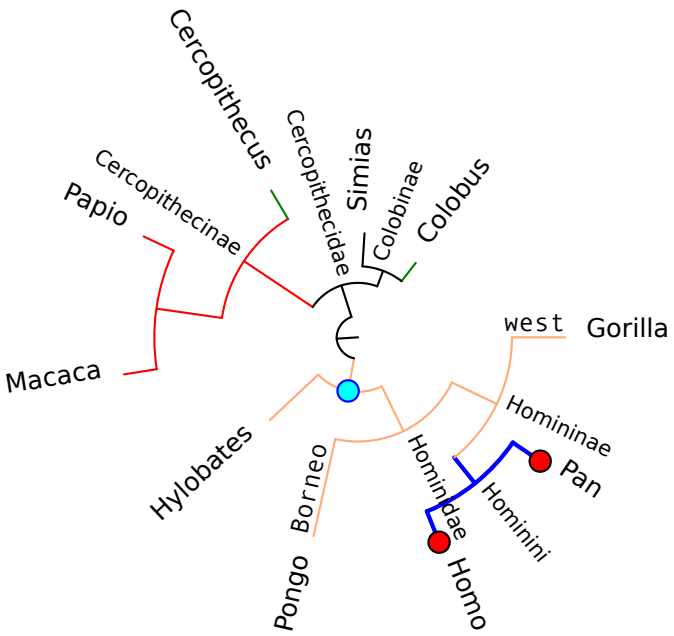
```
$ nw_display -sr -S -w 450 -o ornament.map catarrhini
```

[3] In fact, to annotate that these are the Western gorilla and the Borneo orang-utan, it would be simpler just to label the corresponding leaves accordingly, *i.e.*, `Gorilla_gorilla` and `Pongo_pygmaeus`. But this is just an example...



Ornaments and CSS can be combined:

```
$ nw_display -sr -S -w 450 -o ornament.map -c css.map catarrhini
```



libxml

If libxml is being used (see Appendix C), the handling of ornaments is more elaborate, in that some kinds of elements undergo special treatment. Besides positioning the ornament at the node's location and orienting it along the parent edge, which occur for all elements, the following occurs:

- `<text>` elements are nudged a few pixels from the parent edge, to make the text more readable. They are also transformed so that the text is aligned with the node's position, on both sides of the tree (this involves an additional $180°$ rotation on the left side of the tree).
- `<image>` elements are centered so that instead of having their top left corner at the node's position, they have the middle of the left side (this corresponds to vertical centering on an orthogonal tree). On the left side of the tree, they are also rotated and shifted so that they don't show upside-down.

If applicable, these transforms must be applied to each element separately. This means that the SVG snippet must be *parsed* (instead of just wrapped in a `<g>` element, as is the case when libxml is not being used), and we use libxml's XML parser.

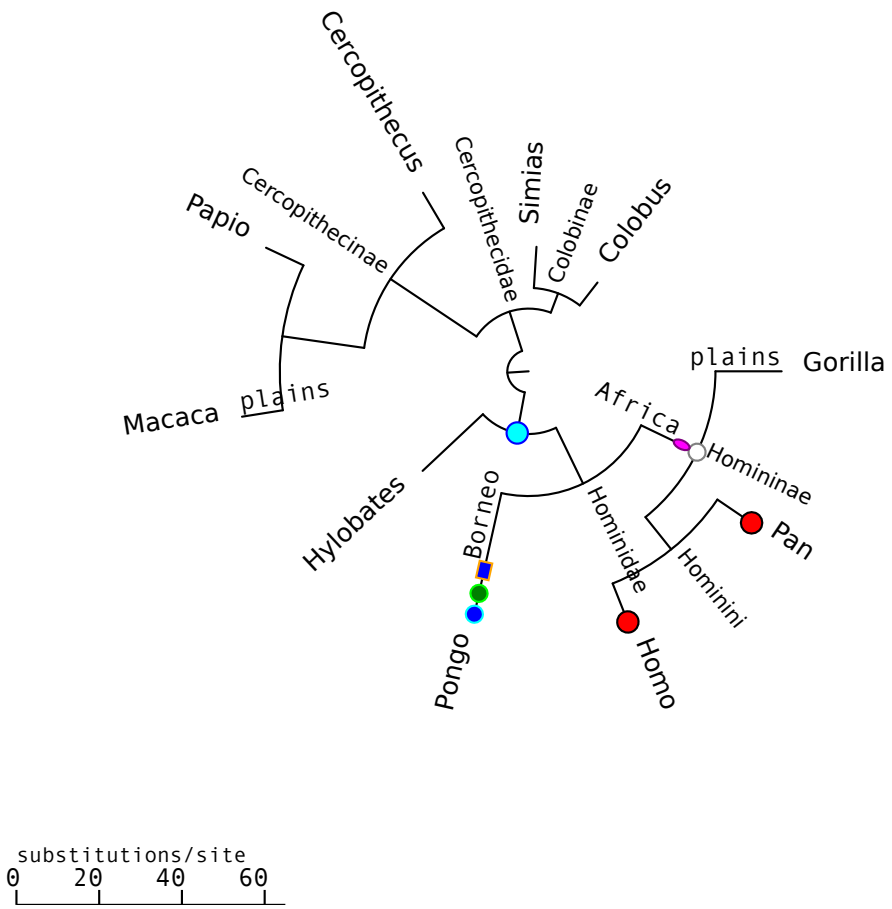
In the following file, the orang-utan (`Pongo`) and hominines have several ornaments, which are spaced out along the radial axis so that they don't overlap. This is

done simply by using the x attribute of texts and rectangles, as well as the cx attribute of circles and ellipses. Again, the node to be adorned lies at (0,0), x values lie on the radial axis, and y values are perpendicular to the x axis.

```
"<circle style='fill:red;stroke:black' r='5'/>" I Homo Pan
"<circle style='fill:cyan;stroke:blue' r='5'/>" C Homo Hylobates
<text>plains</text>                                I Gorilla
<text>plains</text>                                I Macaca
"<text style='font-style:italic' x='-25'>Borneo</text><circle r='4' style='
fill:blue;stroke:cyan'/><circle cx='-10' r='4' fill='green' stroke='lime'/>
<rect x='-25' y='-3' width='8' height='6' stroke='orange' fill='blue'/>" I
Pongo
"<text style='font-face:italic' x='-12'>Africa</text><circle r='4' style='s
troke:grey;fill:white'/><ellipse cx='-8' rx='4' ry='2' style='fill:magenta;
stroke:purple'/>" I Homininae
```

This gives the following:

```
$ nw_display -sr -w 500 -o orn_xml.map catarrhini
```



As hinted above, libxml also allows handling of images:

```
"<image width='100' height='100' xlink:href='100px-square-CH_cow_2.png'/>"
I Bovinae
"<image width='100' height='100' xlink:href='100px-square-Muskdeer.png'/>"
I Moschidae
"<image width='100' height='100' xlink:href='100px-square-Chevrotain.png'/>"
I Tragulidae
"<image width='100' height='100' xlink:href='100px-square-Pronghorn.png'/>"
I Antilocapridae
"<image width='100' height='100' xlink:href='100px-square-Giraffe.png'/>"
I Giraffidae
"<image width='100' height='100' xlink:href='100px-square-Irish_Goat.png'/>"
I Caprinae
"<image width='100' height='100' xlink:href='100px-square-Caribou.png'/>"
I Cervidae
"<image width='100' height='100' xlink:href='100px-square-Springbok.png'/>"
I Antilopinae
```

This gives the following (credits: all images are from Wikipedia):

```
$ nw_display -sr -w 500 -l 'fill:red' -o img_r.map pecora.nw
```

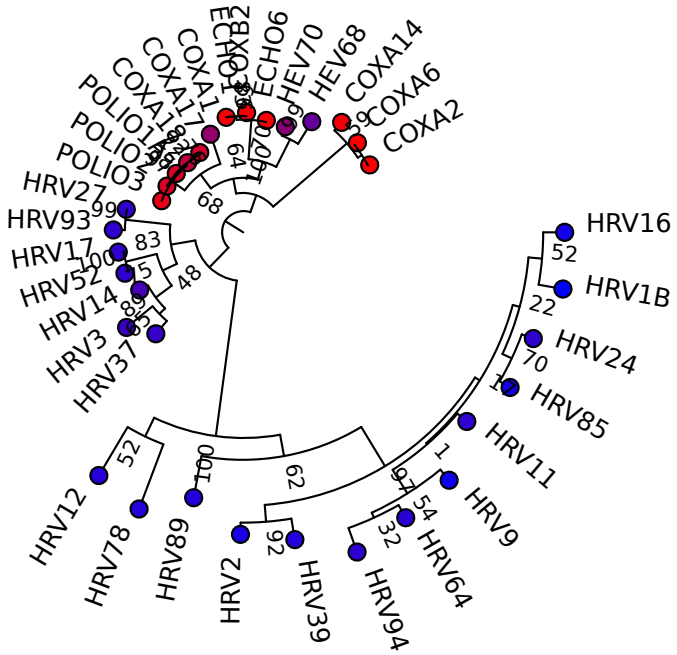
Example: Mapping GC Content

In a study of human rhinoviruses, I have produced a phylogenetic tree, `HRV_ingrp.nw`. I have also computed the GC content of the sequences, and mapped it into a gradient that goes from blue (33.3%) to red (44.5%). I used this gradient to produce a CSS map, `b2r.map`:

```
$ head -5 b2r.map

"<circle r='4' style='fill:#2500d9;stroke:black'/>"      I       HRV78
"<circle r='4' style='fill:#2700d7;stroke:black'/>"      I       HRV12
"<circle r='4' style='fill:#2100dd;stroke:black'/>"      I       HRV89
"<circle r='4' style='fill:#0000ff;stroke:black'/>"      I       HRV1B
"<circle r='4' style='fill:#1300eb;stroke:black'/>"      I       HRV16
```

in which the `fill` values are hexadecimal color codes along the gradient. Then:

```
$ nw_display -sr -S -w 450 -o b2r.map HRV_ingrp.nw
```



As we can see, the high-GC sequences are all found in the same main clade.

Multiple SVG Trees

Like all Newick Utilities, `nw_display` can handle multiple trees, even in SVG mode. The best way to do this was not evident: one can generate one file per tree (but then we break the rule that every program is a filter and so writes to standard output), or one can put all the trees in one SVG document (but then we have to impose tiling or some other arrangement), or one can just output one SVG document after another. This is what we do (this may change in the future). So if you have many trees in document `forest.nw`, you can say:

```
$ nw_display -s forest.nw > forest_svg
```

But `forest_svg` isn't valid SVG – it is a concatenation of many SVG documents. You can just extract them into individual files with `csplit`:

```
$ csplit -sz -f tree_ -b '%02d.svg' forest_svg '/<?xml/' {*}
```

This will produce one SVG file per tree in `forest.nw`, named `tree_01.svg`, `tree_02.svg`, etc.

`nw_display -s` stores its arguments

When run in SVG mode, `nw_display` "remembers" its arguments, that is, it puts them in an XML comment with the keyword `arguments`. It is then trivial to retrieve them:

```
$ nw_display -sr -S -w 450 -o b2r.map HRV_ingrp.nw > HRV_ingrp.svg

$ grep arguments  HRV_ingrp.svg

<!-- arguments: -sr -S -w 450 -o b2r.map HRV_ingrp.nw  -->
```

This is handy when one wants to re-use a set of options on another tree, especially after a while when one doesn't remember the exact values of the parameters, or which was the input tree, etc.

2.1.4 Options not Covered

`nw_display` has many options, and we will not describe them all here - all of them are described when you pass option `-h`. They include support for clickable images (with URLs to arbitrary Web pages), nudging labels, changing the root length, etc.

2.2 Displaying Tree Properties

`nw_stats` displays simple tree properties:

```
$ nw_stats catarrhini

Type:   Phylogram
#nodes: 19
#leaves:        10
#dichotomies:   9
#leaf labels:   10
#inner labels:  6
```

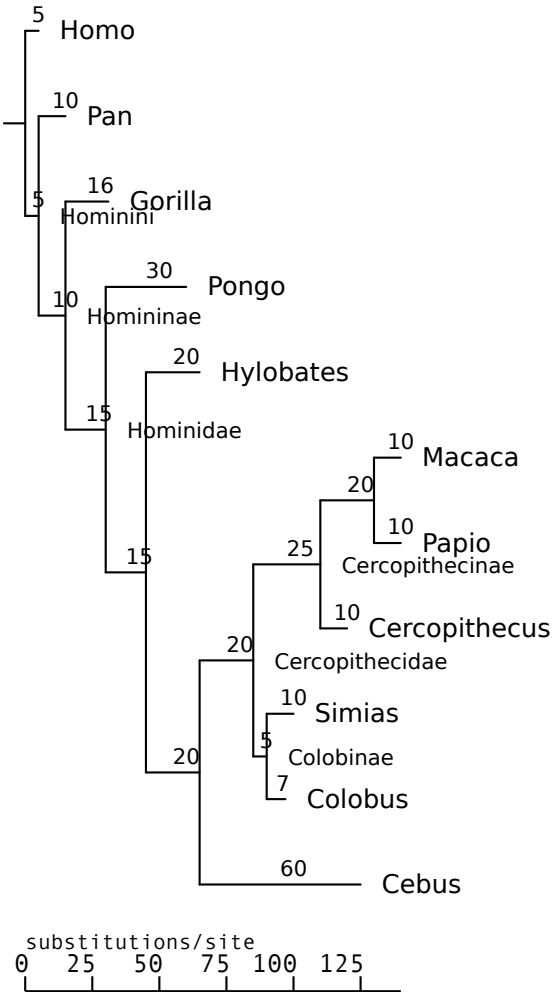
Option `-f l` causes the data to be printed linewise, without headers:

```
$ nw_stats -f l catarrhini

Phylogram       19      10      9       10      6
```

2.3 Rooting and Rerooting

Rooting transforms an unrooted tree into a rooted one, and rerooting changes a rooted tree's root. Some tree-building methods produce rooted trees (e.g., UPGMA), others produce unrooted ones (neighbor-joining, maximum-likelihood). The Newick format is implicitly rooted, in the sense that there is a 'top' node from which all other nodes descend. Some applications regard a tree with a trifurcation at the top node as unrooted.

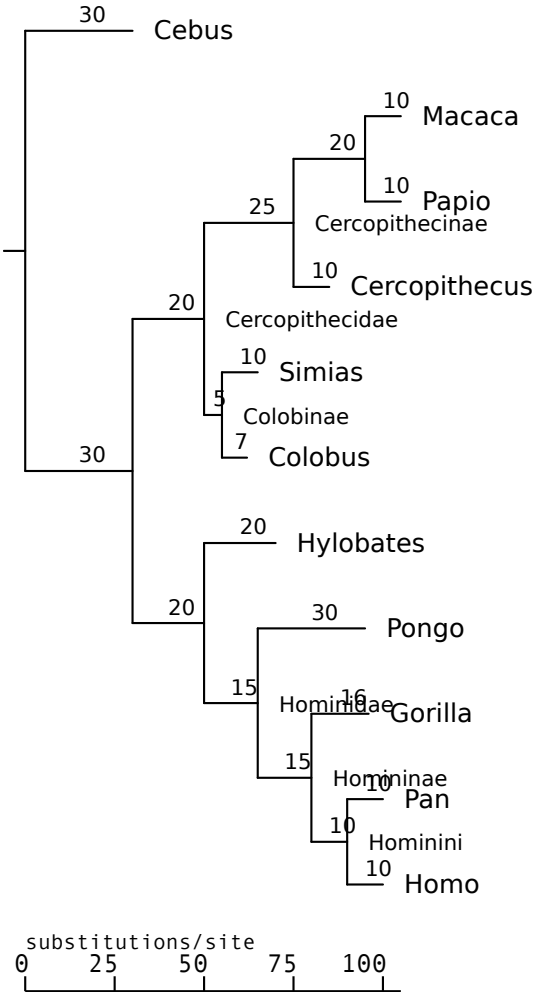
Rooting a tree is usually done by specifying an *outgroup*. In the simplest case, this is a single leaf. The root is then placed in such a way that one of its children is the outgroup, while the other child is the rest of the tree (sometimes known as the *ingroup*). Consider the following primate tree, `simiiformes_wrong`:



It is wrong because `Cebus`, which is a New World monkey (capuchin), should be the sister group of all the rest (Old World monkeys and apes, technically Catarrhini), whereas it is shown as the sister group of the macaque-colobus family, Cercopithecidae. We can correct this by re-rooting the tree using `Cebus` as outgroup:

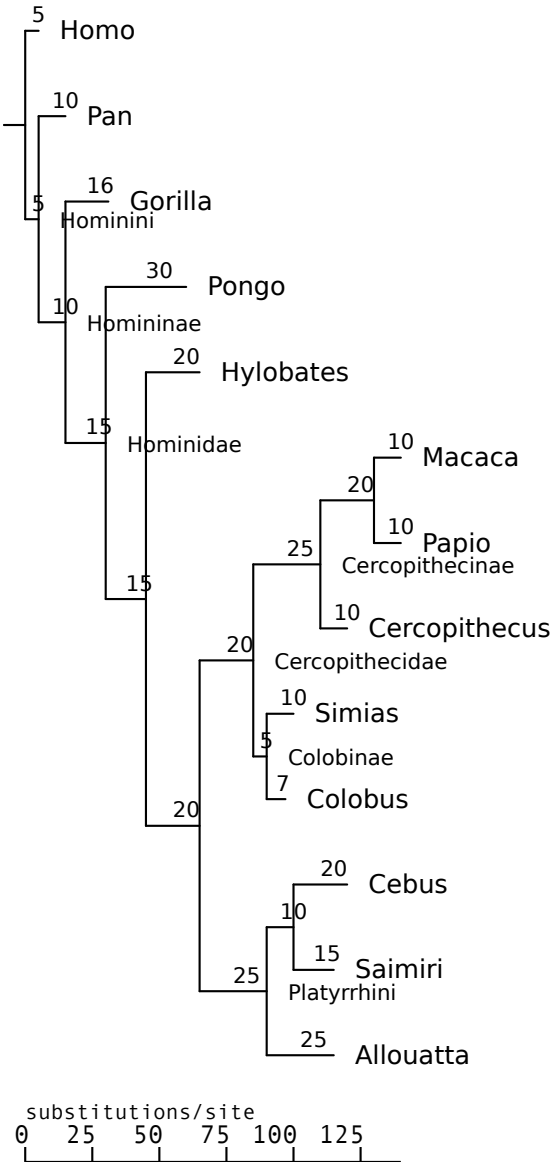
```
$ nw_reroot simiiformes_wrong Cebus | nw_display -s -
```

which produces:



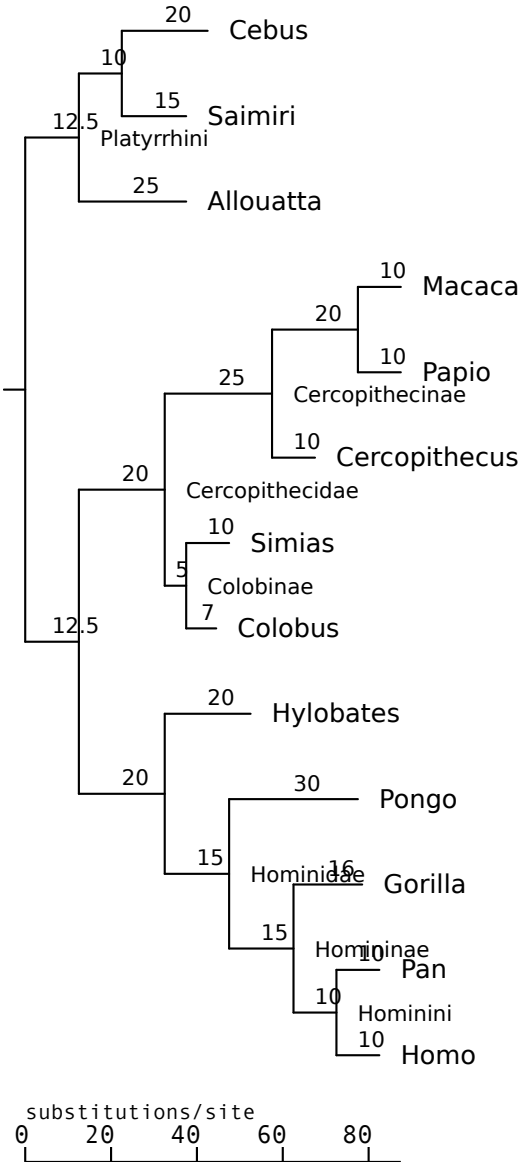
Now the tree is correct. Note that the root is placed in the middle of the ingroup-outgroup edge, and that the other branch lengths are conserved.

