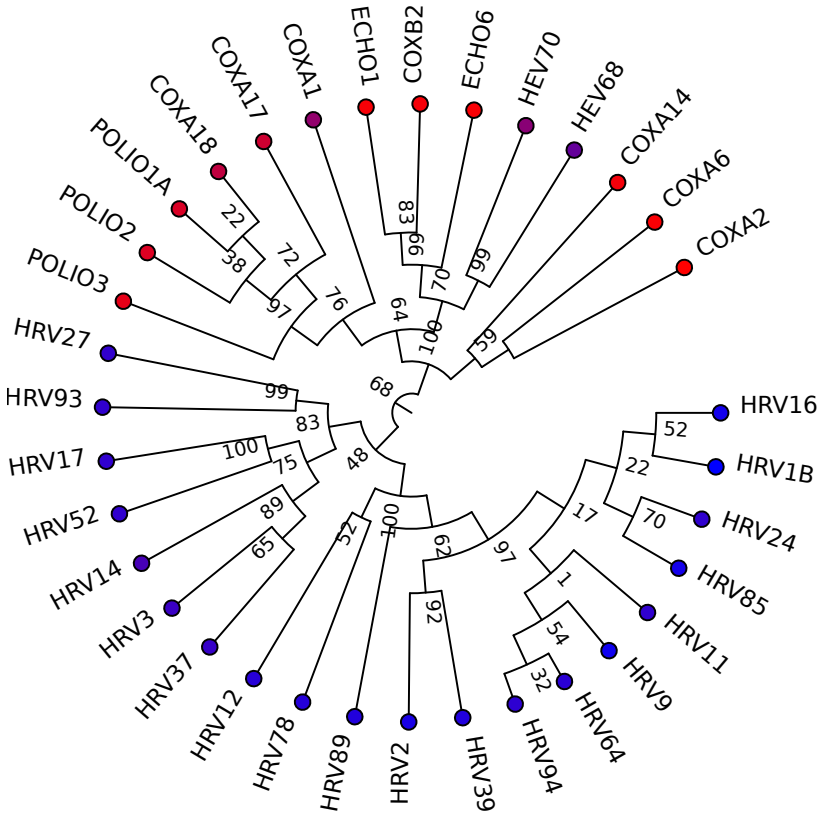
Newick Utilities Tutorial

Version 1.6.0 – June 12, 2012

Thomas Junier `thomas.junier@unine.ch`
Swiss Institute of Bioinformatics
and
University of Neuchâtel, Switzerland
***
but developed (2006-2011) at:
Computational Evolutionary Genomics Group
Department of Genetic Medicine and Development
University of Geneva, Switzerland

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



Contents

Chapter 1

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.

1.1 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.

1.1.1 As Text

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

```
$ nw_display catarrhini
```

```
nw_display: error while loading shared libraries: libnutils.so: cannot open shar
ed object file: No such file or directory
```

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

Fixing Width

`nw_display` will try to find the width of the terminal and use that many columns when drawing the tree. If the output is not a terminal (but, say, a file or pipe), or if

for some reason the width cannot be found or is not defined, then the program uses a default of 80 characters. You can use option `-w` to override the defaults.

```
$ nw_display -w 60 catarrhini

nw_display: error while loading shared libraries: libnutils.so: cannot open shar
ed object file: No such file or directory
```

Fixing Scale

Option `-w` can also be used to fix the scale (in columns/branch length units). To do so, just pass a negative number[1]. For example, to use half a column per unit length, do:

```
$ nw_display -w -0.5 catarrhini

nw_display: error while loading shared libraries: libnutils.so: cannot open shar
ed object file: No such file or directory
```

So a branch of length 10 (such as the parent branch of Homo[2]) occupies five characters (up to rounding error). And to use two columns per unit length, one would pass `-w -2`, etc.

If there is more than one tree in the input, they will all be drawn at the same scale, which makes them easier to compare visually.

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 **??** for an example.

Styles

Even lowly text graphics can have style! Option `-e` allows you to change the way text graphs look:

Argument to `-e`	Effect
`r`	(raw) use just `|`, `-`, and `+`, and `=` for the root.
`c`	(commas) corners are marked with `'` and `,`
`s`	(slashes) corners are marked with `/` and `\`
`v`	(VT100) use VT100 pseudo-graphical characters

Here are some examples:

[1] This way it is impossible to specify both width and scale, which would be absurd as one is always derived from the other.

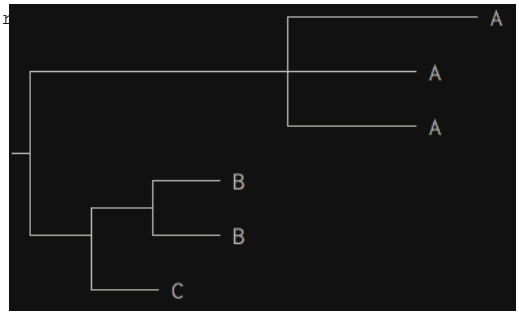
[2] To know the length of an edge, you can use `nw_distance -m p -n` – see **??**.

```
$ nw_display -S -w 40 -e r collapsable

nw_display: error while loading shared libraries: libnutils.so: cannot open shar
ed object file: No such file or directory
```

```
$ nw_display -S -w 40 -e c collapsable

nw_display: error while loading shared libraries: libnut
ils.so: cannot open shared object file: No such file or directory
```

```
$ nw_display -S -w 40 -e s collapsable

nw_display: error while loading shared libra
ed object file: No such file or directory
```

```
$ nw_display -S -w 40 -e v collapsable
```



This is a screenshot because I don't know how to show VT100 special characters in LaTeX :-)

By default, `nw_display` uses "slashes", unless overridden by environment variable `NW_DISPLAY_TEXT_STYLE`. This should have a value of `raw`, `commas`, `slashes`, or `VT100` (case is irrelevant, and actually only the first character is read, so you can abbreviate to `r`, `c`, `s`, or `v`).

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

nw_display: error while loading shared libraries: libnutils.so: cannot open shar
ed object file: No such file or directory
```

1.1.2 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[3]

To produce SVG, pass option `-s`:

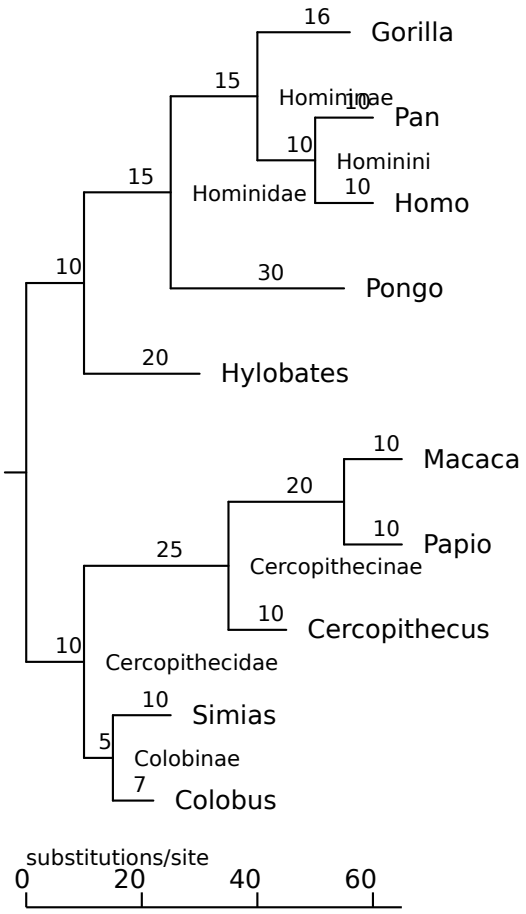
```
$ nw_display -s catarrhini > catarrhini.svg
```

[3] 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.

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 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 **??**).

The following PDF image was produced like this:

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



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

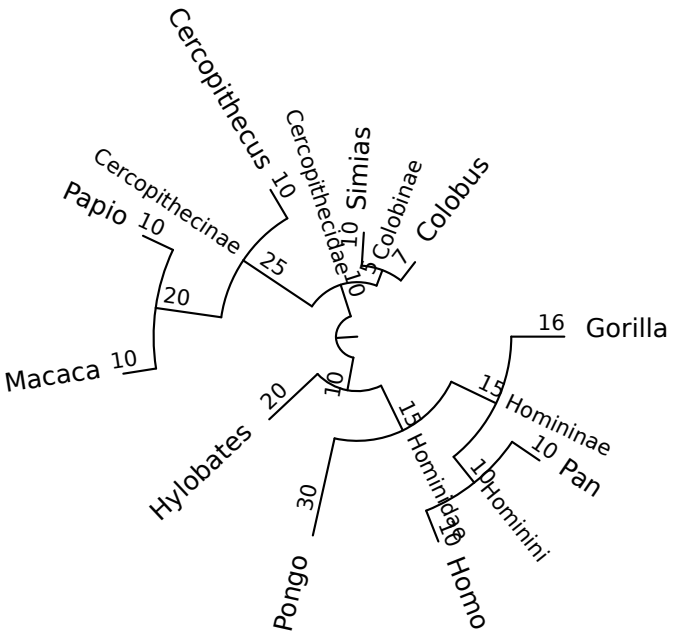
Many options for ASCII trees also work for SVG: `-S` suppresses the scale bar[4], and `-u` specifies its units; `-w` governs the tree's width (or fixes the scale if its argument is negative), except that for SVG the value is in pixels instead of columns; `-I` controls the placement of inner node labels.

[4]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.

Radial trees

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

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



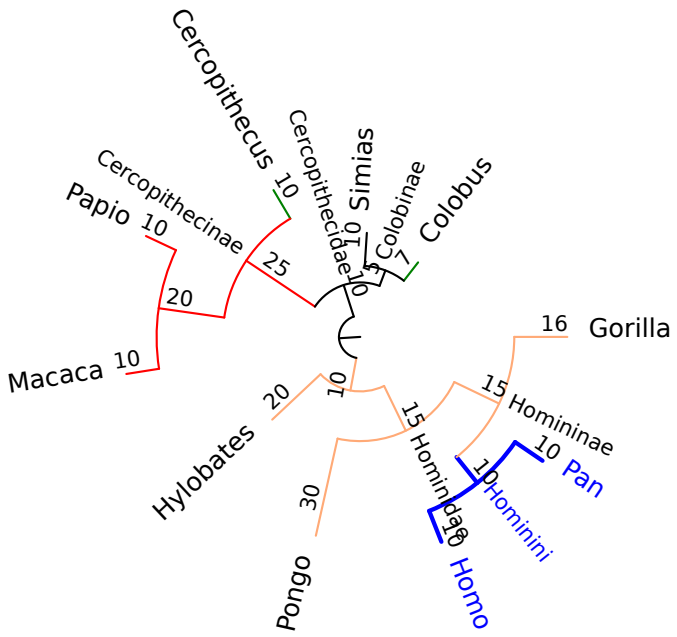
Using CSS

You can modify node style using Cascading Style Sheets (CSS, (`www.w3.org/Style/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. For example, 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
# Labels in color as well
fill:blue L Pan Homo Hominini
```