The outgroup does not need to be a single leaf. The following tree is wrong for the same reason as the one before, except that is has three New World monkey species instead of one, and they appear as a clade (Platyrrhini) in the wrong place:



We can correct this by specifying the New World monkey clade as outgroup:

```
$ nw_reroot simiiformes_wrong_3og Cebus Allouatta | nw_display -s -
```



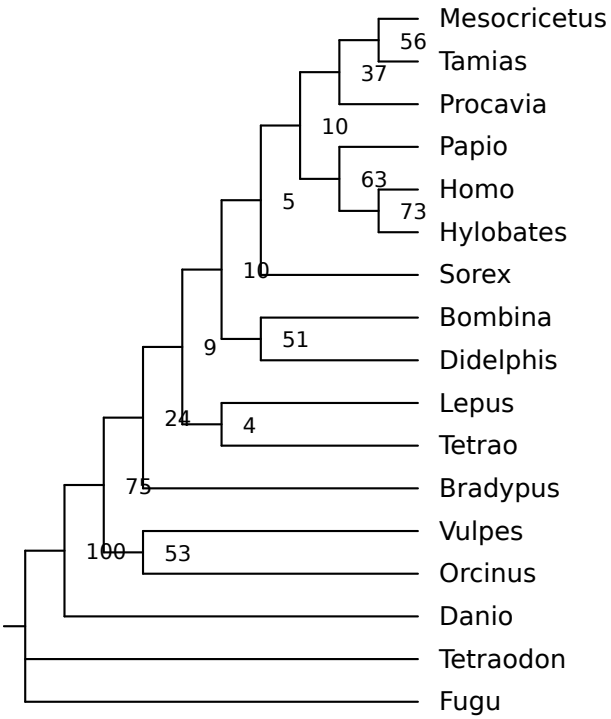
Note that I did not include all three New World monkeys, only `Cebus` and `Allouatta`. This is because it is always possible to define a clade using only two leaves. The result would be the same if I had included all three, though. You can use inner labels too, if there are any:

```
$ nw_reroot simiiformes_wrong_3og Platyrrhini
```

will reroot in the same way (not shown). Beware that inner labels are often used for support values (as in bootstrapping), which are generally not useful for defining clades.

2.3.1 Rerooting on the ingroup

Sometimes the desired species cannot be used for rooting, as their last common ancestor is the tree's root. For example, consider the following tree:



It is wrong because *Danio* (a ray-finned fish) is shown closer to tetrapods than to other ray-finned fishes (*Fugu* and *Tetraodon*). So we should reroot it, specifying that the fishes should form the outgroup. We could try this:

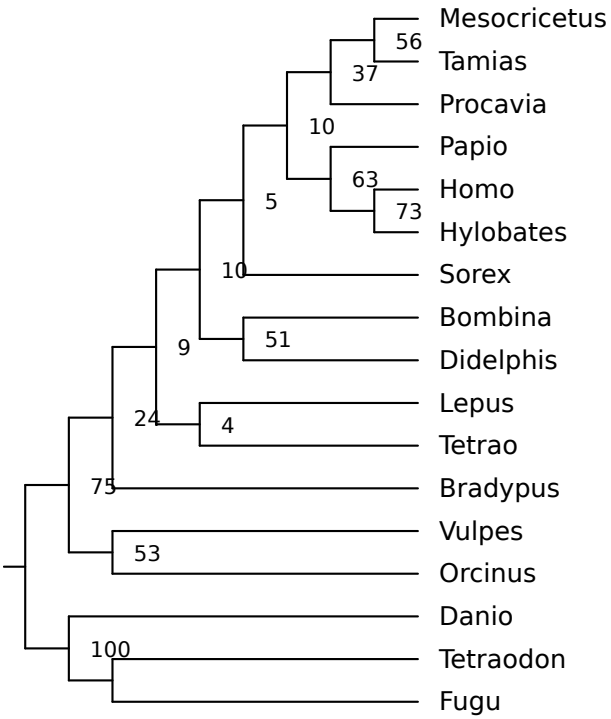
```
$ nw_reroot vrt1cg.nw Fugu Danio
```

But this will fail:

```
Outgroup's LCA is tree's root - cannot reroot. Try -l.
```

This fails because the last common ancestor of the two pufferfish is the root itself. The workaround in this case is to try the ingroup. This is done by passing option `-l` ("lax"), along with *all* species in the outgroup (this is because `nw_reroot` finds the ingroup by complementing the outgroup):

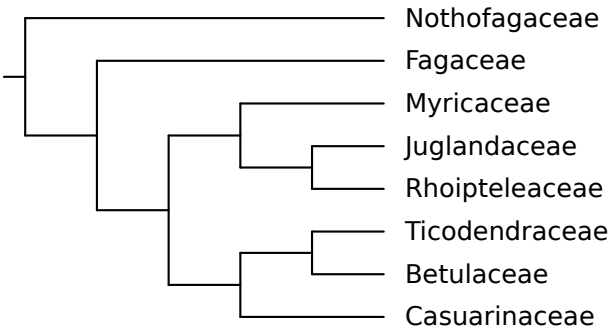
```
$ nw_reroot -l vrt1cg.nw  Danio Fugu Tetraodon | nw_display -s -v 20 -
```



To repeat: all outgroup labels were passed, not just the two normally needed to find the last common ancestor – since, precisely, we can't use the LCA.

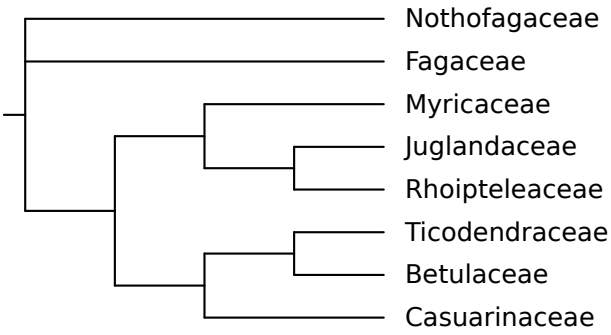
2.3.2 Derooting

Some programs insist on being passed an unrooted tree, e.g. if you want to supply your own tree to PhyML, it has to be "unrooted". Strictly speaking, Newick trees are always rooted, but there is a convention that if the root has three (or more) children, the tree is considered unrooted. You can deroot a tree (in this limited sense) by passing option `-d` to `nw_reroot`. Here is a rooted tree, `fagales.nw`



we can deroot it thus:

```
$ nw_reroot -d fagales.nw | nw_display -s -v 20 -
```

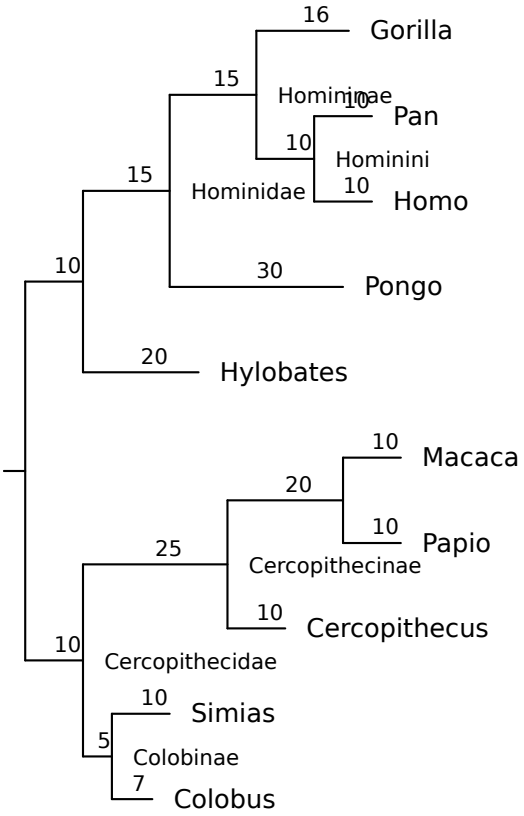


this works as follows. The program finds which of the root's two children (it is assumed to have two, otherwise the tree is already considered unrooted in the above sense) has more children than the other. This is considered the ingroup, and the LCA of the ingroup is spliced out from the tree, attaching its children directly to the root. In this example, the ingroup is the Fagaceae - Casuarinaceae clade, and the derooting results in Fagaceae being directly attached to the root, as is its sister clade (Myricaceae - Casuarinaceae).

2.4 Extracting Subtrees

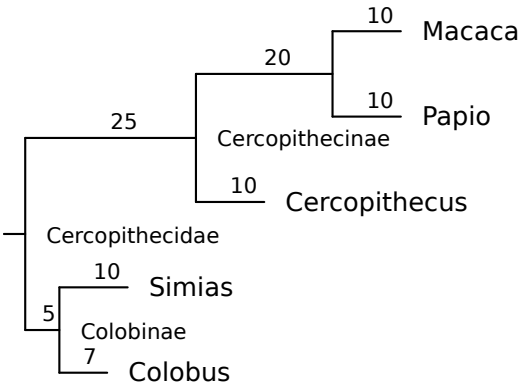
You can extract a clade (AKA subtree) from a tree with `nw_clade`. As usual, a clade is specified by a number of node labels, of which the program finds the last common ancestor, which unequivocally determines the clade (see Appendix A). We'll use the catarrhinian tree again for these examples:

```
$ nw_display -sS catarrhini
```



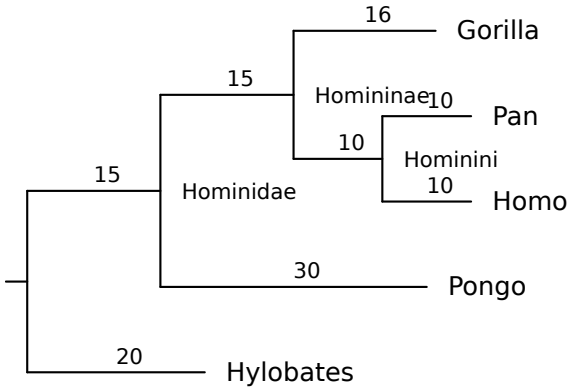
In the simplest case, the clade you want to extract has its own, unique label. This is the case of `Cercopithecidae`, so you can extract the whole cercopithecid subtree (Old World monkeys) using just that label:

```
$ nw_clade catarrhini Cercopithecidae | nw_display -sS -
```



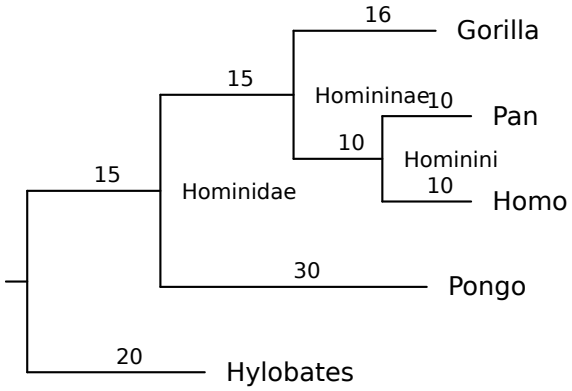
Now suppose I want to extract the apes subtree. These are the Hominidae ("great apes") plus the gibbons (*Hylobates*). But the corresponding node is unlabeled in our tree (it would be `Hominoidea`), so we need to specify (at least) two descendants:

```
$ nw_clade catarrhini Gorilla Hylobates | nw_display -sS -
```



The descendants do not have to be leaves: here I use `Hominidae`, an inner node, and the result is the same.

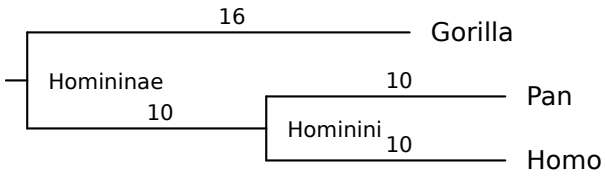
```
$ nw_clade catarrhini Hominidae Hylobates | nw_display -sS -
```



2.4.1 Monophyly

You can check if a set of leaves[4] form a monophyletic group by passing option `-m`: `nw_clade` will report the subtree only if the LCA has no descendant leaf other than those specified. For example, we can ask if the African apes (humans, chimp, gorilla) form a monophyletic group:

```
$ nw_clade -m catarrhini Homo Gorilla Pan | nw_display -sS -v 30 -
```



Yes, they do – it's subfamily Homininae. On the other hand, the Asian apes (orangutan and gibbon) do not:

[4] In future versions I may extend this to inner nodes

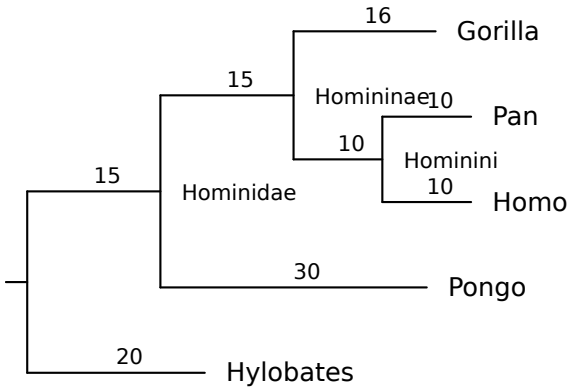
```
$ nw_clade -m catarrhini Hylobates Pongo

[no output]
```

Maybe early hominines split from orangs in South Asia before moving to Africa.

2.4.2 Context

You can ask for n levels above the clade by passing option `-c`:

```
$ nw_clade -c 2 catarrhini Gorilla Homo | nw_display -sS -
```

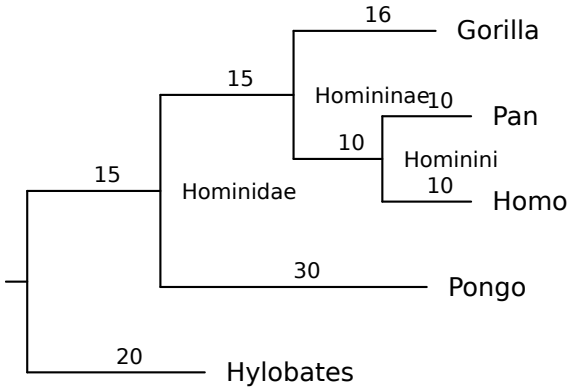


In this case, `nw_clade` computed the LCA of `Gorilla` and `Homo`, "climbed up" two levels, and output the subtree at that point. This is useful when you want to extract a clade with its nearest neighbor(s). I use this when I have several trees in a file and my clade's nearest neighbors aren't always the same.

2.4.3 Siblings

You can also ask for the siblings of the specified clade. What, for example, is the sister clade of the cercopithecids? Ask for `Cercopithecidae` and pass option `-s`:

```
$ nw_clade -s catarrhini Cercopithecidae | nw_display -sS -
```



Why, it's the good old apes, of course. I use this a lot when I want to get rid of the outgroup: specify the outgroup and pass `-s` – behold!, you have the ingroup.

Finally, although we are usually dealing with bifurcating trees, `-s` also applies to multifurcations: if a node has more than one sibling, `nw_clade` reports them all, in Newick order.

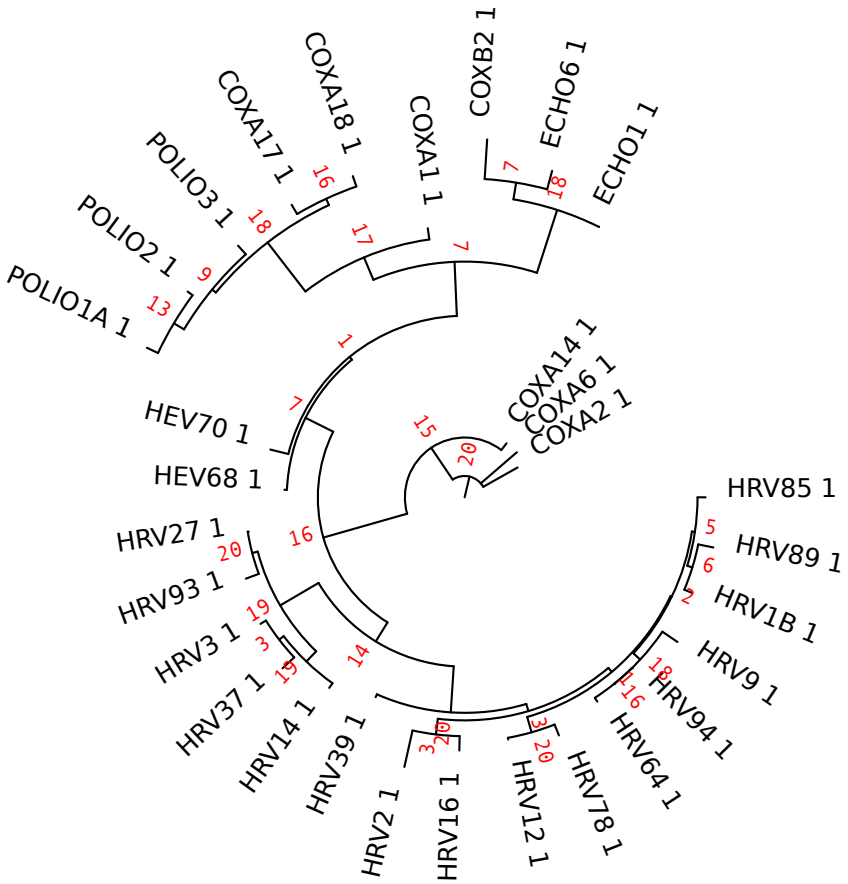
2.4.4 Limits

`nw_clade` assumes that node labels are unique. This should change in the future.

2.5 Computing Bootstrap Support

`nw_support` computes bootstrap support values from a target tree and a file of replicate trees. Say the target tree is in file `HRV.nw` and the replicates (20 of them) are in `HRV_20reps.nw`. You can attribute support values to the target tree like this:

```
$ nw_support HRV.nw HRV_20reps.nw \
| nw_display -sr -S -w 500 -i 'font-size:small;fill:red' -
```



In this case I have colored the support values red. Option `-p` uses percentages instead of absolute counts.

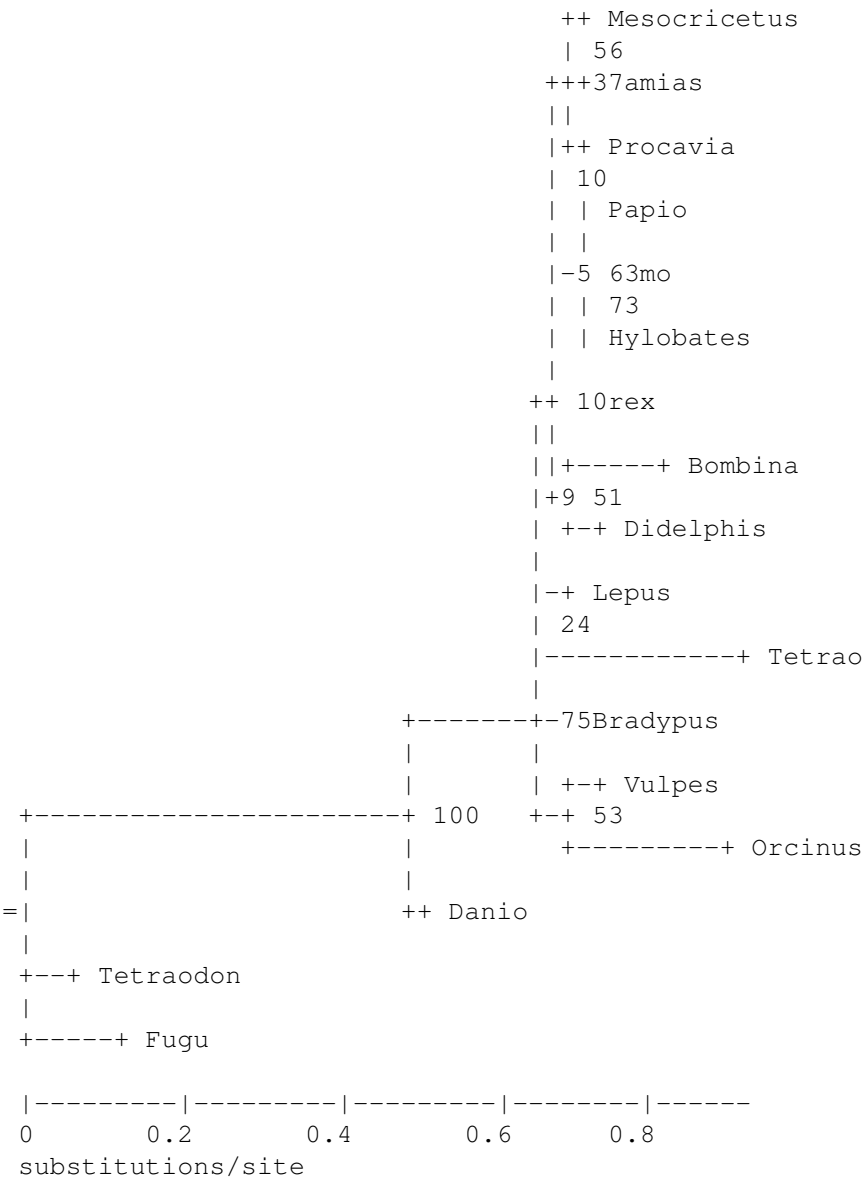
Notes

There are many tree-building programs that compute bootstrap support. For example, PhyML can do it, but for large tasks I typically have to distribute the replicates over several jobs (say, 100 jobs of 10 replicates each). I then collect all replicates files, concatenate them, and use `nw_support` to attribute the values to the target tree.

`nw_support` assumes rooted trees (it may as well, since Newick is implicitly rooted), and the target tree and replicates should be rooted the same way. Use `nw_reroot` to ensure this.