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`, `Individual`, or `Label` (which can be abbreviated to `C`, `I`, or `L`, respectively).
- The remaining element(s) are node labels and specify the nodes to which the style must be applied: if the second element is `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 is `Individual`, then the style is only applied to the nodes themselves. If the second element is `Label`, then the style is applied to the labels.

In our example, `css.map`:

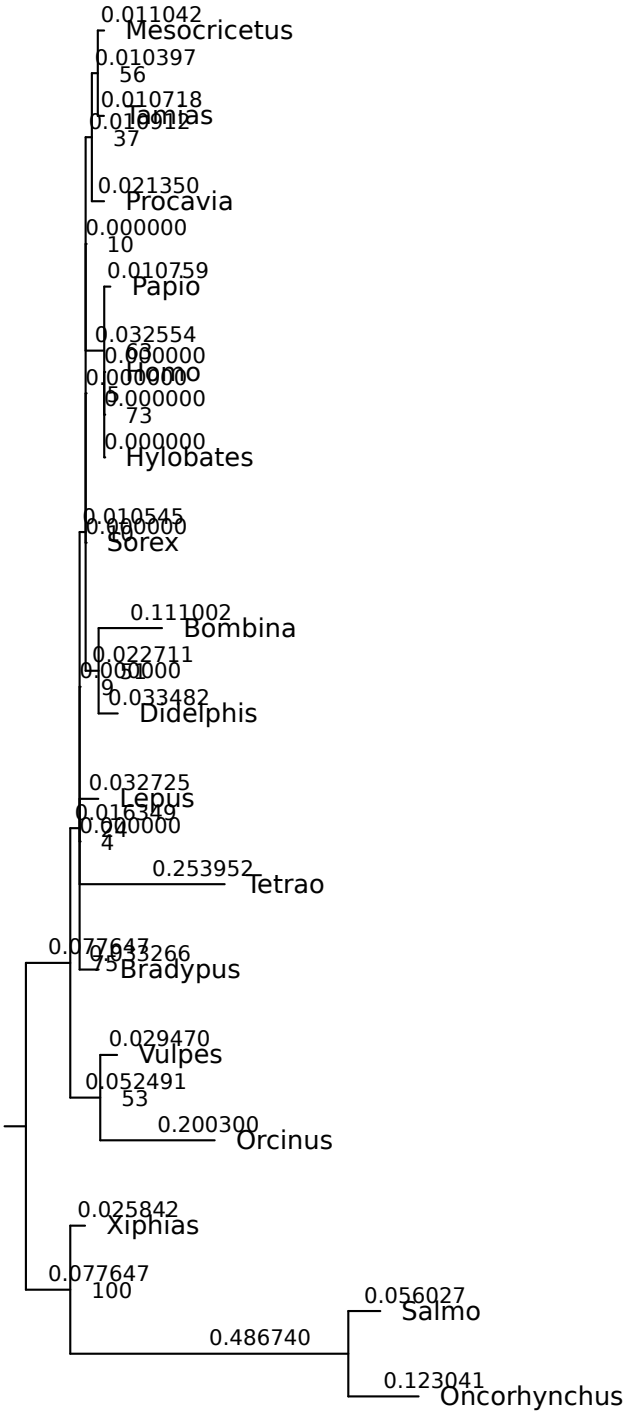
- line 2 states that the style `stroke:red` must be applied to the `Clade` defined by `Macaca` and `Cercopithecus`, which consists of these two nodes, their ancestor `Cercopithecinae`, and `Papio`.
- Line 4 prescribes that style `stroke:#fa7` (an SVG hexadecimal color specification) must be applied to 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 6 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 8 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.
- Line 10 specifies color `blue` for labels `Homo`, `Pan`, and `Hominini`.

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.

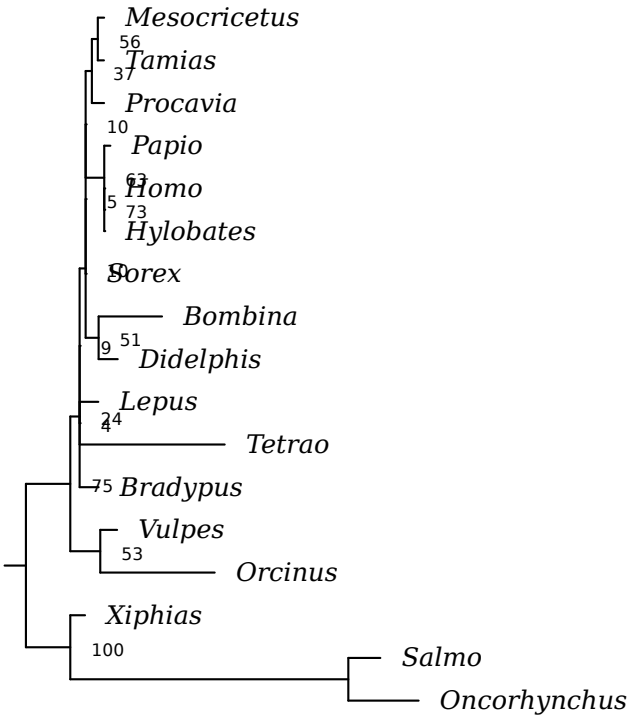
Label styles can also be applied globally. Option `-l` (lowercase l) specifies the leaf label style, option `-i` the inner node label style, and option `-b` the branch length style. For example, the following tree, which was produced using defaults, could be improved a bit:

```
$ 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:

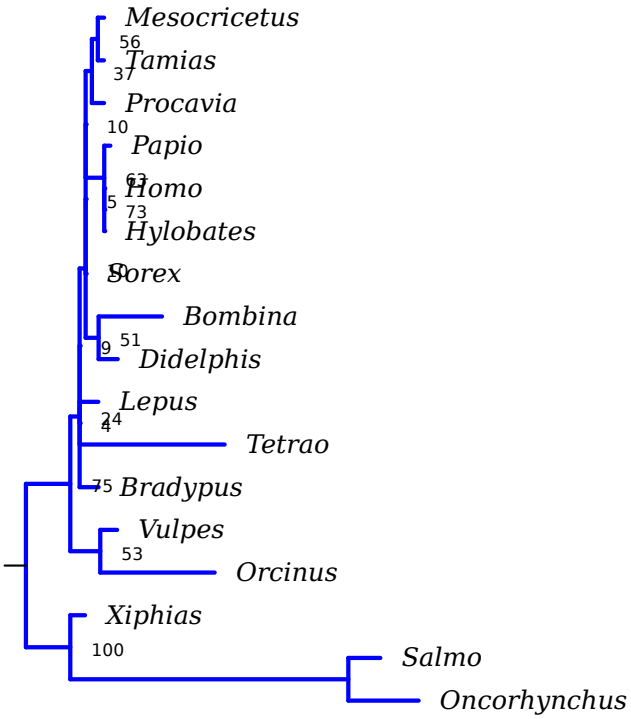
```
$ nw_display -s -S -v 20 -b 'opacity:0' -i 'font-size:8' \
-l 'font-family:serif;font-style:italic' vertebrates.nw
```



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
```