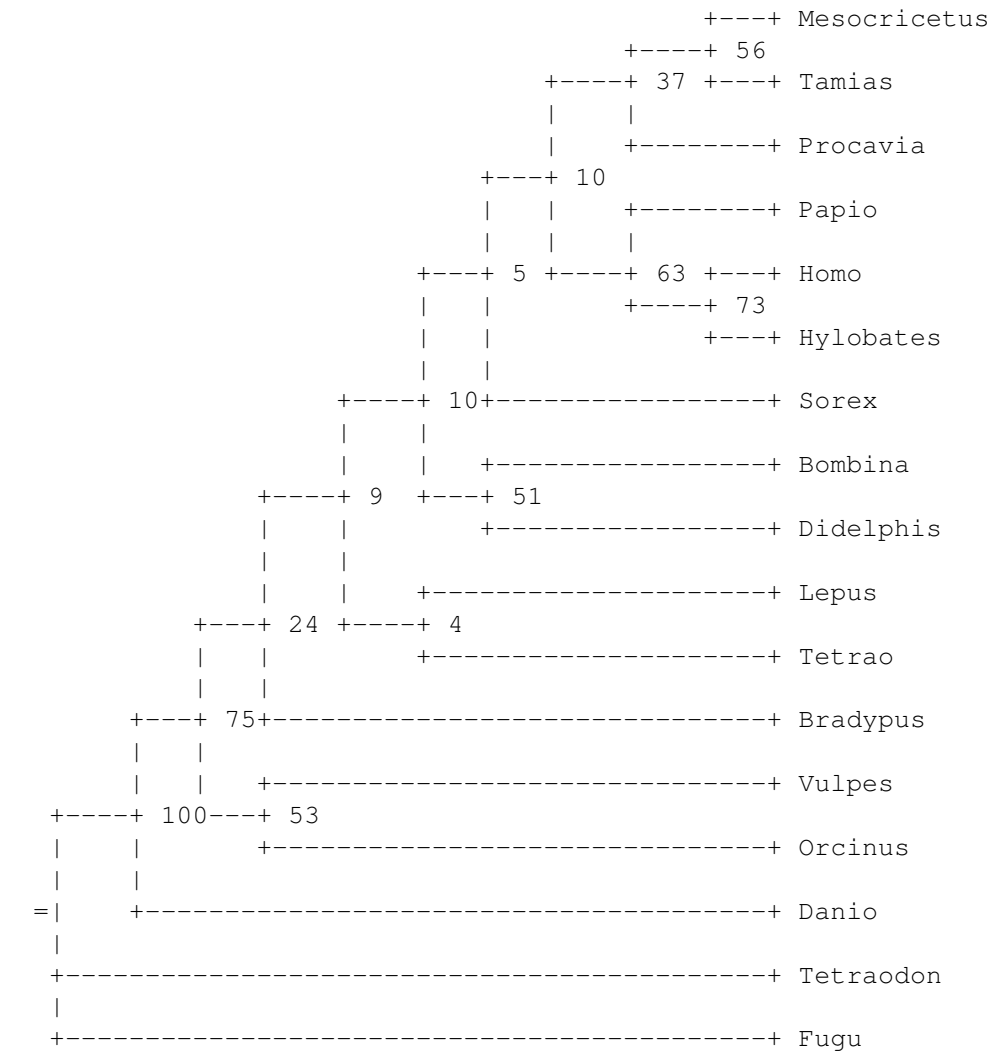
2.6 Retaining Topology

There are cases when one is more interested in the tree's structure than in the branch lengths, maybe because lengths are irrelevant or just because they are so short that they obscure the branching order. Consider the following tree, `vrt1.nw`:



Its structure is not evident, particularly in the upper half. This is because many branches are short in relation to the depth of the tree, so they are not well resolved. A better-resolved tree can be obtained by discarding branch lengths altogether:

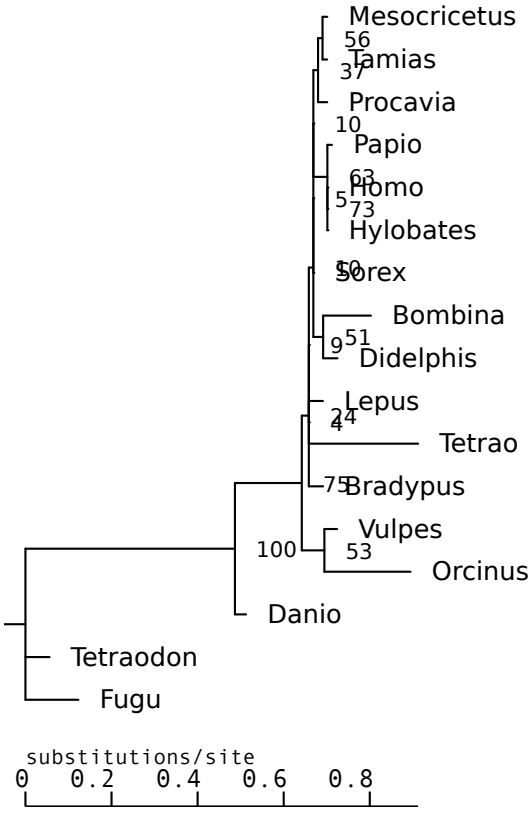
```
$ nw_topology vrt1.nw | nw_display -w 60 -
```



This effectively produces a *cladogram*, that is, a tree that represents ancestry relationships but not amounts of evolutionary change. The inner nodes are evenly spaced over the depth of the tree, and the leaves are aligned, so the branching order is more apparent.

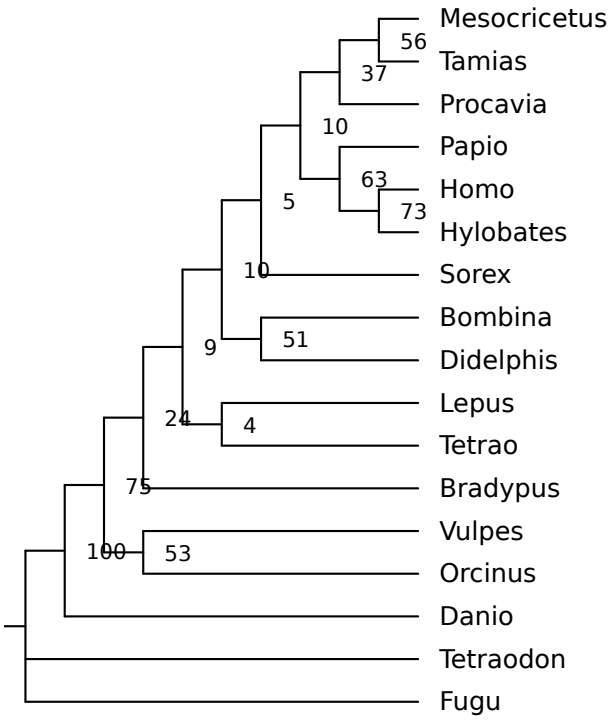
Of course, ASCII trees have low resolution in the first place, so I'll show both trees look in SVG. First the original:

```
$ nw_display -s -v 20 -b "opacity:0" vrt1.nw
```



And now as a cladogram:

```
$ nw_topology vrt1.nw | nw_display -s -v20 -
```



As you can see, even with SVG's much better resolution, it can be useful to display the tree as a cladogram.

`nw_topology` has the following options: `-b` keeps the branch lengths (obviously, using this option alone has no effect); `-I` discards inner node labels, and `-L` discards leaf labels. An extreme example is the following, which discards everything *but* topology:

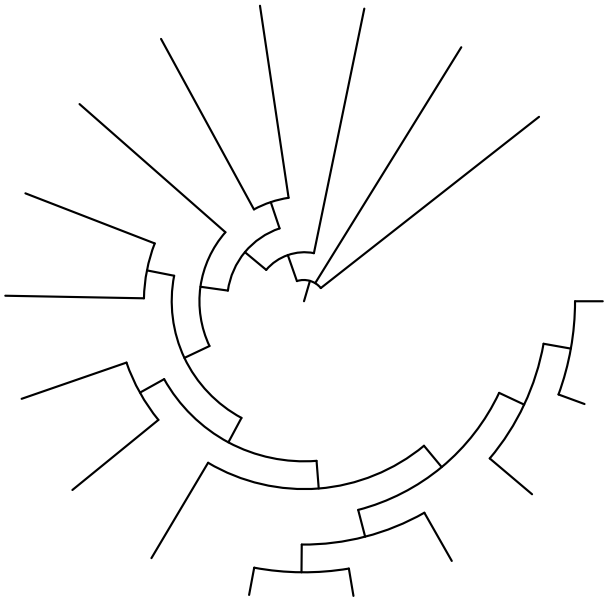
```
$ nw_topology -IL vrt1.nw
```

This produces the following tree, which is still valid Newick:

```
((((((((((,),),(,(,))),),(,)),(,)),),(,)),),,);
```

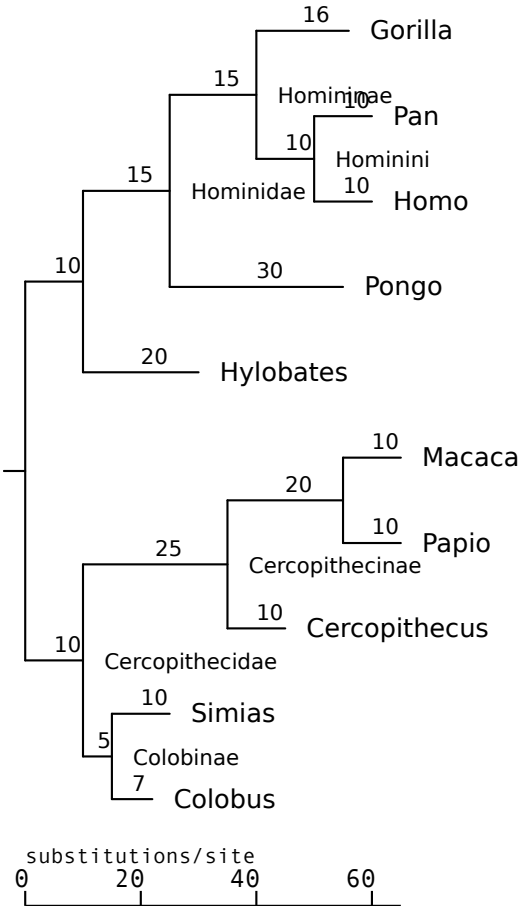
Let's look at it as a radial tree, for a change:

```
$ nw_topology -IL vrt1.nw | nw_display -sr -
```



2.7 Extracting Distances

`nw_distance` prints distances between nodes, in various ways. By default, it prints the distance from the root of the tree to each labeled leaf, in Newick order. Let's look at distances in the catarrhinian tree:




```
$ nw_distance catarrhini

56
60
60
55
30
65
65
45
25
22
```

This means that the distance from the root to `Gorilla` is 56, etc. The distances are in the same units as the tree's branch lengths – usually substitutions per site, but this is not specified in the tree itself. If the tree is a cladogram, the distances are expressed in numbers of ancestors. Option `-n` shows the labels:

```
$ nw_distance -n catarrhini
```

```
Gorilla 56
Pan     60
Homo    60
Pongo   55
Hylobates       30
Macaca  65
Papio   65
Cercopithecus   45
Simias  25
Colobus 22
```

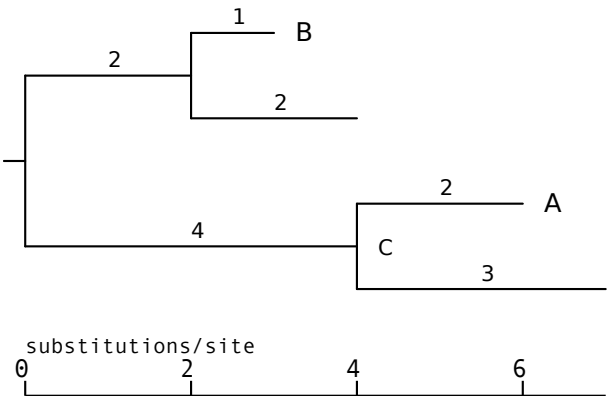
There are two main parameters to `nw_distance`: the *method* and the *selection*. The method determines how to compute the distance (from what node to what node), and the selection determines for which nodes the program is to compute distances. Let's look at examples.

2.7.1 Selection

In this section we will show the different selection types, using the default distance method (*i.e.*, from the tree's root – see below). The selection type is the argument to option `-s`. The nodes appear in the same order as in the Newick tree, except when they are specified on the command line (see below).

To illustrate the selection types, we need a tree that has both labeled and unlabeled leaves and inner nodes. Here it is

```
$ nw_display -s dist_sel_xpl.nw
```



We will use option `-n` to see the node labels.

All labeled leaves

The selection consists of all leaves with a label. This is the default, as leaves will mostly be labeled and we're generally more interested in leaves than inner nodes.

```
$ nw_distance -n dist_sel_xpl.nw

B       3
A       6
```

All labeled nodes

Option `-s l`. This takes all labeled nodes into account, whether they are leaves or inner nodes.

```
$ nw_distance -n -s l dist_sel_xpl.nw
```

```
B       3
A       6
C       4
```

All leaves

Option `-s f`. Selects all leaves, whether they are labeled or not.

```
$ nw_distance -n -s f dist_sel_xpl.nw
```

```
B       3
        4
A       6
        7
```

All inner nodes

Option `-s i`. Selects the inner nodes, labeled or not.

```
$ nw_distance -n -s i dist_sel_xpl.nw
```

```
        2
C       4
```

All nodes

Option `-s a`. All nodes are selected.

```
$ nw_distance -n -s a dist_sel_xpl.nw
```

```
B       3
        4
        2
A       6
        7
C       4
        0
```

Command line selection

The selection consists of the nodes whose labels are passed as arguments on the command line (after the file name). The distances are printed in the same order.

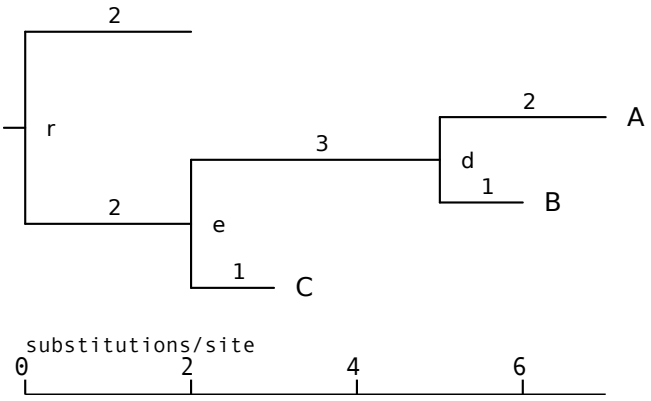
```
$ nw_distance -n dist_sel_xpl.nw A C
```

```
A       6
C       4
```

2.7.2 Methods

In this section we will take the default selection and vary the method. The method is passed as argument to option `-m`. I will also use an *ad hoc* tree to illustrate the methods:

```
$ nw_display -s dist_meth_xpl.nw
```



As explained above, the default selection consists of all labeled leaves – in our case, nodes `A`, `B` and `C`.

Distance from the tree's root

This is the default method: for each node in the selection, the program prints the distance from the tree's root to the node. This was shown above, so I won't repeat it here.

Distance from the last common ancestor

Option `-m l`. The program computes the LCA of all nodes in the selection (in our case, node `e`), and prints out the distance from that node to all nodes in the selection.

```
$ nw_distance -n -m l dist_meth_xpl.nw

A       5
B       4
C       1
```

Distance from the parent

Option `-m p`. The program prints the length of each selected node's parent branch.

```
$ nw_distance -n -m p dist_meth_xpl.nw

A       2
B       1
C       1
```

Matrix

Option `-m m`. Computes the pairwise distances between all nodes in the selection, and prints it out as a matrix.

```
$ nw_distance -n -m m dist_meth_xpl.nw

        A       B       C
A       0       3       6
B       3       0       5
C       6       5       0
```

2.7.3 Alternative formats

Option `-t` changes the output format. For matrix output, (`-m m`), the matrix is triangular.

```
$ nw_distance -t -m m dist_meth_xpl.nw

3
6       5
```

When labels are printed (option `-n`), the diagonal is shown

```
$ nw_distance -n -t -m m dist_meth_xpl.nw

A       0
B       3       0
C       6       5       0
```

For all other formats, the values are printed in a line, separated by TABs.

```
$ nw_distance -n -t -m p dist_meth_xpl.nw

A       B       C
2       1       1
```

2.8 Finding subtrees in other trees

`nw_match` tries to match a (typically smaller) "pattern" tree to one or more "target" tree(s). If the pattern matches the target, the target tree is printed. Intuitively, a pattern matches a target if one can superimpose it onto the target without "breaking" either. More accurately, the following happens (in both trees):

1. leaves with labels found in both trees are kept, the other ones are pruned
2. inner labels are discarded
3. both trees are ordered (as done by `nw_order`, see 2.15)
4. branch lengths are discarded

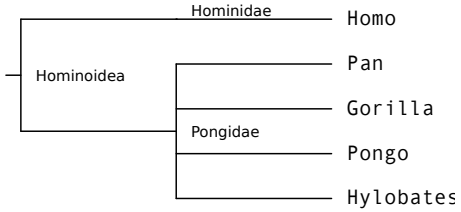
At this point, the modified pattern tree is compared to the modified target, and if the Newick strings are identical, the match is successful.

Example: finding trees with a specified subtree topology

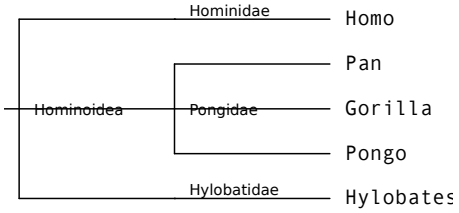
File `hominoidea.nw` contains seven trees corresponding to successive theories about the phylogeny of apes (these were taken from `http://en.wikipedia.org/wiki/Hominoidea`). Let us see which of them group humans and chimpanzees as a sister clade of gorillas (which is the current hypothesis).

Here are small images of each of the trees in `hominoidea.nw`:

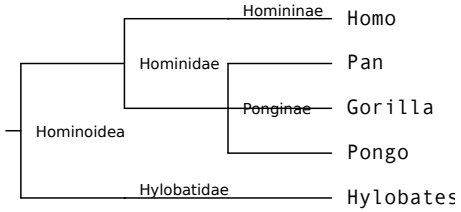
1 (until 1960)



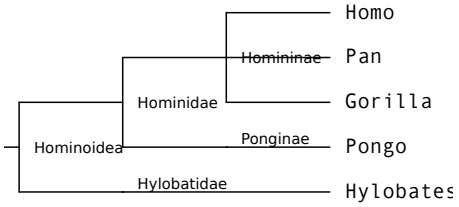
2 (Goodman, 1964)



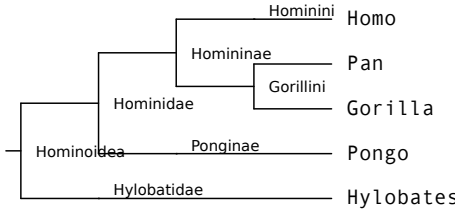
3 (gibbons as outgroup)



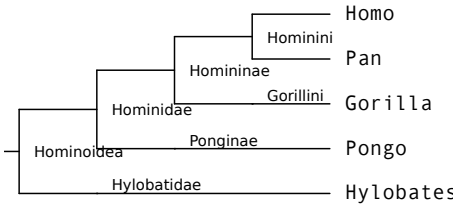
4 (Goodman, 1974: orangs as outgroup)



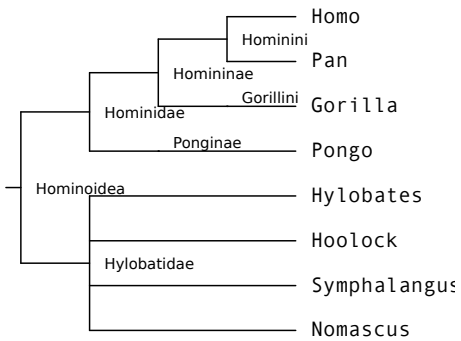
5 (resolving trichotomy)



6 (Goodman, 1990: gorillas as outgroup)

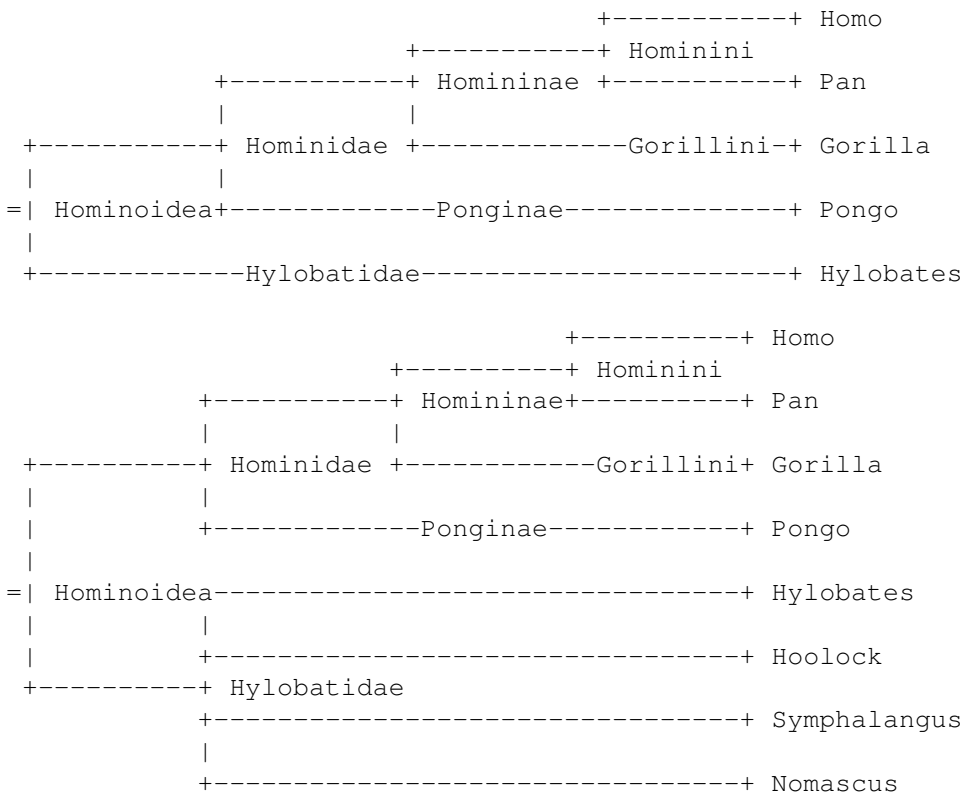


7 (split of *Hylobates*)



Trees #6 and #7 match our criterion, the rest do not. To look for matching trees in `hominoidea.nw`, we pass the pattern on the command line:

```
$ nw_match hominoidea.nw '(Gorilla,(Pan,Homo));' | nw_display -w 60 -
```



Note that only the pattern tree's topology matters: we would get the same results with pattern `((Homo,Pan),Gorilla);`, `((Pan,Homo),Gorilla);`, etc., but not with `((Gorilla,Pan),Homo);` (which would select trees #1, 2, 3, and 5. In future versions I might add an option for strict matching.