1.1.3 Ornaments

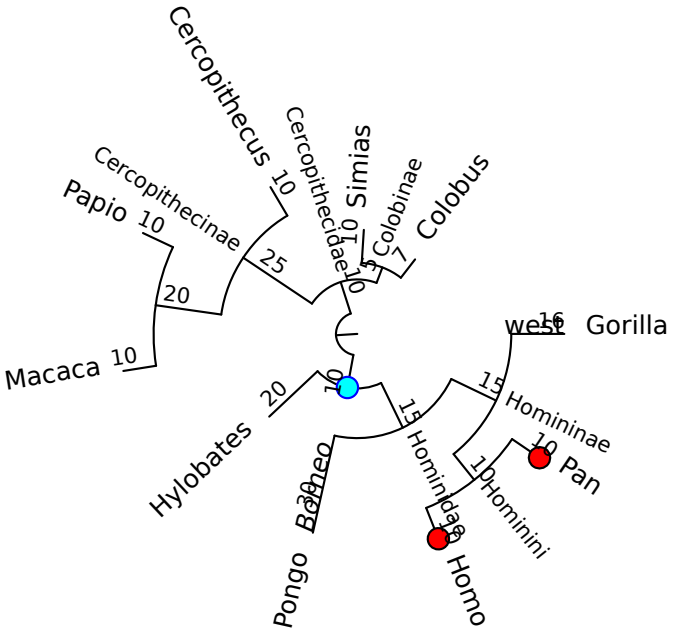
Ornaments are arbitrary snippets of SVG code that are displayed at specified node positions. Like CSS, this is done with a map. The ornament map has the same syntax as the CSS map, except that you specify SVG elements rather than CSS styles. The `Individual` keyword means that all nodes named on a given 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[5]. 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
```

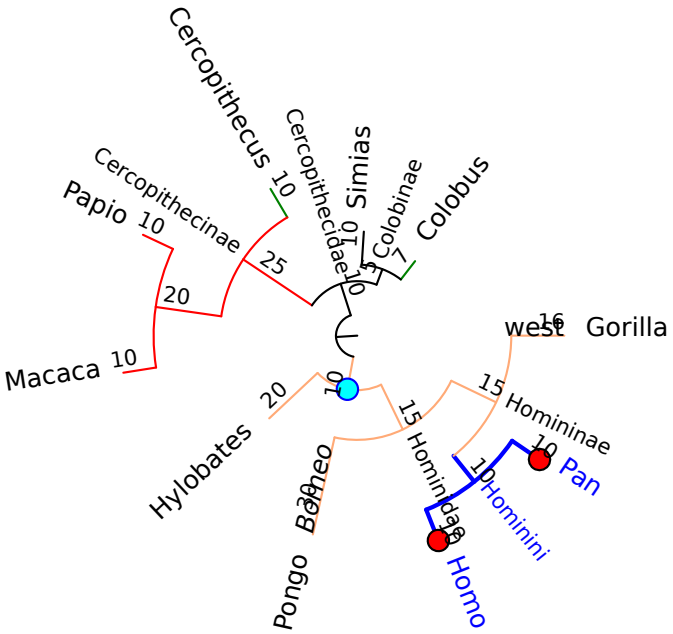
```
$ nw_display -sr -S -w 450 -o ornament.map catarrhini
```

[5] 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 **??**), 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.

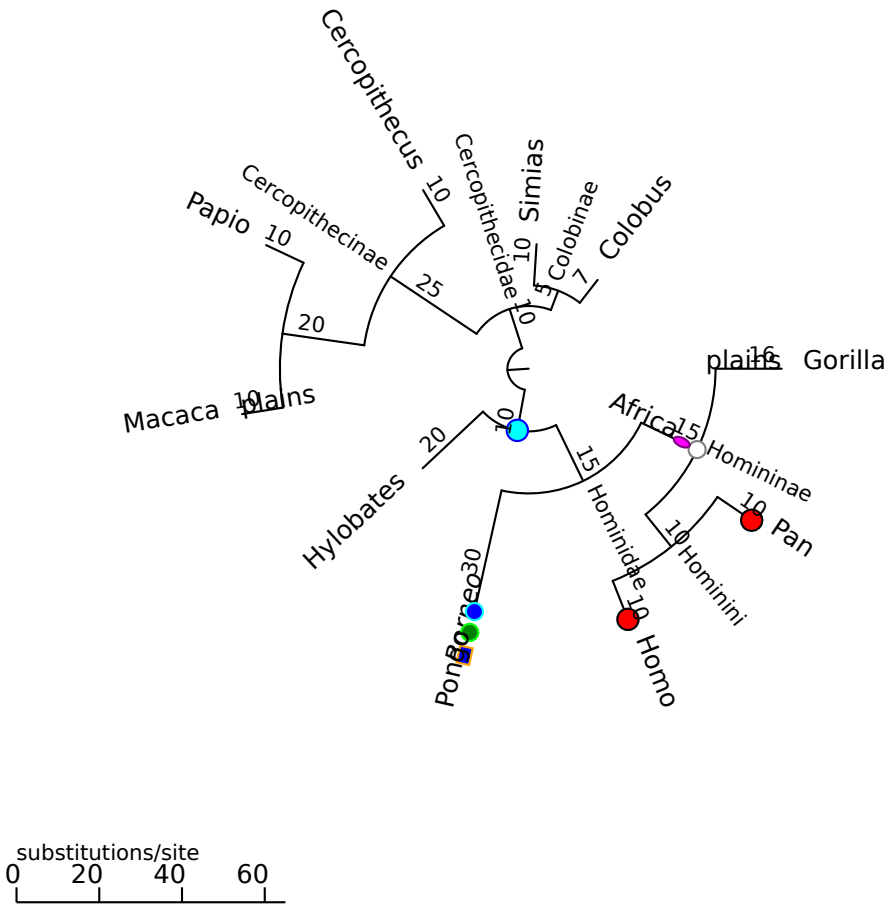
If applicable, these transforms must be applied to each element separately. This means that the SVG snippet must be *parsed* (instead of just wrapped in a `<g>` element, as is the case when libxml is not being used), and we use libxml's XML parser.

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
```