The behaviour of `nw_match` can be reversed by passing option `-v` (like `grep -v`): it will print trees that *do not* match the pattern. Finally, note that `nw_match` only works on leaf labels (for now), and assumes that labels are unique in both the pattern and the target tree.

2.9 Renaming nodes

Renaming nodes is the rather boring operation of changing a node's label. It can be done *e.g.* for the following reasons:

- building a higher-level tree (*i.e.*, a families tree from a tree of genera, etc)
- mapping one namespace into another (see 2.9.1)
- correcting wrong names

Renaming is done with `nw_rename`. This takes a *renaming map*, which is just a text file with the old and new names on the same line.

2.9.1 Breaking the 10-character limit in PHYLIP alignments

A technical hurdle with phylogenies is that some programs do not accept names longer than, say, 10 characters in the PHYLIP alignment. But of course, many scientific names

or sequence IDs are longer than that. One solution is to rename the sequences, before constructing the tree, using a numerical scheme, *e.g.*, *Strongylocentrotus purpuratus* → `ID_01`, etc. This means we have an alignment of the following form:

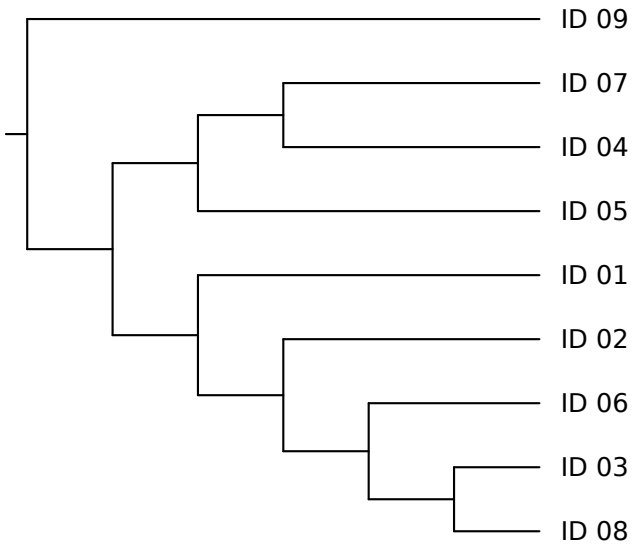
```
 154 259
ID_01     PTTSNSAPAL DAAETGHTSG ...
ID_02     SVSSHSVPAL DAAETGHTSS ...
...
```

together with a renaming map, `id2longname.map`:

```
ID_01 Strongylocentrotus_purpuratus
ID_02 Harpagofututor_volsellorhinus
...
```

The alignment's IDs are now sufficiently short, and we can use it to make a tree. It will look something like this:

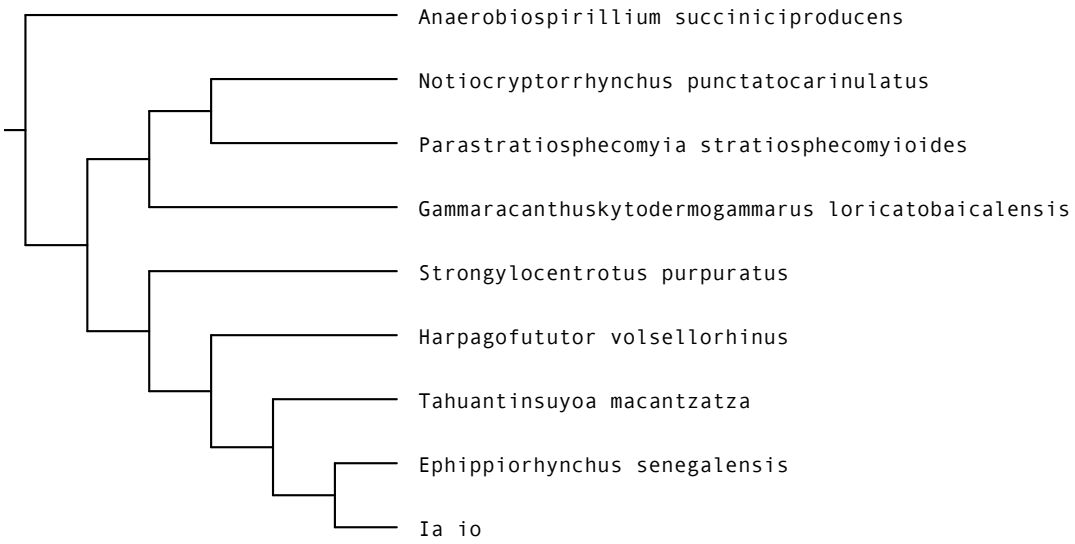
```
$ nw_display -s short_IDs.nw -v 30
```



Not very informative, huh? But we can put back the original, long names :

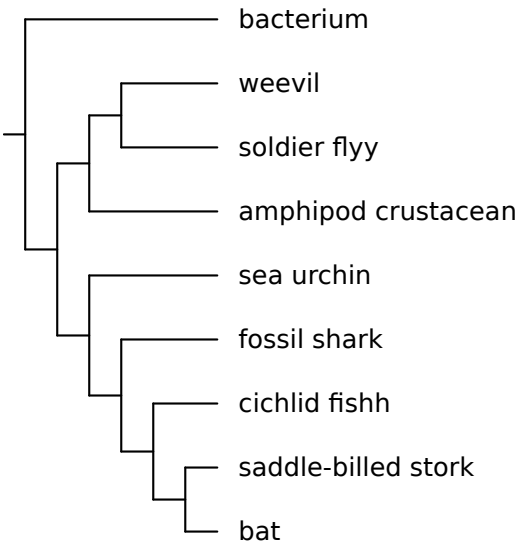
```
$ nw_rename short_IDs.nw id2longname.map \
| nw_display -s -l 'font-size:small;font-style:italic' -w 500 -v 30 -W 6 -
```

(option `-W` specifies the mean width of label characters, in pixels – use it when the default is wrong, as in this case with very long labels and small characters)



Now that's better…although exactly what these critters are might not be evident. Not to worry, I've made another map and I can rename the tree a second time on the fly:

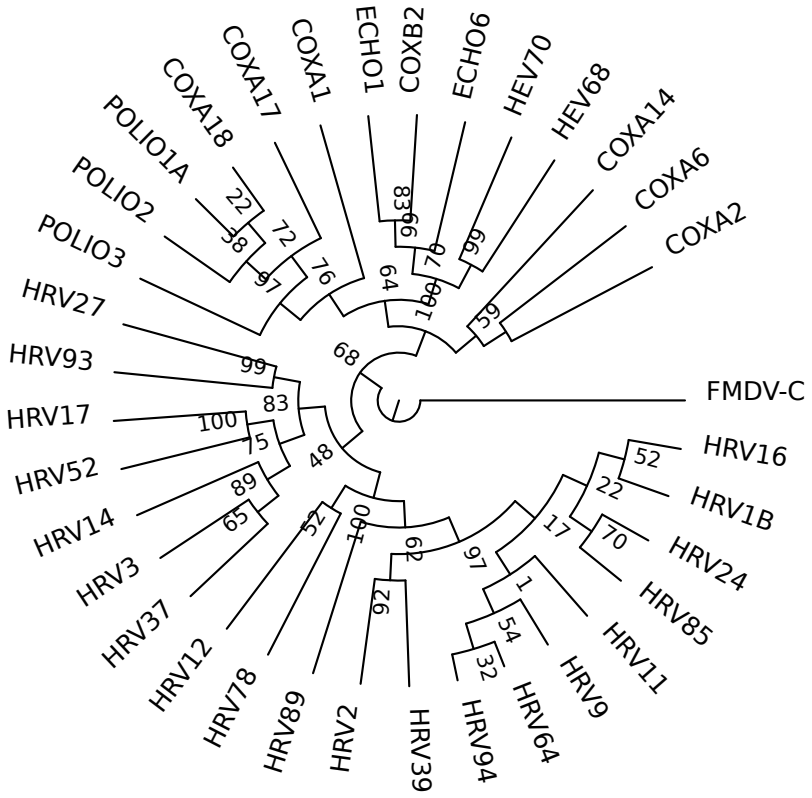
```
$ nw_rename short_IDs.nw id2longname.map \
| nw_rename - longname2english.map \
| nw_display -s -v 30 -W 10 -
```



2.9.2 Higher-rank trees

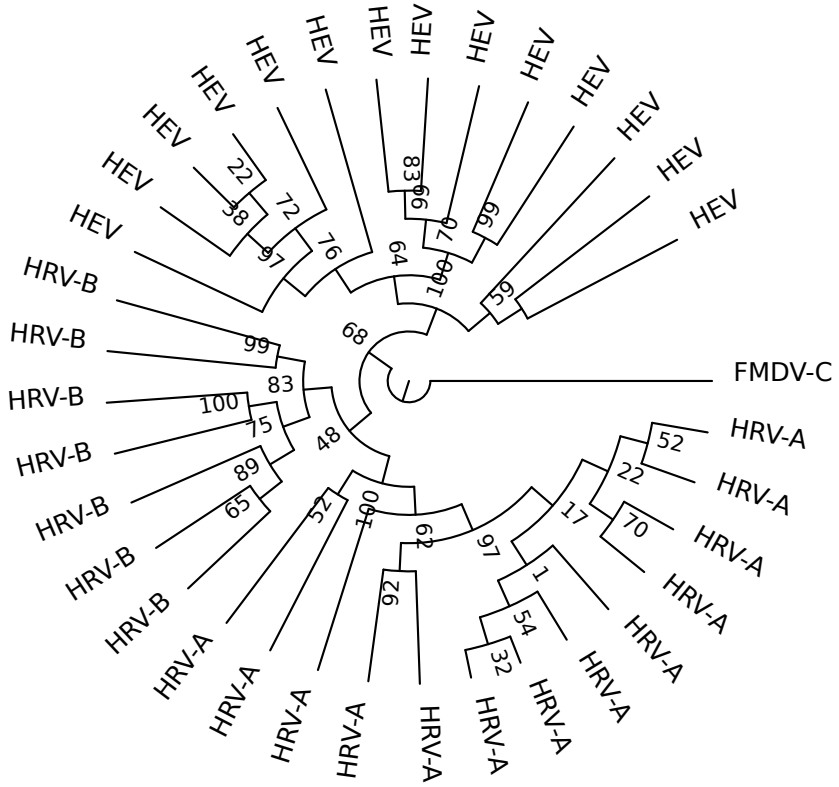
Here is a tree of a few dozen enterovirus and rhinovirus isolates. I show it as a cladogram (using `nw_topology`, see 2.6) because branch lengths do not matter here. I know that these isolates belong to three species in two genera: human rhinovirus A (`hrv-a`), human rhinovirus B (`hrv-b`, and enterovirus (`hev`).

```
$ nw_topology HRV_FMDV.nw | nw_display -sr -w 400 -
```



I want to see if the tree correctly groups isolates of the same species together. So I use a renaming map that maps an isolate name to its species (note by the way that the map file can have comment, whitespace-only and empty lines (which are all ignored), just like CSS maps (see 2.1.2):

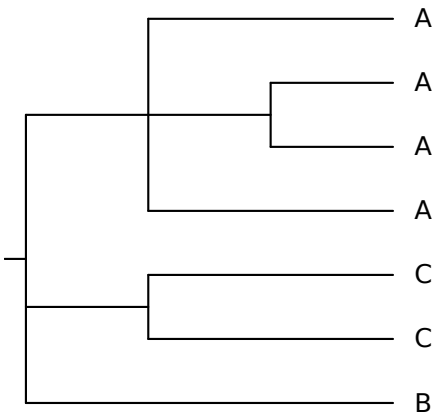
```
# These species belong to HRV-A
HRV16 HRV-A
HRV1B HRV-A
...
# HRV-B
HRV37 HRV-B
HRV14 HRV-B
...
# Enterovirus
POLIO1A HEV
COXA17 HEV

$ nw_rename HRV_FMDV.nw HRV_FMDV.map \
| nw_topology - | nw_display -srS -w 400 -
```



As we can see, it does. This would be even better if we could somehow simplify the tree so that clades of the same species were reduced to a single leaf. And, that's exactly what `nw_condense` does (see below).

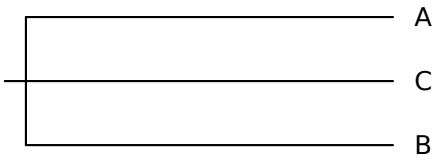
2.10 Condensing

Condensing a tree means reducing its size in a systematic, meaningful way (compare this to *pruning* (2.11) which arbitrarily removes branches, and to *trimming* (2.12) which cuts a tree at a specified depth). Currently the only condensing method available is simplifying clades in which all leaves have the same label - for example because they belong to the same taxon, etc. Consider this tree:



it has a clade that consists only of A, another of only C, plus a B leaf. Condensing will replace those clades by an A and a C leaf, respectively:

```
$ nw_condense condense1.nw | nw_display -s -w 200 -v 30 -
```

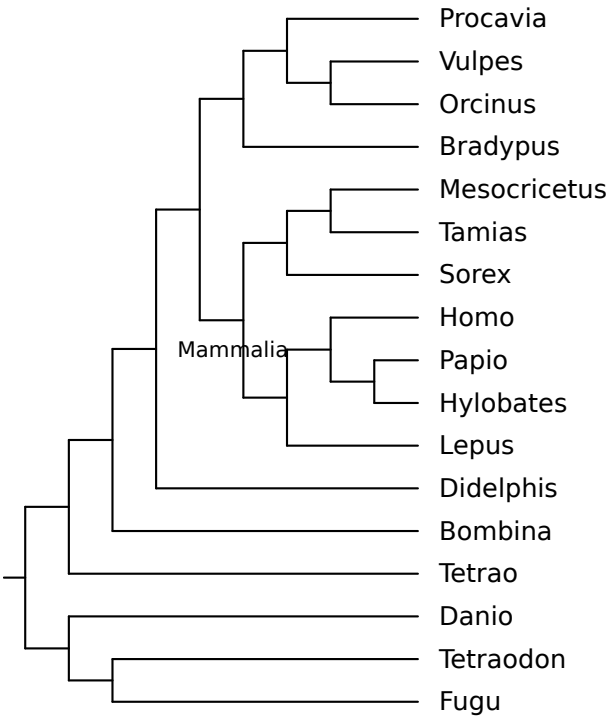


Now the A and B leaves stand for whole clades. The tree is simpler, but the information about the relationships of A, B and C is conserved, while the details of the A and C clades is not. A typical use of this is producing genus trees from species trees (or any higher-level tree from a lower-level one), or checking consistency with other data: For example condensing the virus tree of section 2.9.2 gives this:

The relationships between the species is now evident – as is the fact that the various isolates do cluster within species in the first place. This need not be the case, and renaming-then-condensing is a useful technique for checking this kind of consistency in a tree (see 3.1 for more examples).

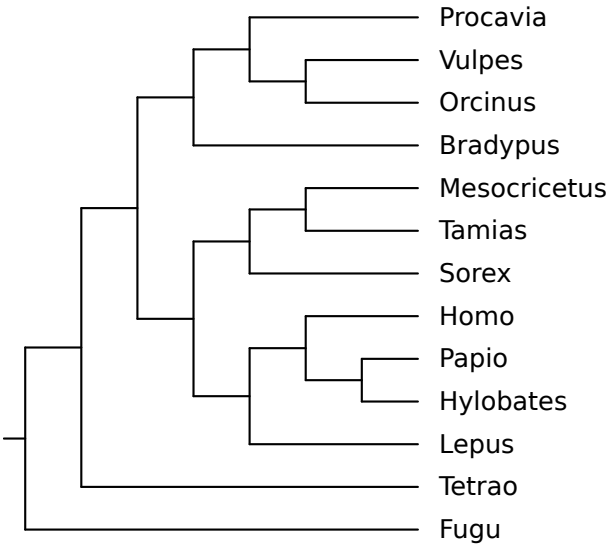
2.11 Pruning

Pruning is simply removing arbitrary nodes. Say you have the following tree (as it happens, it contains a glaring error since the sister clade of mammals is the amphibian rather than the bird):



and say you only need a subset of the species, perhaps because you want to compare this tree to another tree with fewer species. Specifically, let's say you don't need to show *Tetraodon*, *Danio*, *Bombina*, and *Didelphis*. You just pass those labels to `nw_prune`:

```
$ nw_prune vrt2_top.nw Tetraodon Danio Bombina Didelphis \
| nw_display -s -v 20 -
```

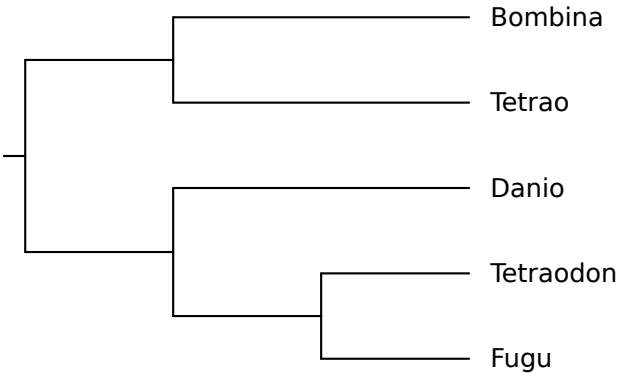


Note that each label is removed individually. The discarding of *Didelphis* is the cause of the disappearance of the node labeled Mammalia. And the embarrassing error is

hidden by the removal of *Bombina*.

You can also discard internal nodes, if they are labeled (in future versions it will be possible to discard a clade by specifying descendants, just like `nw_clade`). For example, you can discard the whole mammalian clade like this:

```
$ nw_prune vrt2_top.nw Mammalia | nw_display -s -
```



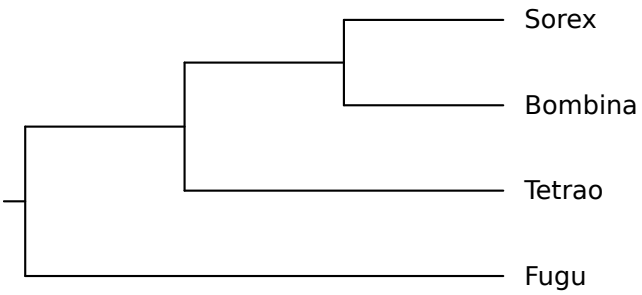
By the way, *Tetrao* and *Tetraodon* are not the same thing, the first is a bird (grouse), the second is a pufferfish.

2.11.1 Keeping selected Nodes

By passing option `-v` (think of `grep -v`), the nodes whose labels are passed are *kept*, and the other ones are discarded (except unlabeled nodes). And I really mean this: if a node's label is not on the command line, it goes away, even if it is an inner node - this can have surprising results.[5].

Suppose I think that the tree is unfairly biased towards mammals (and in a lesser way, actinopterygians), and want to keep only the following genera: *Fugu*, *Tetrao*, `Bombina`, *Sorex*. I could, of course, generate the list of all leaves that must go away, but it is easier to do this:

```
$ nw_prune -v vrt2_top.nw Mammalia Fugu Bombina Tetrao Sorex \
| nw_display -s -
```

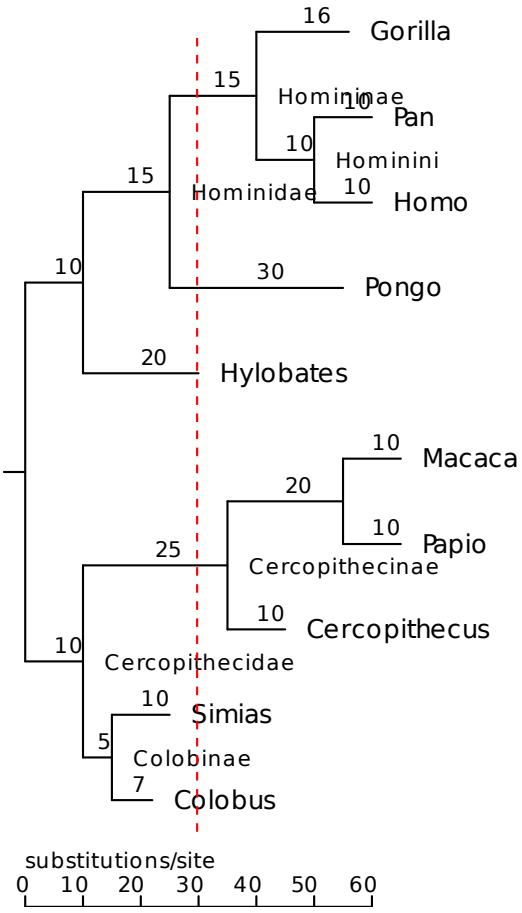


Note that I also passed `Mammalia`, for the reason discussed above: the node with this label would go away if I did not, resulting in a different tree (try it out).

[5]In future versions there will be an option for finer control of this behaviour

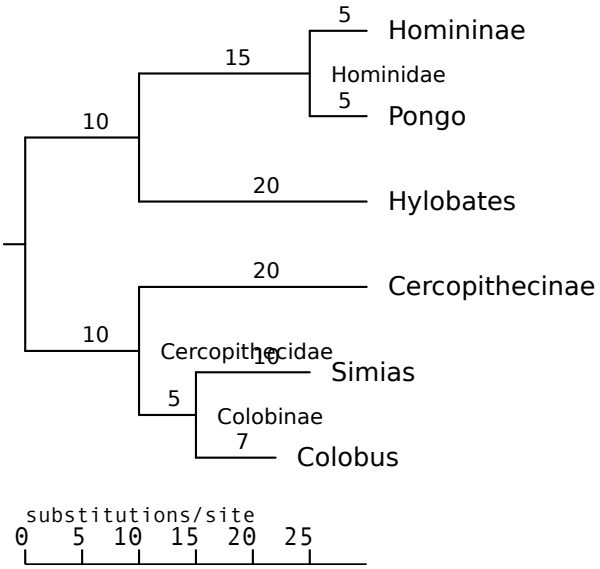
2.12 Trimming trees

Trimming a tree means cutting the nodes whose depth is larger than a specified threshold. Here is what will happen if I cut the catarrhini tree at depth 30:



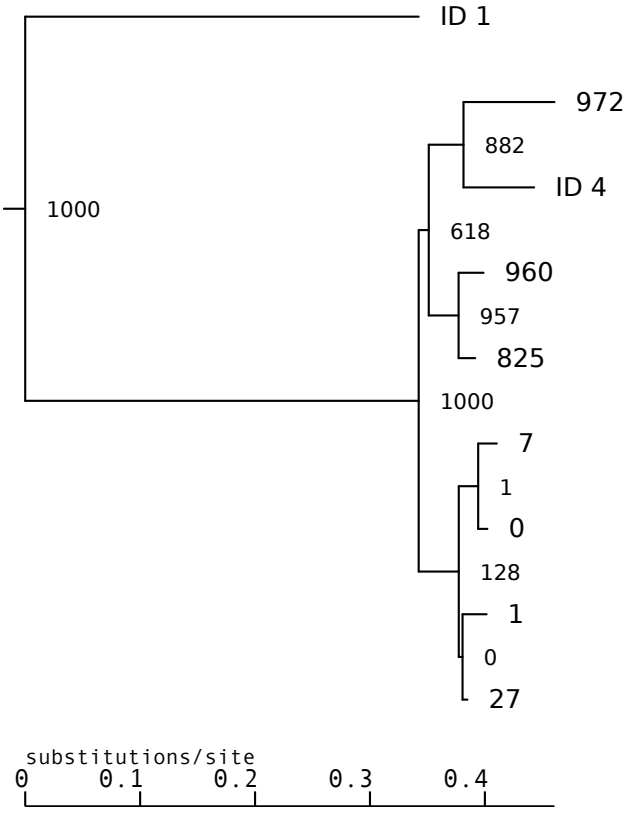
The tree will be "cut" on the red line, and everything right of it will be discarded:

```
$ nw_trim catarrhini 30 | nw_display -s -
```



By default, depth is expressed in branch length units – usually substitutions per site. By passing the `-a` switch, it is measured in number of ancestors, instead. Here are the first four levels of a huge tree (it has more than 1000 leaves):

```
$ nw_trim -a big.rn.nw 4 | nw_display -s -b 'opacity:0' -
```



The leaves with labels of the form `ID *` are also leaves in the original tree, the other leaves are former inner nodes whose children got trimmed. Their labels are the (absolute) bootstrap support values of those nodes. Note that the branch lengths are conserved. It is apparent that the ingroup's lower half has very poor support. This would be harder to see without trimming the tree, due to its huge size.

Trimming cladograms

By definition, cladograms do not have branch lengths, so you need to express depth in numbers of ancestors, and thus you want to pass `-a`.

2.13 Indenting

`nw_indent` reformats Newick on several lines, with one node per line, nodes of the same depth in the same column, and children nodes to the right of their parent. This shows the structure more clearly than the compact form, but since whitespace is ignored in the Newick format[6], the indented form is still valid. For example, this is a tree in compact form, in file `falconiformes`:

```
(Pandion:7,(((Accipiter:1,Buteo:1):1,(Aquila:1,Haliaeetus:2):1):2,
(Milvus:2,Elanus:3):2):3,Sagittarius:5,((Micrastur:1,Falco:1):3,
(Polyborus:2,Milvago:1):2):2);
```

[6] except between quotes

And this is the same tree, indented:

```
$ nw_indent falconiformes

(
  Pandion:7,
  (
    (
      (
        Accipiter:1,
        Buteo:1
      ):1,
      (
        Aquila:1,
        Haliaeetus:2
      ):1
    ):2,
    (
      Milvus:2,
      Elanus:3
    ):2
  ):3,
  Sagittarius:5,
  (
    (
      Micrastur:1,
      Falco:1
    ):3,
    (
      Polyborus:2,
      Milvago:1
    ):2
  ):2
);
```

The structure is much more clear, it is also relatively easy to edit manually in a text editor - while still being valid Newick.

Another advantage of indenting is that it is resistant to certain errors which would cause `nw_display` to fail.[7] For example, there is an error in this tree:

```
(Pandion:7,((Buteo:1,Aquila:1,Haliaeetus:2):2,(Milvus:2,
Elanus:3):2):3,Sagittarius:5((Micrastur:1,Falco:1):3,
(Polyborus:2,Milvago:1):2):2);
```

[7] This is because indenting is a purely lexical process, hence it does not need a syntactically correct tree.

yet it is hard to spot, and trying `nw_display` won't help as it will abort with a parse error. With `nw_indent`, however, you can at least look at the tree:

```
(
  Pandion:7,
  (
    (
      Buteo:1,
      Aquila:1,
      Haliaeetus:2
    ):2,
    (
      Milvus:2,
      Elanus:3
    ):2
  ):3,
  Sagittarius:5
  (
    (
      Micrastur:1,
      Falco:1
    ):3,
    (
      Polyborus:2,
      Milvago:1
    ):2
  ):2
);
```

While the error is not exactly obvious, you can at least view the Newick. It turns out there is a comma missing after `Sagittarius:5`.

The indentation can be varied by supplying a string (option `-t`) that will be used instead of the default (which is two spaces). If you want to indent by four spaces instead of two, you could say this:

```
$ nw_indent -t '    ' accipitridae

(
    (
        Buteo:1,
        Aquila:1,
        Haliaeetus:2
    ):2,
    (
        Milvus:2,
        Elanus:3
    ):2
):3;
```

Option `-t` can also be used to highlight indentation:

```
$ nw_indent -t '|   ' accipitridae

(
|   (
|   |   Buteo:1,
|   |   Aquila:1,
|   |   Haliaeetus:2
|   ):2,
|   (
|   |   Milvus:2,
|   |   Elanus:3
|   ):2
):3;
```

Now the indentation levels are easier to see, but at the expense of the tree no longer being valid Newick.

Finally, option `-c` ("compact") does the reverse: it removes all indentation and produces a compact tree. You can use this when you want to produce a compact Newick file after editing. For example, using Vim, after loading a Newick tree I do

```
gg!}nw_indent -
```

to indent the file, then I edit it, then compact it again:

```
gg!}nw_indent -c -
```

2.14 Extracting Labels

To get a list of all labels in a tree, use `nw_labels`:

```
$ nw_labels catarrhini

Gorilla
Pan
Homo
Hominini
Homininae
Pongo
Hominidae
Hylobates
Macaca
Papio
Cercopithecus
Cercopithecinae
Simias
Colobus
Colobinae
Cercopithecidae
```

The labels are printed out in Newick order. To get rid of internal labels, use `-I`:

```
$ nw_labels -I catarrhini

Gorilla
Pan
Homo
Pongo
Hylobates
Macaca
Papio
Cercopithecus
Simias
Colobus
```

Likewise, you can use `-L` to get rid of leaf labels, and with `-t` the labels are printed on a single line, separated by tabs (here the line is folded due to lack of space).

```
$ nw_labels -tL catarrhini

Hominini        Homininae       Hominidae       Cercopithecinae Colobinae
Cercopithecidae
```

If you just want the root's label, pass `-r`. In conjunction with `nw_clade` (see 2.4), this is handy to get support values of nodes defined by their descendants. For example, the following shows the support value of the clade defined by `HRV39` and `HRV85` in a virus tree similar to that of 2.1.3:

```
$ nw_clade HRV_cg.nw HRV85 HRV39 | nw_labels -r -

100
```

2.14.1 Counting Leaves in a Tree

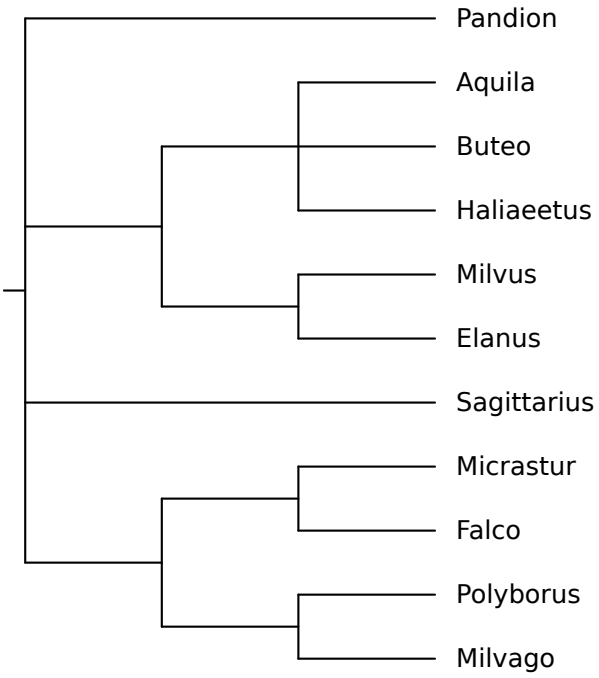
A simple application of `nw_labels` is a leaf count (assuming each leaf is labeled - Newick does not require labels):

```
$ nw_labels -I catarrhini | wc -l

10
```

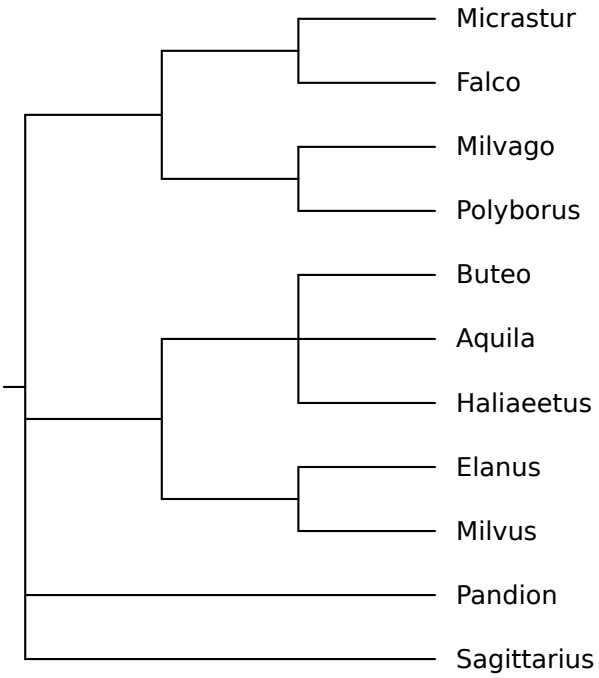
2.15 Ordering Nodes

Two trees that differ only by the order of children nodes within the parent convey the same biological information, even if the text (Newick) and graphical representations differ. For example, files `falconiformes_1` and `falconiformes_2` are different, and they yield different images:

```
$ nw_display -sS -v 30 falconiformes_1
```




```
$ nw_display -sS -v 30 falconiformes_2
```



But do they represent different phylogenies? In other words, do they differ by more than just the ordering of nodes? To check this, we pass them to `nw_order` and use `diff` to compare the results[8]:

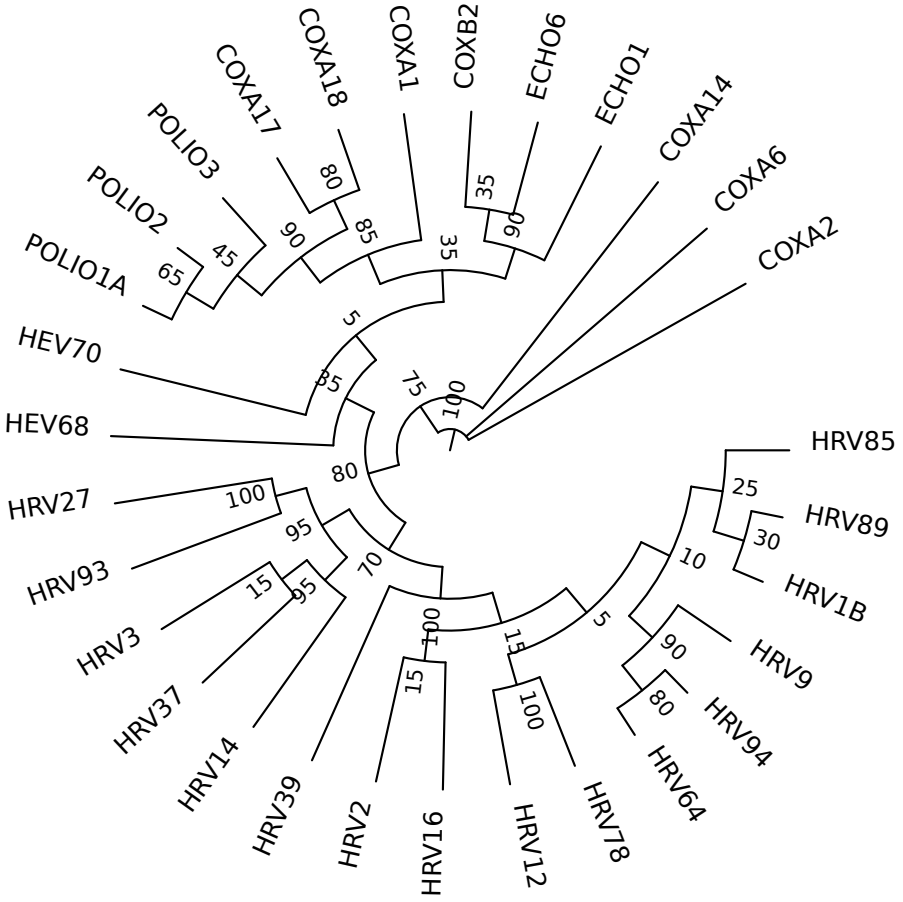
```
$ nw_order falconiformes_1 > falconiformes_1.ord.nw ; \
nw_order falconiformes_2 > falconiformes_2.ord.nw ; \
diff -s falconiformes_1.ord.nw falconiformes_2.ord.nw

Files falconiformes_1.ord.nw and falconiformes_2.ord.nw are identical
```

So, after ordering, the trees are the same: they tell the same biological story. Note that these trees are cladograms. If you have trees with branch lengths, this approach will only work if the lengths are identical, which may or may not be what you want. You can get rid of the branch lengths using `nw_topology` (see 2.6).

2.15.1 Variants

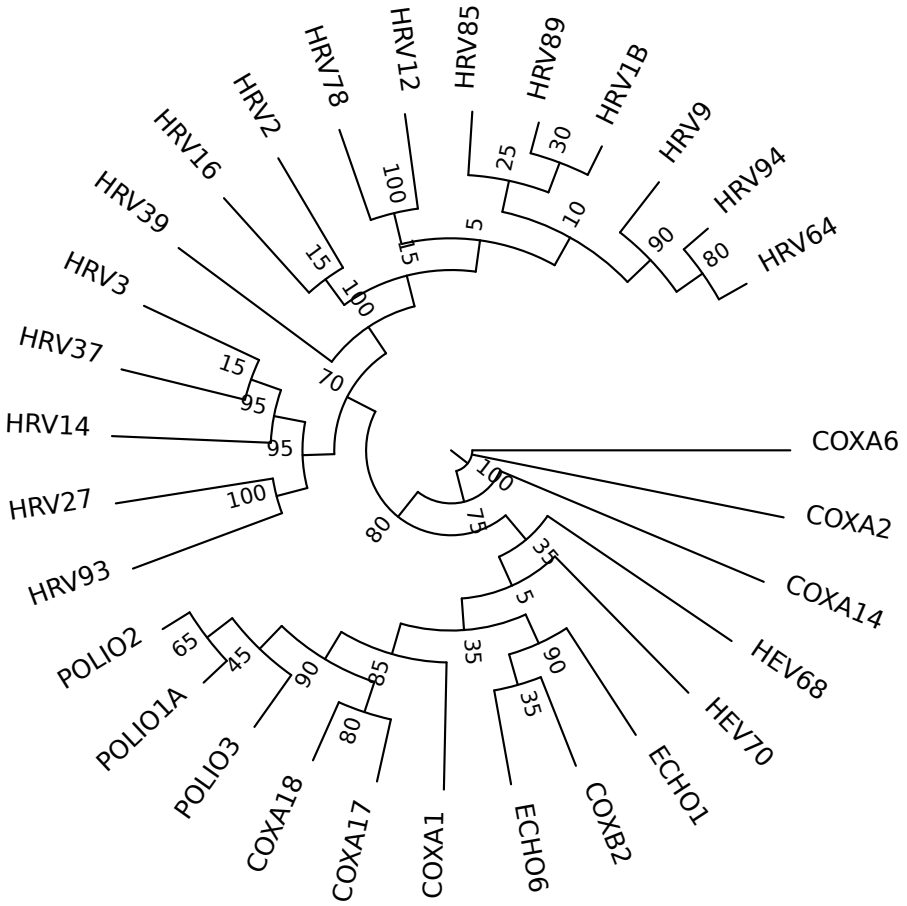
Other ordering criteria are available through option `-c`. To order a tree by number of descendants (*i.e.*, "light" nodes before "heavy" nodes), pass `-c n`. This has the effect of "ladderizing" trees which are heavily imbalanced. Consider this tree:

[8] One could also compute a checksum using `md5sum`, etc



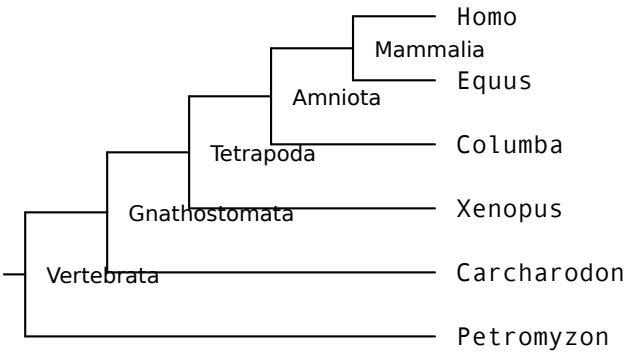
Here is the same tree, reordered by number of descendants: light nodes appear before (clockwise) heavy nodes:

```
$ nw_order -c n HRV_cg.nw \
| nw_display -sSr -b 'visibility:hidden' -v 30 -w 450 -
```



De-ladderizing

Incidentally, ”ladderizing” a tree may not be a good idea, because it lends itself to misinterpretations. For example, the following tree leads some people (including professional biologists, apparently [1]) to the following mistakes:



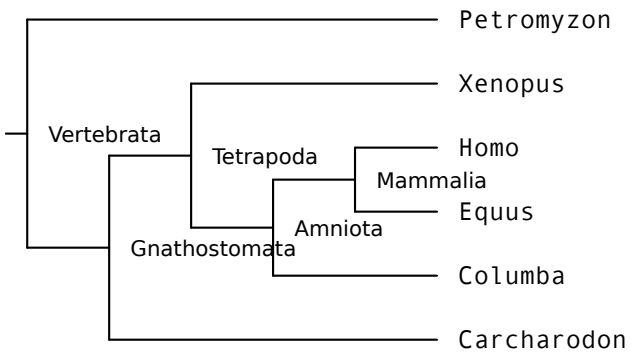
- there is a ”chain of being” with ”higher” and ”lower” organisms, with (surprise!) humans at the top; ”higher” can be interpreted in various ways, including ”more

perfect", or "more evolved" or even morally superior. This is known as the *scala naturæ* fallacy.

- there is a "main line" that progressively leads to (surprise!) humans, with "offshoots" along the way – lowly lampreys branching out first, then sharks, etc.
- early-branching species (this is itself an error) are "primitive": in our case, it would mean that the last common ancestor of lampreys and humans was a lamprey (or very like one); that the LCA of humans and sharks was very much like a modern shark, etc.

For a comprehensive discussion of errors in tree-thinking, see [2]. Now, to prevent these errors, one can reorder the tree in such a way as to remove the ladder. This is done by passing `-c d`. The tree is topologically identical, so it tells the same biological story:

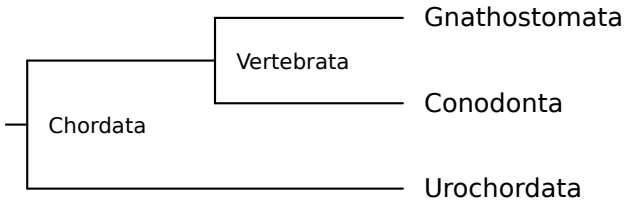
```
$ nw_order -c d scala.nw \
| nw_display -s -v 30 -l 'font-style:italic' -
```



It is less easy now to construe that there is a chain of being, or that evolution is progressive, etc. Unfortunately, some folks take the new tree to mean that humans are more closely related to amphibians (*Xenopus*) than to birds (*Columba*). There is no substitute to actually learn how to interpret trees, I'm afraid.

2.16 Converting Node Ages to Durations

Sometimes you have information about the *age* of a node rather than the length of its branch. Consider the following phylogeny of major chordate groups:



Suppose we have the following information about the age of certain events (not that it matters, I found it in Wikipedia and the Palaeos website (`www.palaeos.com`):

event	**age** (million years ago)
split of vertebrates into gnathostomes and conodonts	530
extinction of conodonts	200
split of chordates into vertebrates and urochordates	540

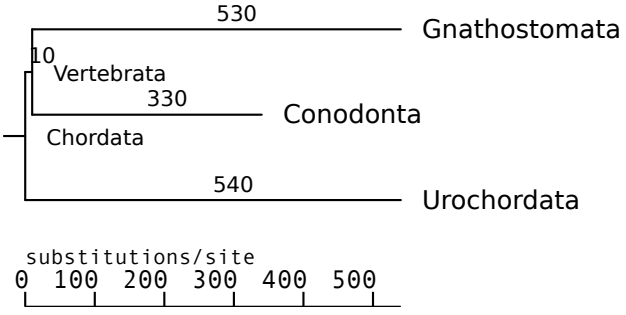
We can use the "branch length" field of Newick to specify ages, like this:

```
$ cat age.nw

(
        (
                Gnathostomata,
                Conodonta:200
        )Vertebrata:530,
        Urochordata
)Chordata:540;
```

The "branch length" of `Vertebrata` becomes `530`, because the vertebrate lineage split into conodonts and gnathostomes at that date[9]. Note that the `Gnathostomata` leaf has no age: this means that there are still living gnathostomes (such as you and I[10]); the same goes for urochordates. In other words, a leaf with no age has an implicit age of zero. This also ensures that the leaves of the extant taxa are aligned. The `Conodonta`, on the other hand, has an age although it is a leaf: this is because the conodonts went extinct, around `200` Mya.

Now, if we were to display this tree without further ado, it would be nonsense. We have to convert the ages into durations, and this is the function of `nw_duration`:

```
$ nw_duration age.nw | nw_display -s -
```

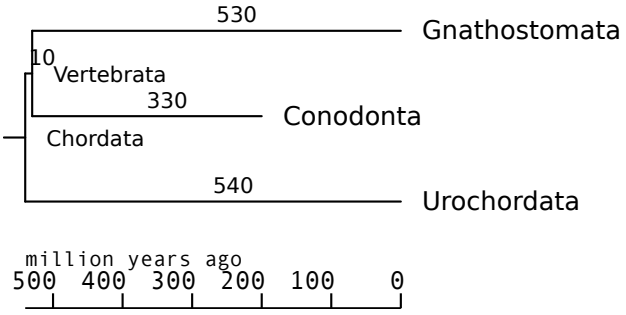


We can improve the graph by supplying option `-t` to `nw_display`: this aligns the origin of the scale bar with the leaves and counts backwards. To top it off, we'll specify the units as million years ago:

```
$ nw_duration age.nw | nw_display -s -t -u 'million years ago' -
```

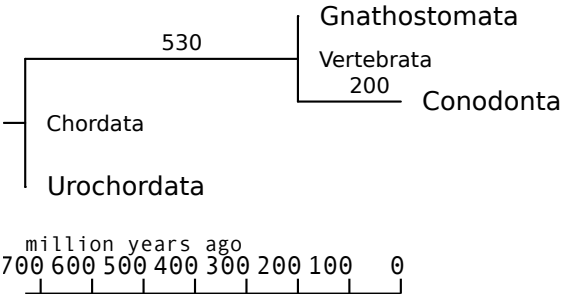
[9] Of course there are other branches in the vertebrate lineage, but they are not shown in this tree

[10] Well, at least *I* am one



Since you're curious, here is what the `age.nw` tree looks like if we "forget" to run it through `nw_duration`:

```
$ nw_display -s -t -u 'million years ago' age.nw
```

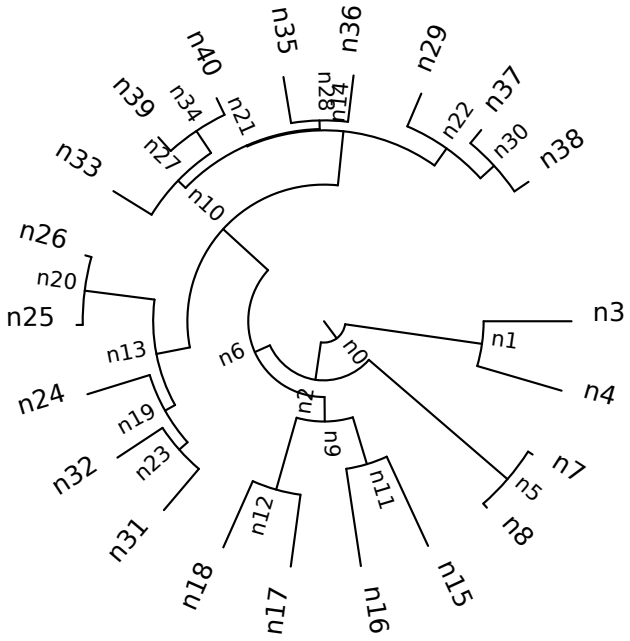


Now it looks as though only the conodonts are still alive, while the gnathostomes and urochordates each had brief flashes of existence 200 and 730 million years ago, respectively. Don't show this to a palaeontologist.

2.17 Generating Random Trees

`nw_gen` generates clock-like random trees, with exponentially distributed branch lengths. Nodes are sequentially labeled.

```
$ nw_gen -s 0.123 | nw_display -sSr -
```



Here I pass option `-s`, whose argument is the pseudo-random number generator's seed, so that I get the same tree every time I produce this document. Normally, you will not use it if you want a different tree every time. Other options are `-d`, which specifies the tree's depth, and `-l`, which sets the average branch length.

I use random trees to test the other applications, and also as a kind of null model to test to what extent a real tree departs from the null hypothesis.

2.18 Stream editing

2.18.1 Background

The programs we have seen so far are all specialized for a given task, hopefully one of the more frequent ones. It is, of course, impossible to foresee every way in which one may need to process a tree, and with this we hit a limit to specialization. In some cases, a more general-purpose program may offer a solution.

As an analogy, consider the task of finding lines in a text file that match a given pattern. This can be done, for example, in the following ways, from the general to the specific:

- a Perl program
- a `sed` one-liner
- a `grep` command

Perl is a general-purpose language, it just happens to be rather good at processing text. Sed is specialized for editing text streams, and `grep` is designed for precisely the line-finding task in question[11]. We should expect `grep` to be the most efficient, but we should not expect it to be able to perform any significantly different task. By contrast, Perl may be (I haven't checked!) less efficient than `grep`, but it can handle pretty much

[11]The name "grep" comes from the `sed` expression `g/re/p`, where "re" stands for "regular expression"

any task. Sed will lie in between. The programs we have seen so far are `grep`-like: they are specialized for one task (and hopefully, they are efficient).

The programs described in this section are more `sed`-like: they are less specialized, usually less efficient, but more flexible than the ones shown up to now. They were in fact inspired by `sed` and `awk`, which perform an action on the parts of input (usually lines) that meet some condition. Rather than lines in a file, the programs presented here work with nodes in a tree: each node is visited in turn, and if it meets a user-specified condition, a user-specified action is performed. In other words, they are node-oriented stream editors for trees.

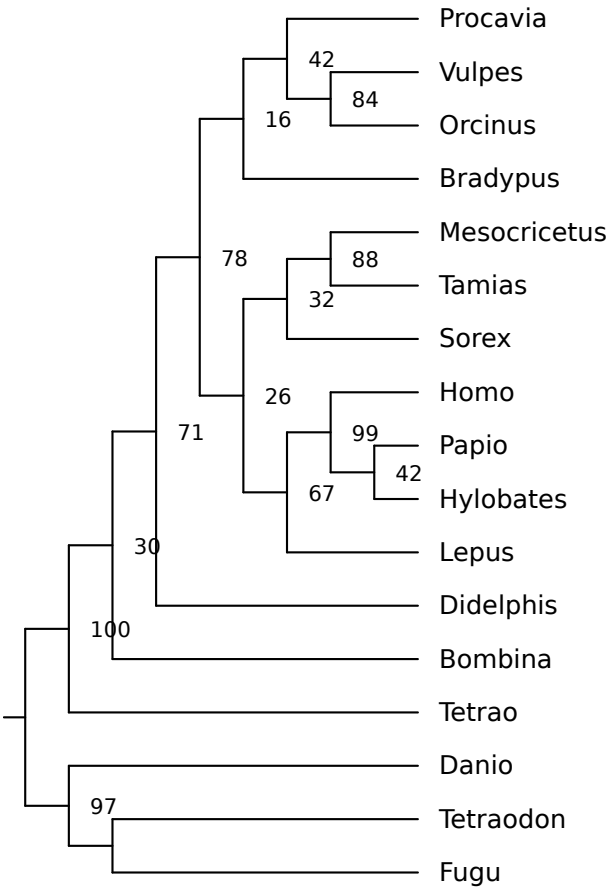
As a word of warning, I should say that these programs are among the more experimental in the Newick Utilities package. This is why there are two programs (at least one more is on the way) that do basically the same thing, although differently and with different capabilities: `nw_ed` is the simplest (and first), it is written entirely in C and it is fairly limited. `nw_sched` was developed to address `nw_ed`'s limitations: by embedding a Scheme (`http://www.r6rs.org`) interpreter (GNU Guile, `http://www.gnu.org/software/guile/`), its flexibility is, for practical purposes, limitless. Of course, this comes at the price of linking to an external library, which may not be available. Therefore `nw_ed`, for all its limitations, will stay in the package as it does not depend on Guile. Finally, I understand that Scheme is not the only solution as an embedded language, and that many people (myself included) find learning it a challenge. Therefore, my intention is to try the same approach with Lua (`http://www.lua.org`), which is designed as an embeddable language, is even smaller than Guile, and by many accounts easier to learn.

2.18.2 `nw_ed`

`nw_ed` iterates over the nodes in a specific order, and for each node it evaluates a logical expression provided by the user. If the expression is true, `nw_ed` performs a user-specified action. By default, the (possibly modified) tree is printed at the end of the run.

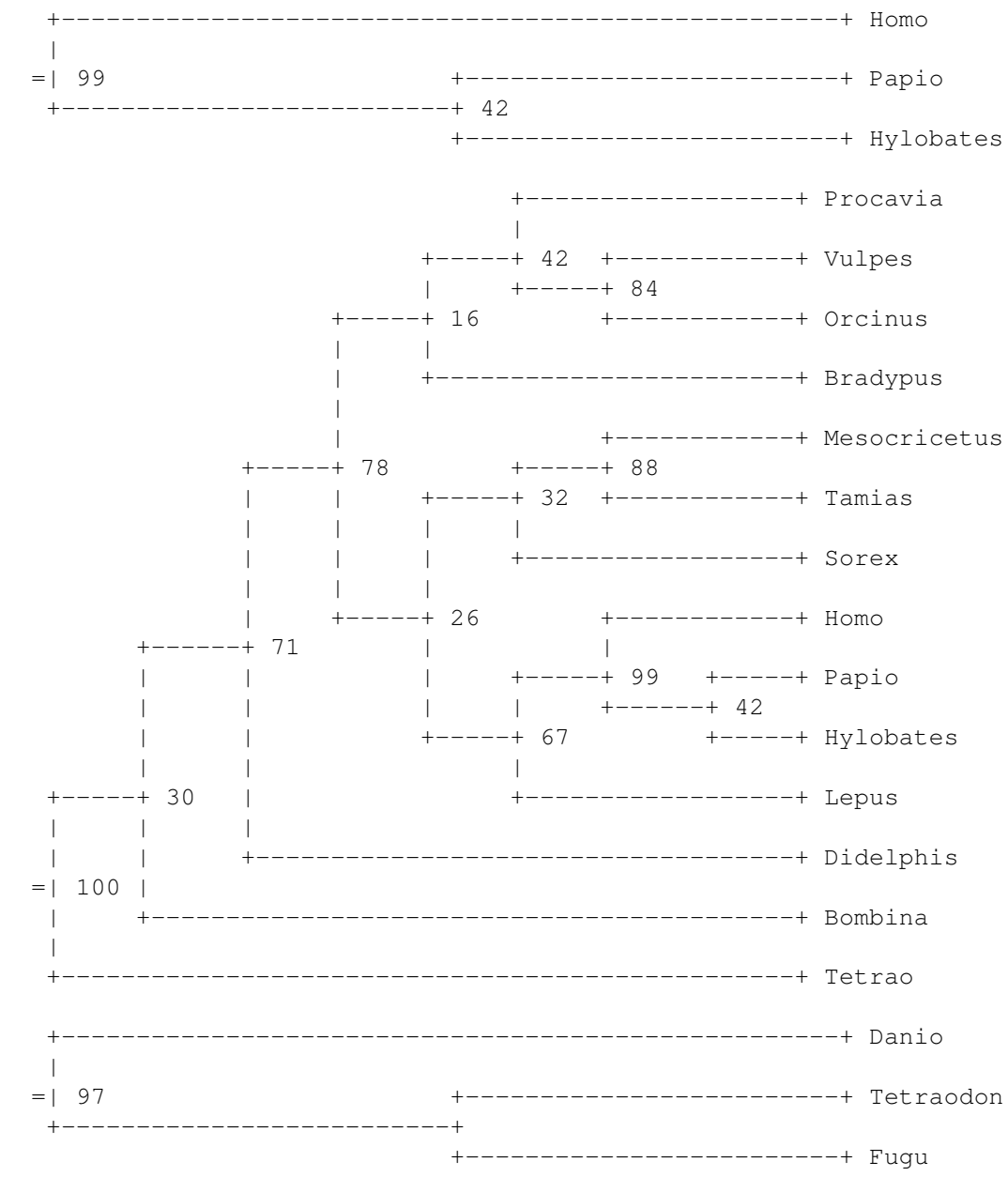
Let's look at an example before we jump into the details. Here is a tree of vertebrate genera, showing support values:

```
$ nw_display -s -v 25 vrt2cg.nw
```



Let's extract all well-supported clades, using a support value of 95% or more as the criterion for being well-supported:

```
$ nw_ed -n vrt2cg.nw 'b >= 95' s | nw_display -w 65 -
```



This instructs `nw_ed` to iterate over the nodes, in Newick order, and to print the subtree (action `s`) for all nodes that match the expression `b >= 95`, which means "interpret the node label as a (bootstrap) support value, and evaluate to true if that value is greater than 95". As we can see, `nw_ed` reports the three well-supported clades (primates, tetrapods, and ray-finned fishes), in Newick order. We also remark that one of the clades (primates) is contained in another (tetrapods). Finally, option `-n` suppresses the printing of the whole tree at the end of the run, which isn't useful here.

The two parameters of `nw_ed` (besides the input file) are an *address* and an *action*. Addresses specify sets of nodes, and actions are performed on them.

Addresses

Currently, addresses are logical expressions involving node properties, and the action is performed on the nodes for which the expression is true. They are composed of numbers, logical operators, and node functions.

The functions have one-letter names, to keep expressions short (after all, they are passed on the command line). There are two types, numeric and Boolean.

name	type	meaning
`a`	numeric	number of ancestors of node
`b`	numeric	node's support value (or zero)
`c`	numeric	node's number of children (direct)
`D`	numeric	node's number of descendants (includes children)
`d`	numeric	node's depth (distance to root)
`i`	Boolean	true iff node is strictly internal (i.e., not root!)
`l` (ell)	Boolean	true iff node is a leaf
`r`	Boolean	true iff node is the root

The logical and relational operators work as expected, here is the list, in order of precedence, from tightest to loosest-binding. Anyway, you can use parentheses to override precedence, so don't worry.

symbol	operator
`!`	logical negation
`==`	equality
`!=`	inequality
`<`	greater than
`>`	lesser than
`>=`	greater than or equal to
`<=`	lesser than or equal to
`&`	logical and
`|`	logical or

Here are a few examples:

expression	selects:
`l`	all leaves
`l & a <= 3`	leaves with 3 ancestors or less
`i & (b >= 95)`	internal nodes with support greater than 95%
`i & (b < 50)`	unsupported nodes (less than 50%)
`!r`	all nodes except the root
`c > 2`	multifurcating nodes

Actions

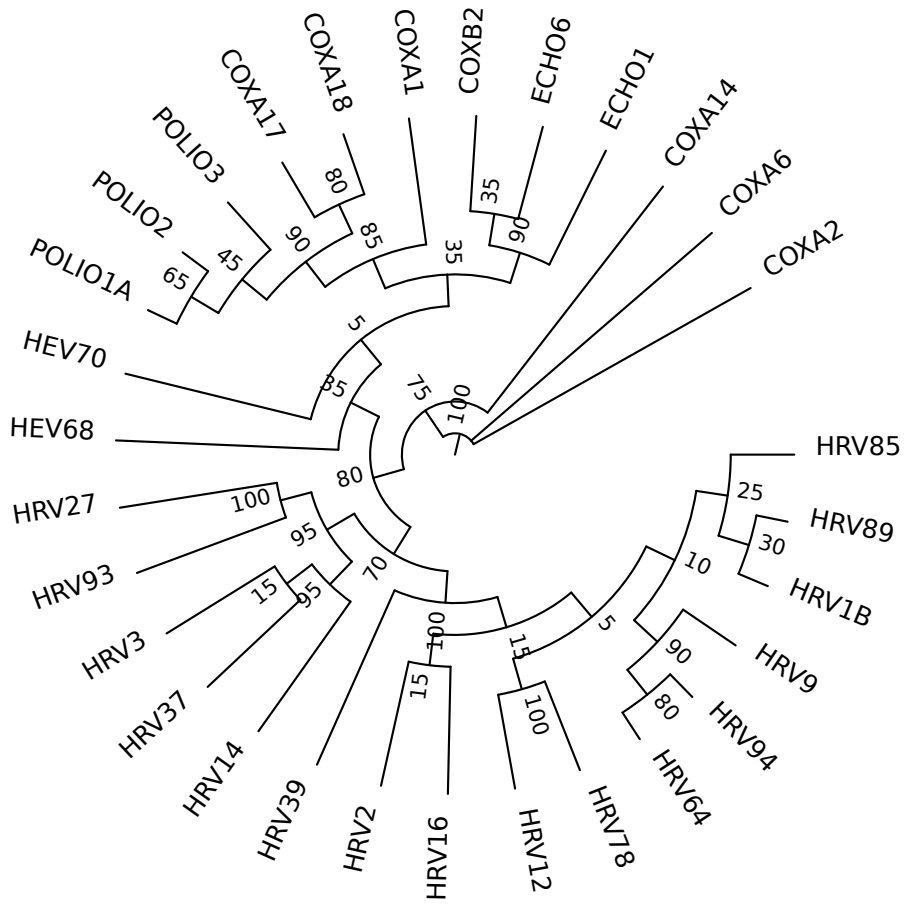
The actions are also coded by a single letter, for the same reason. For now, the following are implemented:

code	effect	modifies tree?
`d`	delete the node (and all descendants)	yes
`l`	print the node's label	no
`o`	splice out the node	yes
`s`	print the subtree rooted at the node	no

nw_ed is somewhat experimental; it is also the only program that is not deliberately "orthogonal" to the rest, that is, it can emulate some of the functionality of other utilities.

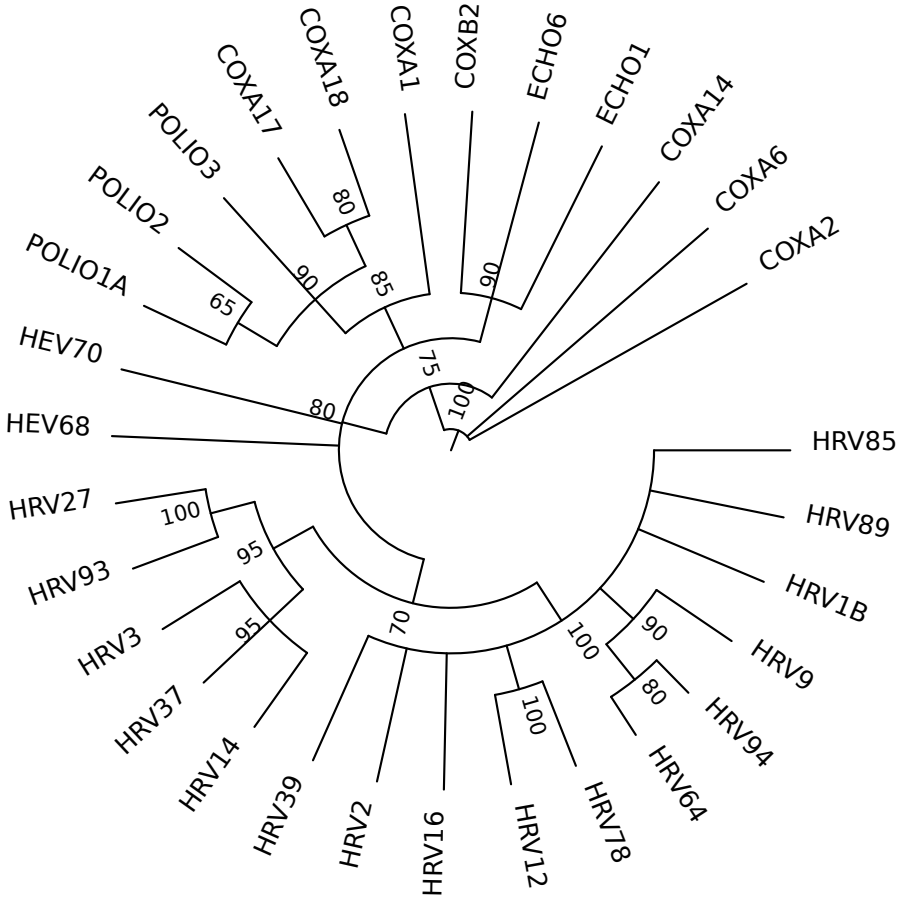
2.18.3 Opening Poorly-supported Nodes

When a node has low support, it may be better to splice it out from the tree, reflecting uncertainty about the true topology. Consider the following tree, HRV_cg.nw:



The inner node labels represent bootstrap support, as a percentage of replicates. As we can see, some nodes are much better supported than others. For example, the (COXB2,ECHO6) node (top of the figure) has a support of only 35%, and in the lower right of the graph there are many nodes with even poorer support. Let's "open" the nodes with less than 50% support. This means that those nodes will be spliced out, and their children attached to their "grandparent":

```
$ nw_ed HRV_cg.nw 'i & b < 50' o | nw_display -sr -w 450 -
```

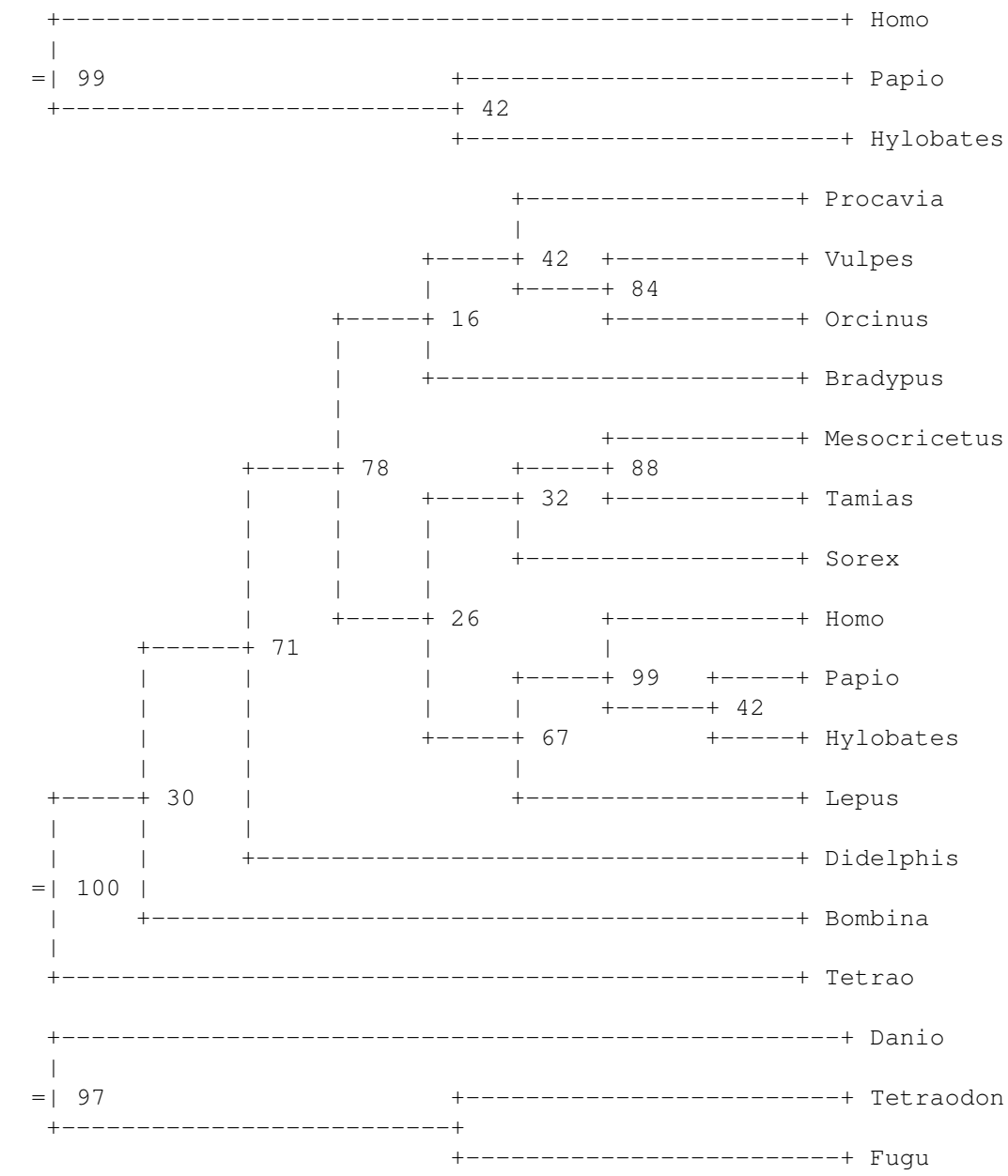


Now `COXB2` and `ECHO6` are siblings of `ECHO1`, forming a node with 90% support. What this means is that the original tree strongly supports that these three species form a clade, but is much less clear about the relationships *within* the clade. Opening the nodes makes this fact clear by creating multifurcations. Likewise, the lower right of the figure is now occupied by a highly multifurcating (8 children) but perfectly supported (100%) node, none of whose descendants has less than 80% support.

2.18.4 `nw_sched`

As mentioned above, `nw_sched` works like `nw_ed`, but embeds a Scheme interpreter so that the address (also called the *selector*) and action are passed as a Scheme expression. The Scheme language (see link above) has a simple syntax, but it can be slightly surprising at first. To understand the following examples, you just need to know that operators *precede* their arguments, as do function names, so that the sum of 2 and 2 is written `(+ 2 2)`, the sine of x is `(sin x)`, `(< 3 2)` is false, etc.
As a first example, let's again extract all well-supported clades from the tree of vertebrate genera, as we did with `nw_ed`.

```
$ nw_sched -n vrt2cg.nw "((& (def? 'b) (>= b 95)) (s))" \
| nw_display -w 65 -
```



The expression `((& (def? 'b) (>= 95 b)) (s))` parses as follows:

- the first element (or `car`, in Scheme parlance), `(& (def? 'b) (>= 95 b))`, is the selector. It is a Boolean expression, namely a conjunction (`&`)[12] of the expressions `(def? 'b)` and `(>= 95 b)`. The former checks that variable `b` (bootstrap support) is defined[13], and the latter is true iff `b` is not smaller than 95.
- the second element (`cadr` in Scheme jargon), `(s)`, is the action – in this case, a

[12] `&` is a short name for the Scheme form `and`, which is defined by `nw_sched` to allow for shorter expressions on the command line.

[13] In `nw_ed`, `b` was zero if the support was not defined. `nw_sched` distinguishes between undefined and zero, which is why one has to check that `b` is defined before using it. `def?` is just a shorter name for `defined?`.

call to function `s`, which has the same meaning as action `s` in `nw_ed`, namely to print out the subclade rooted at the current node.

Selectors

Like `nw_ed` addresses, `nw_sched` selectors are Boolean expressions normally involving node properties which are available as predefined variables. As the program "visits" each node in turn, the variables are set to reflect the current node's properties. As in `nw_ed`, the variables have short names, to keep expressions concise. The predefined variables are shown in the table below.

name	type	meaning
`a`	integer	number of ancestors
`b`	rational	support value
`c`	integer	number of children (direct descendants)
`D`	integer	total number of descendants (includes children)
`d`	numeric	depth (distance to root)
`i`	Boolean	true iff node is strictly internal (i.e., not root!)
`lbl`	string	label
`l` (ell)	Boolean	true iff node is a leaf
`L`	rational	parent edge length
`r`	Boolean	true iff node is the root

Variables `b` and `lbl` are both derived from the label, but `b` is interpreted as a number, and is undefined if the conversion to a number fails, or if the node is a leaf. Edge length and depth (`L` and `d`) are undefined (not zero!) if not specified in the Newick tree, as in cladograms.

Whereas `nw_ed` defines logical and relational operators, `nw_sched` just uses those of the Scheme language. It just defines a few shorter names to help keep command lines compact:

Scheme	`nw_sched` short form	meaning
`not`	`!`	logical negation
`and`	`&`	logical and
`or`	`|`	logical or
`defined?`	`def?`	checks if arg is defined

Here are a the same examples as above, but for `nw_sched`:

expression	selects:
`l` (lowercase ell)	all leaves
`(& l (<= a 3))`	leaves with 3 ancestors or less
`(& i (def? 'b) (>= b 95))`	internal nodes with support greater than 95%
`(& i (def? 'b) (b < 50)`	unsupported nodes (less than 50%)
`(! r)`	all nodes except the root
`(> c 2)`	multifurcating nodes

When it is clear that all inner nodes will have a defined support value, one can leave out the `(def? 'b)` clause.

Actions

Actions are arbitrary Scheme expressions, so they are much more flexible than the fixed actions defined by `nw_ed`. `nw_sched` defines most of them, as well as a few new ones, as Scheme functions[14]:

[14]Note that you must use Scheme's function call syntax to call the function, *i.e.*, `(function [args...])`.

code	effect	modifies tree?
`L! <len>`	sets the node's parent-edge length to `len`	yes
`lbl! <lbl>`	sets the node's label to `lbl`	yes
`o`	splice out the node	yes
`p <arg>`	print `arg`, then a newline	no
`s`	print the subtree rooted at the node	no
`u`	delete ("unlink") the node (and all descendants)	yes

The `l` action of `nw_ed`, which prints the current node's label, can be achieved in `nw_sched` with the more general `p` function: `(p lbl)`.

The `L!` function sets the current node's parent-edge length. It accepts a string or a number. If the argument is a string, it attempts to convert it to a number. If this fails, the edge length is undefined. The `lbl!` function sets the current node's label. Its argument is a string.

The delete action was renamed to `u` because the `d` symbol stands for the node's depth.

Examples

Formatting Lengths
Some phylogeny programs return Newick trees with an unrealistic number of decimal places. For example, the `HRV.nw` tree has six:

```
$ nw_indent HRV.nw | tail
            ):0.936634
          ):0.770246
        ):0.051896
      ):0.438878
    ):1.235120,
    COXA14_1:0.121281
  ):0.544944,
  COXA6_1:0.675458,
  COXA2_1:0.557975
);
```

Here I use `nw_indent` to show each node on a line for clarity, and show only the last ten.[15] To format[16] the lengths to two decimal places, do the following:

```
$ nw_sched HRV.nw "((def? 'L) (L! (format #f \"~,2f\" L)))" \
| nw_indent - | tail
            ):0.94
          ):0.77
        ):0.05
      ):0.44
    ):1.24,
    COXA14_1:0.12
  ):0.54,
  COXA6_1:0.68,
  COXA2_1:0.56
);
```

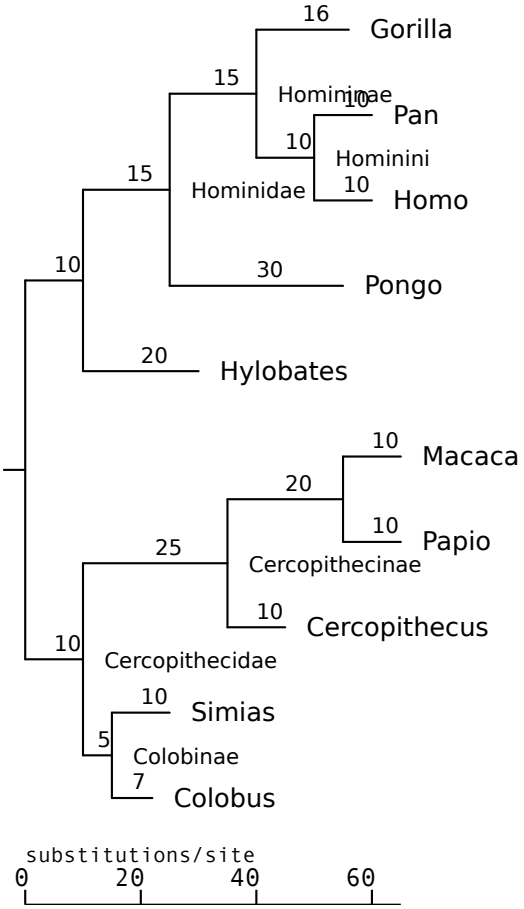
Multiplying lengths by a constant

[15] the *first* ten lines contain only opening parentheses.
[16] `nw_sched` automatically loads the `format` module so that the full-fledged `format` function is available.

It may be necessary to have two trees which only differ by a constant multiple of the branch lengths. This can be used, for example, to test competing hypotheses about evolution rates.

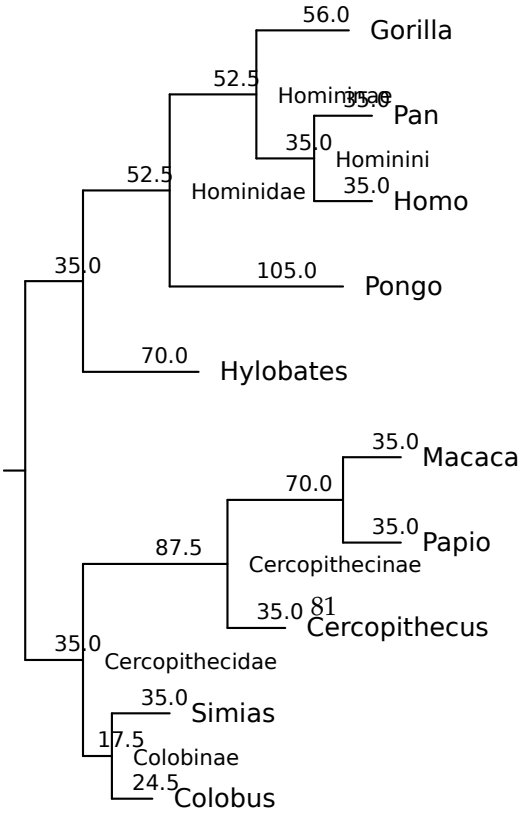
Here is our good friend the Catarrhinine tree again:

```
$ nw_display -s catarrhini
```



To multiply all its branch lengths by (say) 3.5, do the following:

```
$ nw_sched catarrhini "((! r) (L! (* 3.5 L)))" | nw_display -s -
```



`nw_sched` can emulate other programs in the package, when these iterate on every node and perform some action. There is no real reason to use `nw_sched` rather than the original, since `nw_sched` will be slower (after all, it has to start the Scheme interpreter, parse the Scheme expressions, etc.). But it can serve as illustration.

program	`nw_sched` equivalent
`nw_labels`	`'((! (string-null? lbl)) (p lbl))'`
`nw_labels -I`	`'((! (| i (string-null? lbl))) (p lbl))'`
`nw_topology`	`'(#t (L! ""))'`
`nw_topology -I`	`'(#t (if l (L! "") (begin (L! "") (lbl! ""))))'`

The `string-null?` predicate in the `nw_labels` replacements is checked because the original `nw_labels` does not print empty labels. The `begin` form in the `nw_topology` replacement is used to perform two actions: setting the length and setting the label. Note also that the check for node type is done in the action rather than the selector. The `if` on the selector is implicit. Future versions will allow more than one (selector, action) pair (just as Awk does) and allow the `nw_topology -I` equivalent to be written `'((#t (L! "")) ((! l) (lbl! "")))` – set length to undefined for every node, and set label to empty for all non-leaf nodes. I also intend to put an implicit `begin` around the action.

Chapter 3

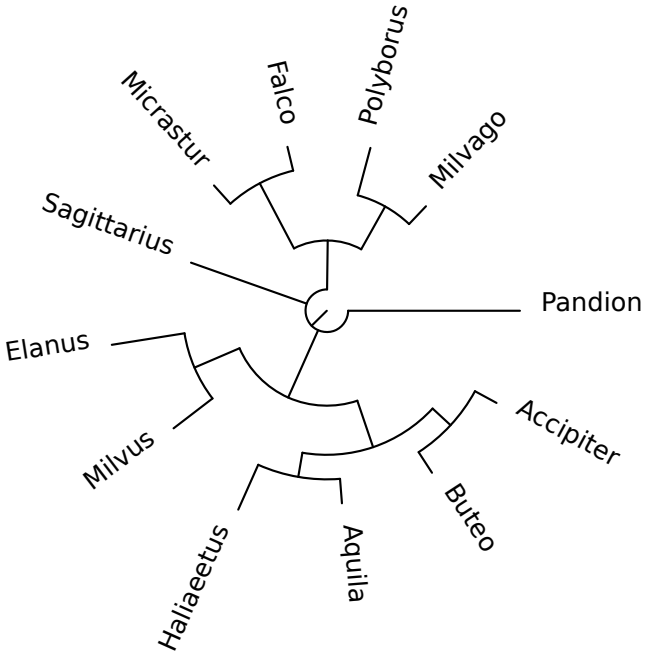
Advanced Tasks

The tasks presented in this chapter are more complex than that of chapter 2, and generally involve many Newick Utilities as well as other programs.

3.1 Checking Consistency with other Data

3.1.1 By condensing

One can check the consistency of a tree with respect to additional information by renaming and condensing. For example, I have the following tree of Falconiformes (diurnal raptors: eagles, falcons, etc):



Now I also have the following information about the family to which each genus belongs:

Genus	**Family**
Accipiter	Accipitridae
Aquila	Accipitridae
Buteo	Accipitridae
Elanus	Accipitridae
Falco	Falconidae
Haliaeetus	Accipitridae
Micrastur	Falconidae
Milvago	Falconidae
Milvus	Accipitridae
Pandion	Pandionidae
Polyborus	Falconidae
Sagittarius	Sagittariidae

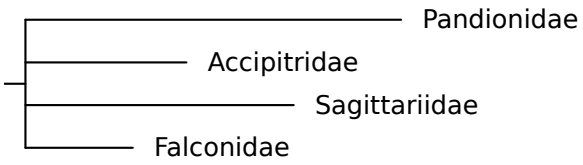
Let's see if the tree is consistent with this information. If it is, all families should form clades. To check this, I will rename each leaf by replacing the genus name by the family name, then condense the tree. If the original tree is consistent, the final tree should have one leaf per family.

First, I create a renaming map (see 2.9) based on the above information (here are the first three lines):

```
$ head -3 falc_map

Accipiter       Accipitridae
Buteo   Accipitridae
Aquila  Accipitridae
```

Then I use it to rename, and then I condense the tree:

```
$ nw_rename falconiformes falc_map \
| nw_condense - \
| nw_display -s -S -v 20 -b opacity:0 -
```



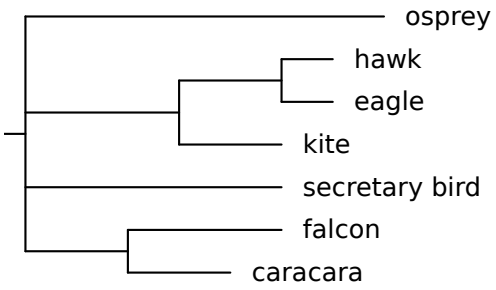
As we can see, there is one leaf per family, so the above information is consistent with the tree.

Let's see if common English names are also consistent with the tree. Here is one possible table of vernacular names of the raptor genera:

Genus	English name
Accipiter	hawk (sparrowhawk, goshawk, etc)
Aquila	eagle
Buteo	hawk
Elanus	kite
Falco	falcon
Haliaeetus	eagle (sea eagle)
Micrastur	falcon (forest falcon)
Milvago	caracara
Milvus	kite
Pandion	osprey
Polyborus	caracara
Sagittarius	secretary bird

And here is the corresponding tree:

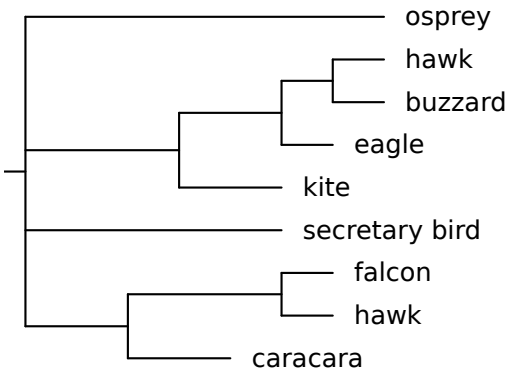
```
$ nw_rename falconiformes falconiformes_vern1.map \
| nw_condense - \
| nw_display -s -S -v 20 -b opacity:0 -
```



So the above common names are consistent with the tree. However, some species have many common names. For example, the *Buteo* hawks are often called "buzzards" (in Europe), and two species of falcons have been called "hawks" (in North America): the peregrine falcon (*Falco peregrinus*) was called the "duck hawk", and the American kestrel (*Falco sparverius*) was called the "sparrow hawk".[1] If we map these common names to the tree and condense, we get this:

```
$ nw_rename falconiformes falconiformes_vern2.map \
| nw_condense - \
| nw_display -s -S -v 20 -b opacity:0 -
```

[1] This is confusing because there are true hawks called "sparrow hawks", *e.g.* the Eurasian sparrow hawk *Accipiter nisus*. To add to the confusion, the specific name *sparverius* looks like the English word "sparrow", and also resembles the common name of *Accipiter nisus* in many other languages: *épervier* (fr), *Sperber* (de), *sparviere* (it). Oh well. Let's not drop scientific names just yet!



Distinguishing buzzards from other hawks fits well with the tree. On the other hand, calling a falcon a hawk does not, hence the name "hawk" appears in two different places.

3.2 Bootscanning

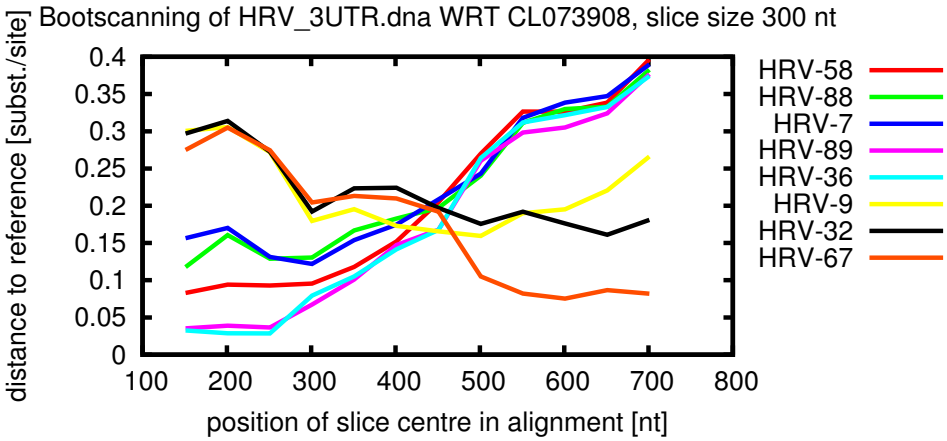
Bootscanning is a technique for finding recombination breakpoints in a sequence. It involves aligning the sequence of interest (called *query* or *reference*) with related sequences (including the putative recombinant's parents) and computing phylogenies locally over the alignment. Recombination is expected to cause changes in topology. The tasks involved are shown below:

1. align the sequences → multiple alignment
2. slice the multiple alignment → slices
3. build a tree for each slice → trees
4. extract distance from query to other sequences (each tree) → tabular data
5. plot data → graphics

The distribution contains a script, `src/bootscan.sh`, that performs the whole process. Here is an example run:

```
$ bootscan.sh HRV_3UTR.dna HRV-93 CL073908
```

where `HRV_3UTR.dna` is a FastA file of (unaligned) sequences, `HRV-93` is the outgroup, and `CL073908` is the query. Here is the result:

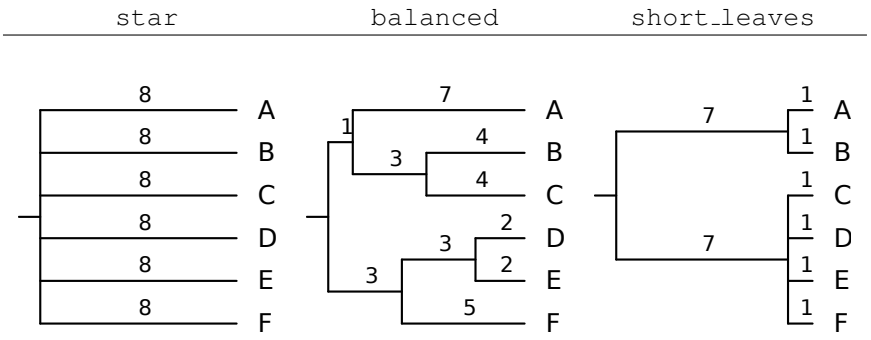


until position 450 or so, the query sequence's nearest relatives (in terms of substitutions/site) are `HRV-36` and `HRV-89`. After that point, it is `HRV-67`. This suggests that there is a recombination breakpoint near position 450.

The script uses `nw_reroot` to reroot the trees on the outgroup, `nw_clade` and `nw_labels` to get the labels of the ingroup, `nw_distance` to extract the distance between the query and the other sequences, as well as the usual `sed`, `grep`, etc. The plot is done with gnuplot.

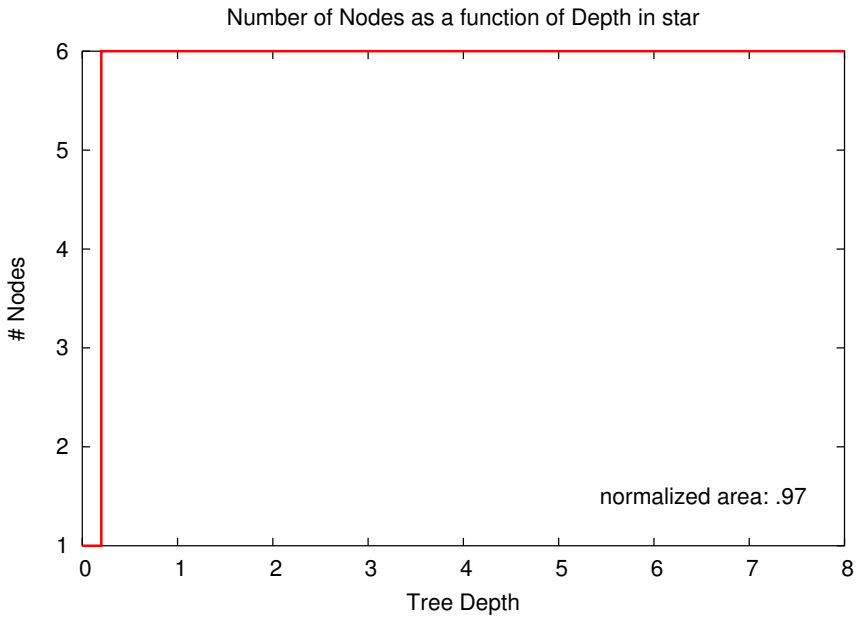
3.3 Number of nodes vs. Tree Depth

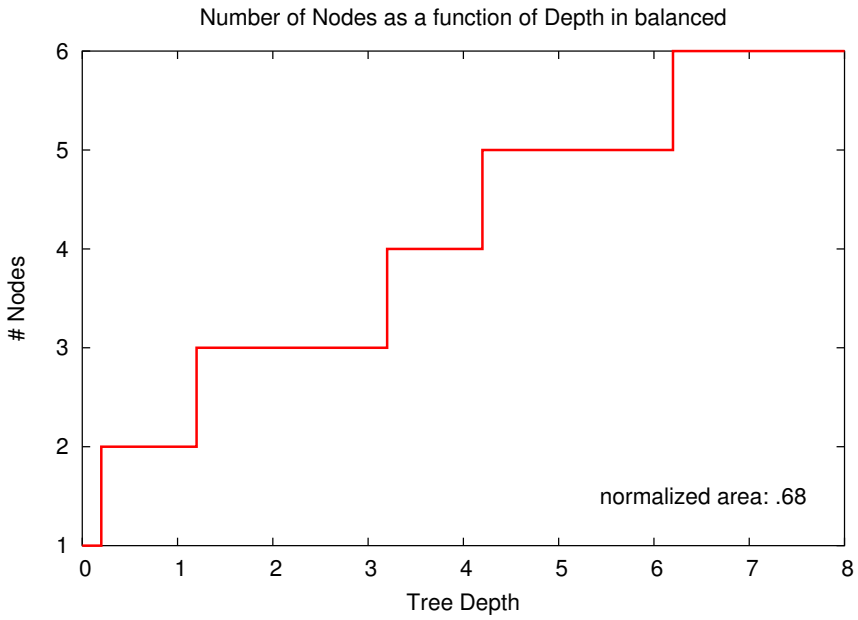
A simple measure of a tree's shape can be obtained by computing the number of nodes as a function of depth. Consider the following trees:

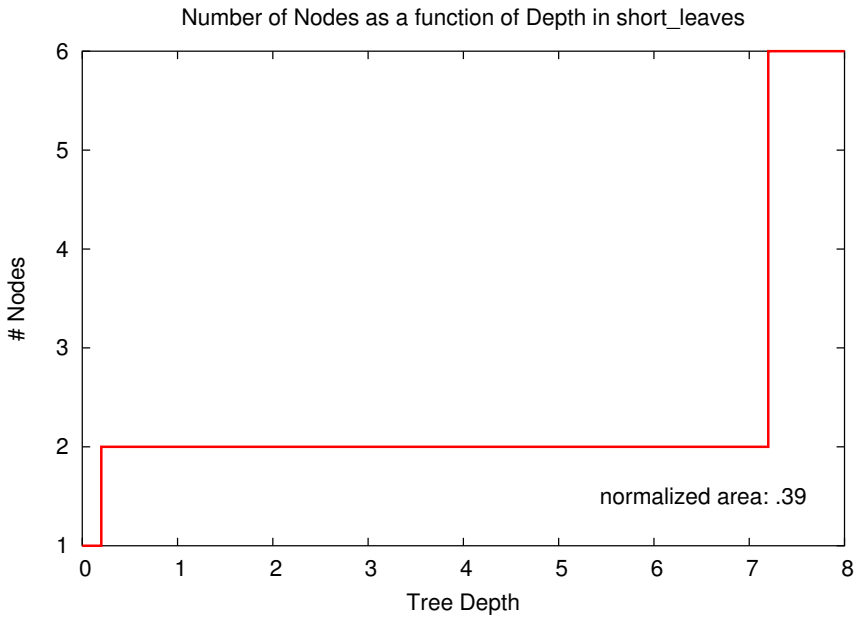


they have the same depth and the same number of leaves. But their shapes are very different, and they tell different biological stories. If we assume that they are clock-like (i.e., that the mutation rate is constant over the whole tree), `star` shows an early radiation, `short_leaves` shows two stable lineages ending in recent branching, while `balanced` shows branching spread over time.

The nodes-vs-depth graphs for these trees are as follows:


Number of Nodes as a function of Depth in star
Nodes
Tree Depth
normalized area: .97
1
2
3
4
5
6
0
1
2
3
4
5
6
7
8


Number of Nodes as a function of Depth in balanced
Nodes
Tree Depth
normalized area: .68
1
2
3
4
5
6
0
1
2
3
4
5
6
7
8



The graphs show the (normalized) area under the curve: it is close to 1 for star-like trees, close to 0 for trees with very short leaves, and intermediary for more balanced trees.

The images were made with the `nodes_vs_clades.sh` script (in directory `src`), in the following way:

```
$ nodes_vs_clades.sh star 40
```

where 40 is just the sampling density (how many points to take on the x axis). The script uses `nw_distance` to get the tree's depth, `nw_ed` to sample the number of nodes at a given depth, and `nw_indent` to count the leaves, plus the usual `awk` and friends. The plot is done with gnuplot.

Chapter 4

Python Bindings

Although the Newick Utilities are primarily designed for shell use, it is also possible to use their functions from Python programs: all the core functionality of the utilities is bundled in a C library, `libnw`, which can be accessed through Python's `ctypes` module. The distribution contains a file, `newick_utils.py`, that provides the Python to C mappings; it also builds an object-oriented interface over it.

Let's say we want to add a utility that prints simple statistics about trees, like the number of nodes, the depth, whether it is a cladogram or a phylogram, etc (in other words, a Python version of `nw_stats`). We will call it `nw_info.py`, and we'll pass it a Newick file on standard input, so the usage will be something like:

```
$ nw_info.py < data/catarrhini
```

The overall structure of this program is simple: iteratively read all the input trees, and do something with each of them:

```
1  from newick_utils import *
2
3  for tree in Tree.parse_newick_input():
4      pass # process tree here!
```

Line 1 imports definitions from the `newick_utils.py` module. Line 3 is the main loop: the `Tree.parse_newick_input` reads standard input and yields an instance of class `Tree` for each Newick string. We can now work with it, using methods of class `Tree` or adding our own:

```
1  #!/usr/bin/env python
2
3  from newick_utils import *
4
5  def count_polytomies(tree):
6          count = 0
7          for node in tree.get_nodes():
8                  if node.children_count() > 2:
9                          count += 1
10         return count
11
12 for tree in Tree.parse_newick_input():
13         type = tree.get_type()
14         if type == 'Phylogram':
```

```
15                  # also sets nodes' depths
16                  depth =  tree.get_depth()
17          else:
18                  depth = None
19          print 'Type:', type
20          print '#Nodes:', len(list(tree.get_nodes()))
21          print '  #leaves:', tree.get_leaf_count()
22          print '#Polytomies:', count_polytomies(tree)
23          print "Depth:", depth
```

When we run the program, we get:

```
$ nw_info.py < catarrhini

Type: Phylogram
#Nodes: 19
  #leaves: 10
#Polytomies: 0
Depth: 65.0
```

As you can see, most of the work is done by methods called on the `tree` object, such as `get_leaf_count` which (surprise!) returns the number of leaves of a tree. But since there is no method for counting polytomies, we added our own function, `count_polytomies`, which takes a `Tree` object as argument.

As another example, a simple implementation of `nw_reroot` is found in `src/nw_reroot.py`. It demonstrates two approaches: a heavily object-oriented one, in which the user mainly calls methods on Python objects, and a "thin" one, in which the calls are essentially to C functions through `libnw`. While not as fast as `nw_reroot`, its performance is still quite acceptable, especially in "thin" mode.

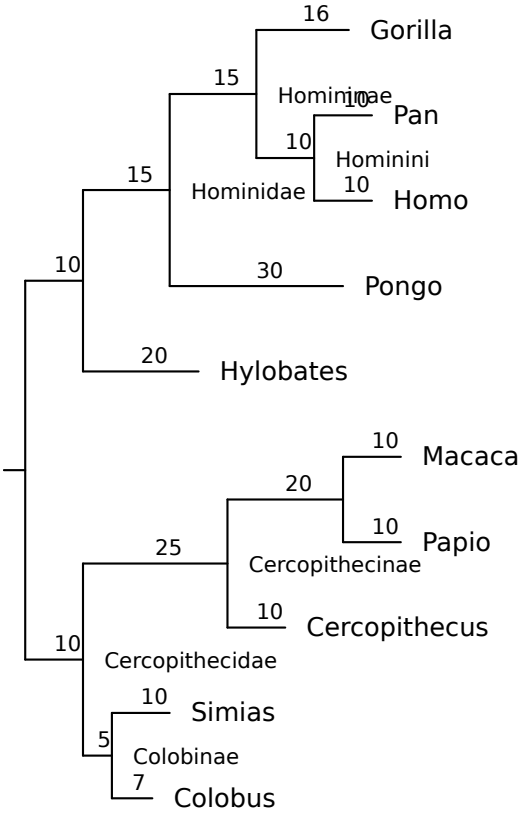
4.1 API Documentation

Detailed information about all classes and methods available for accessing the Newick Utilities library from Python is found in file `newick_utils.py`. Note that the library must be installed on your system, which means that you must compile from source.

Appendix A

Defining Clades by their Descendants

When you need to specify a clade using the Newick Utilities, you either give the label of the clade's root, or the labels of (some of) its descendants. Since inner nodes rarely have labels (or worse, have unusable labels like bootstrap support values), you will often need to specify clades by their descendants. Consider the following tree:



Suppose we want to specify the Hominoidea clade - the apes. It is the clade that contains `Homo`, `Pan` (chimps), `Gorilla`, `Pongo` (orangutan), and `Hylobates` (gibbons).

The clade is not labeled in this tree, but this list of labels defines it without ambiguity. In fact, we can define it unambiguously using just `Hylobates` and `Homo` - or `Hylobates` and any other label. The point is that *you never need more than two labels to unambiguously define a clade*.

You cannot choose any two nodes, however: the condition is that the last common ancestor of the two nodes be the root of the desired clade. For instance, if you used `Pongo` instead of `Hylobates`, you would define the Hominidae clade, leaving out the gibbons.

A.1 Why not just use node numbers?

Some applications attribute numbers to all inner nodes and allow users to specify clades by referring to this number. Such a scheme is not workable when one has many input trees, however, because there is no guarantee that the same clade (assuming it is present) will have the same number in different trees.

Appendix B

Newick order

There are many ways of visiting a tree. One can start from the root and proceed to the leaves, or the other way around. One can visit a node before its children (if any), or after them. Unless specified otherwise, the Newick Utilities process trees as they appear in the Newick data. That is, for tree `(A,(B,C)d)e;` the order will be A, B, C, d, e.

This means that a child always comes before its parent, and in particular, that the root comes last. This is known as reverse post-order traversal, but we'll just call it "Newick order".

Appendix C

Installing the Newick Utilities

C.1 For the impatient

```
./configure && make && sudo make install
```

C.2 From source

C.2.1 Prerequisites

I have tested the Newick Utilities on various distributions of Linux, as well as on Mac OS X and Cygwin[1]. On Linux, chances are you already have development tools preinstalled, but some distributions (*e.g.*, Ubuntu) do not install GCC, etc. by default. Check that you have GCC, Bison, and Flex. The same goes for Cygwin. On MacOS X, you need to install XCode (`http://developer.apple.com/tools/xcode`).

If you're using a non-stable version (such as the one from the Git repository, as opposed to a tarball), you will probably also need the GNU autotools, including Libtool.[2]

See C.4 for version numbers.

C.2.2 Optional Software

If libxml is present on your system, the Newick Utilities can use it to produce better SVG graphics. This is the default behaviour, but it is optional: if the library is not present (or not found), or if you specify not to use it (see below), the build process will work all the same, and most programs will not be affected in any way (currently this only affects ornaments to radial SVG trees).

Likewise, the system will build `nw_sched` unless Guile is not found or you explicitly disable it (see below).

Note that you need *both* the library *and* the headers for the build to succeed.

C.2.3 Build Procedure

The package uses the GNU autotools, like many other open source software packages. So all you need to do is the usual

[1] I use Linux as a main development platform. Although I try my best to get the package to compile on Macs and Cygwin, I don't always succeed.

[2] It may work without them, it's only that I don't explicitly *try* to make them work independently of the autotools – that's what stable releases are for, among other things.

```
$ tar xzf newick-utils-x.y.z.tar.gz
$ cd newick-utils-x.y.z
$ ./configure
$ make
$ make check
# make install
```

The `make check` is optional, but you should try it anyway. Note that the `nw_gen` test may fail - this is due to differences in pseudo-random number generators, as far as I can tell.

With non-stable releases, it may be necessary to reconfigure (this generally does not happen when using the tarball generated by the build system). So if you get weird error messages, try the following (you'll need the GNU autotools):

```
$ autoreconf -i
```

or even

```
$ autoreconf -fi
```

before launching `./configure` as above.

Variants

To prevent the use of libxml, pass `--without-libxml` to `./configure`. Likewise, to prevent the use of Guile, pass `--without-guile`.

If you have headers (such as Guile or LibXML's) in a non-standard location, pass that location via the `CPPFLAGS` environment variable when running `./configure`. Likewise, if you have libraries in a non-standard location, use `LDFLAGS`. The syntax is that of the `-I` and `-L` options to `gcc`, respectively. For example,

`LDFLAGS='-L/opt/lib' CPPFLAGS='-I/opt/include' ./configure`

would cause `/opt/lib` and `/opt/include` to be searched for libraries and headers, respectively. Note that there is no space between the `-I` and the `/opt/...`, etc.

C.3 As binaries

Since version 1.1, there are also binaries for some platforms. The name of the archive matches `newick-utils-<version>-<platform>-<enabled|disabled>-extra.tar.gz`. "enabled-extra" means that the binary depends on libXML and libguile and will expect to find them (as shared libs) on your system. "disabled-extra" means that the binary will not depend on those libraries, but of course the corresponding functionality (see C.2.2) won't be available. Simply do:

```
$ tar xzf newick-utils-<version,etc>.tar.gz
$ cd newick-utils-<version>
```

The binaries are in `src`. You can copy/move the binaries wherever it suits you.

C.4 Versions

Here are the versions I use (as reported by passing `--version` to the program listed in column 2):

tool	program	version	required for
GNU Autoconf	`autoconf`	2.67	non-stable source
GNU Automake	`automake`	1.11.1	non-stable source
GNU Bison	`bison`	2.4.1	source
Flex	`flex`	2.5.35	source
GNU Guile	`guile`	1.8.7	(optional)
GCC	`gcc`	4.4.5	source
GNU Libtool	`ltmain.sh`	2.2.6b	source
libxml	`xml2-config`	2.7.7	(optional)
GNU Make	`make`	3.81	source

It may also work with different versions. In case of problems, try to upgrade to the above.

Bibliography

[1] D. A. Baum, S. D. Smith, and S. S. Donovan. Evolution. The tree-thinking challenge. *Science*, 310:979–980, Nov 2005.

[2] T. R. Gregory. Understanding evolutionary trees. *Evolution: Education and Outreach*, 1(2):121–137, Apr 2008.

[3] T. Junier and E. M. Zdobnov. The Newick Utilities: High-throughput Phylogenetic tree Processing in the UNIX Shell. *Bioinformatics*, May 2010.