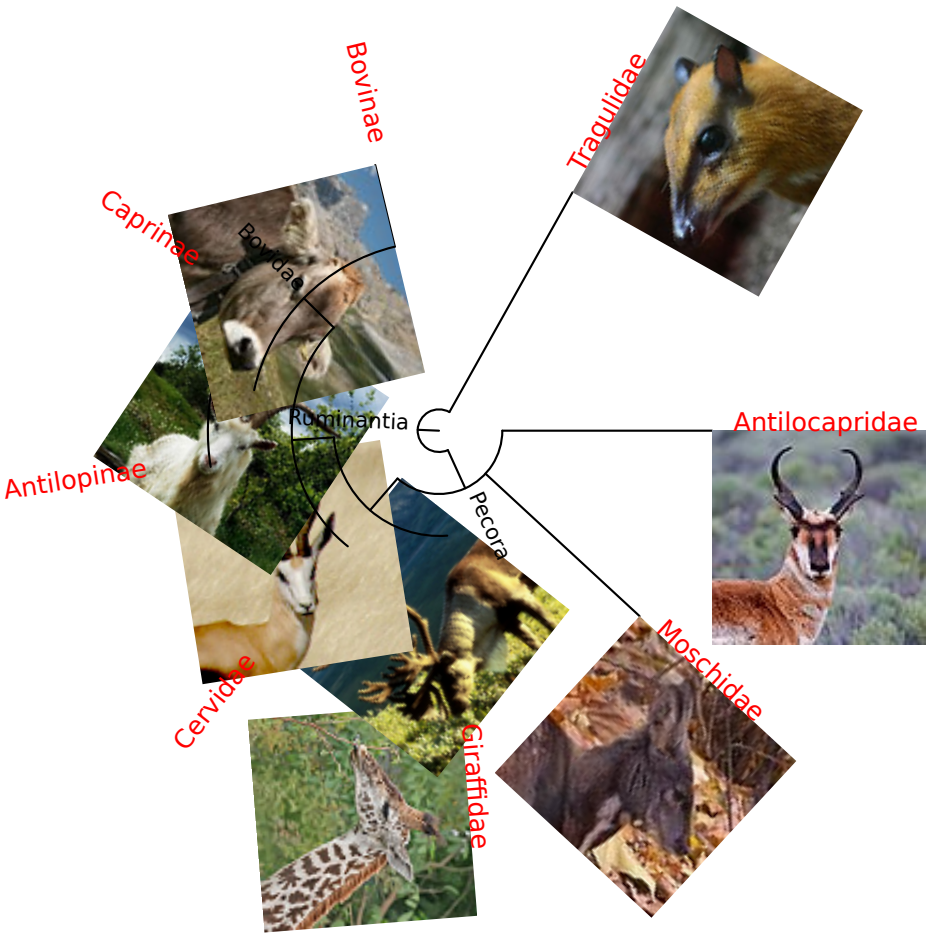


As hinted above, libxml also allows handling of images:

```
"<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

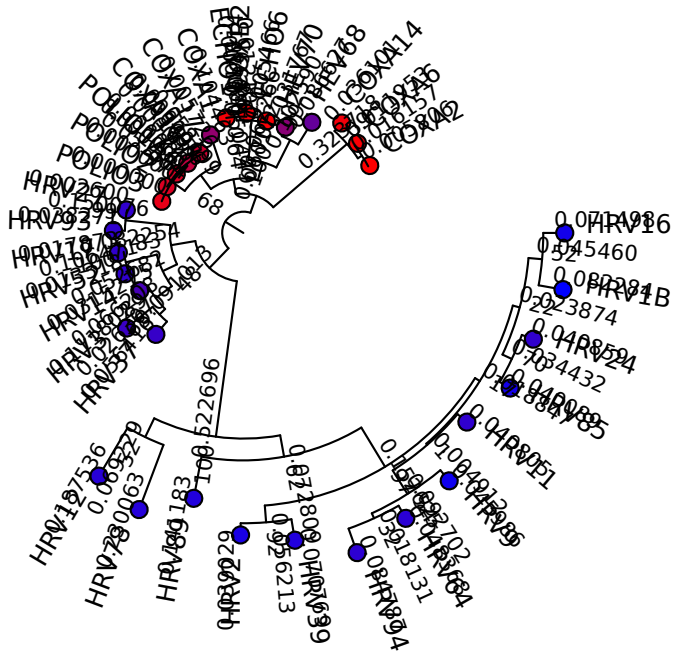
In a study of human rhinoviruses, I have produced a phylogenetic tree, `HRV_ingrp.nw`. I have also computed the GC content of the sequences, and mapped it into a gradient that goes from blue (33.3%) to red (44.5%). I used this gradient to produce a CSS map, `b2r.map`:

```
$ 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

nw_display: error while loading shared libraries: libnutils.so: cannot open shar
ed object file: No such file or directory
```

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.

1.1.4 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.

1.2 Displaying Tree Properties

`nw_stats` displays simple tree properties:

```
$ nw_stats catarrhini

nw_stats: error while loading shared libraries: libnutils.so: cannot open shared
 object file: No such file or directory
```

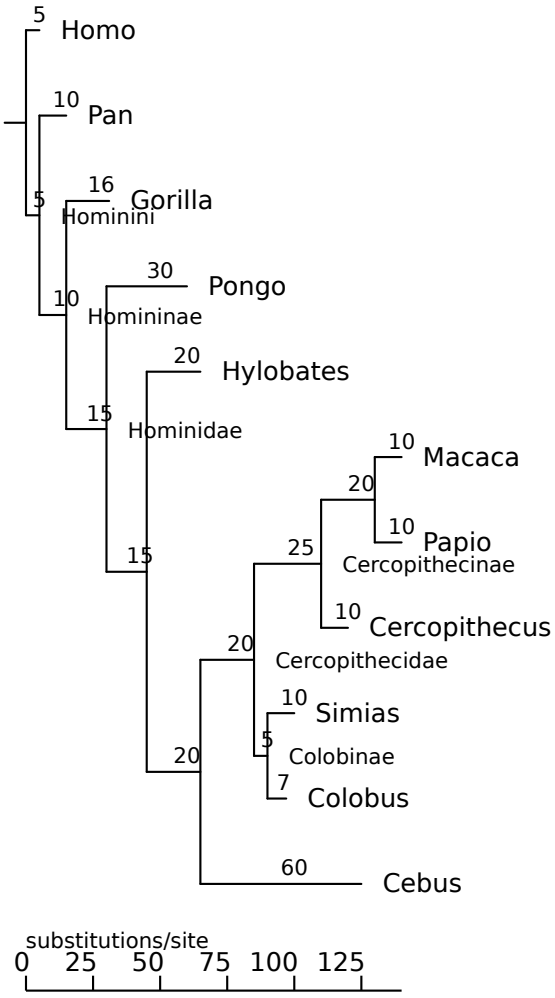
Option `-f l` causes the data to be printed linewise, without headers:

```
$ nw_stats -f l catarrhini

nw_stats: error while loading shared libraries: libnutils.so: cannot open shared
 object file: No such file or directory
```

1.3 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.

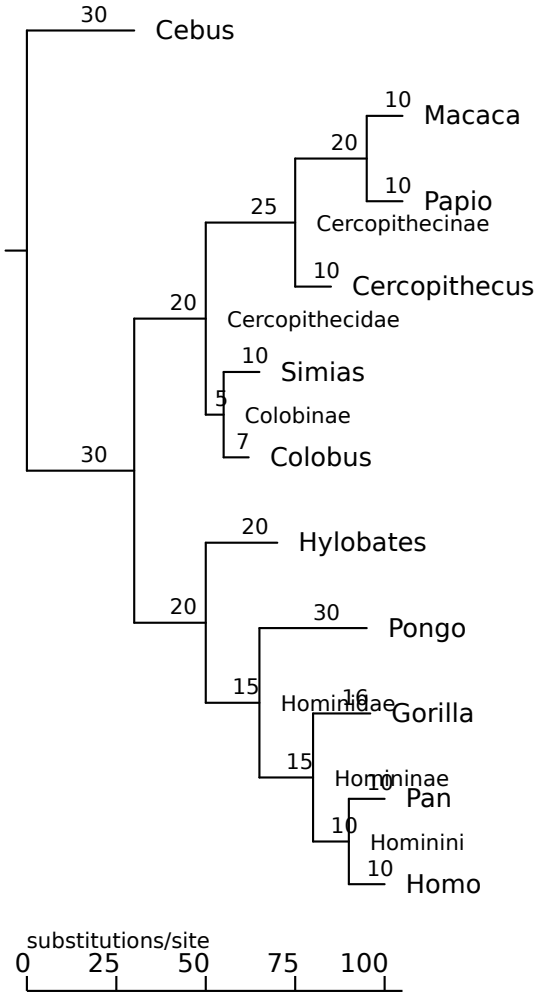
Rooting a tree is usually done by specifying an *outgroup* (but see **??**). In the simplest case, this is a single leaf. The root is then placed in such a way that one of its children is the outgroup, while the other child is the rest of the tree (sometimes known as the *ingroup*). Consider the following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

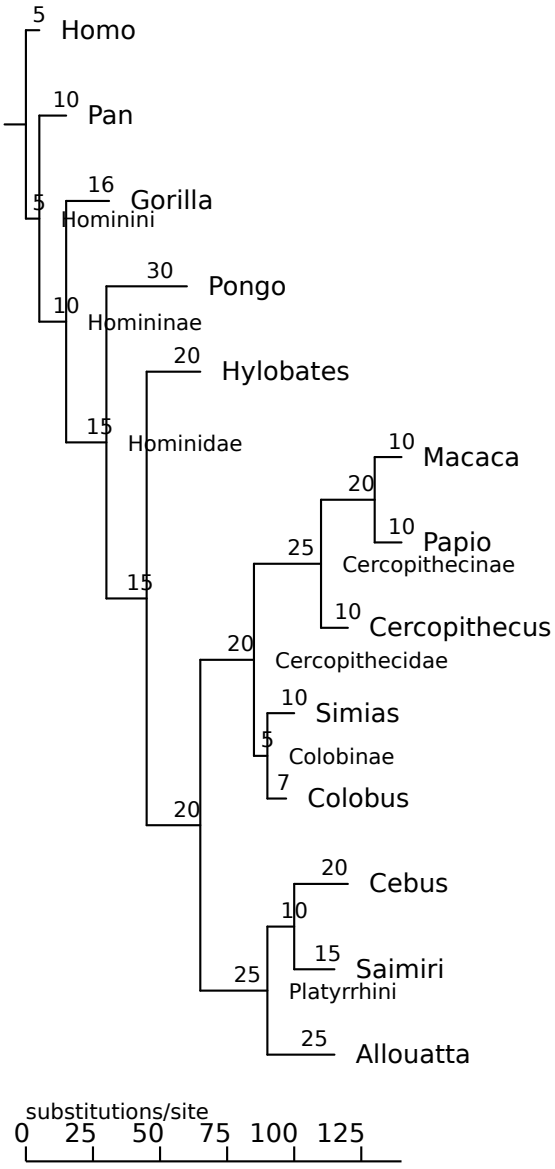
```
$ nw_reroot simiiformes_wrong Cebus | nw_display -s -
```

which produces:



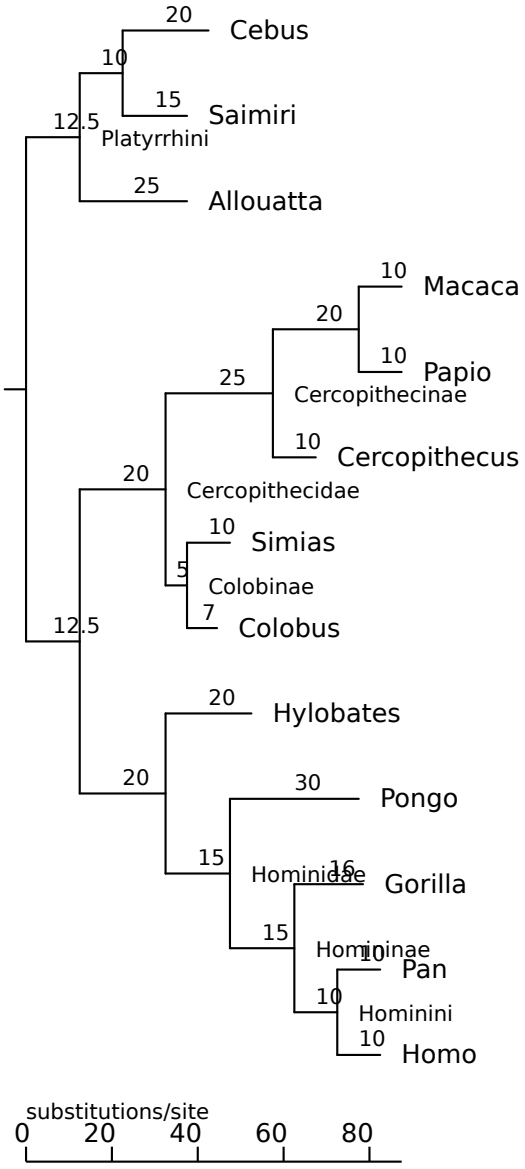
Now the tree is correct. Note that the root is placed in the middle of the ingroup-outgroup edge, and that the other branch lengths are conserved.

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 -
```



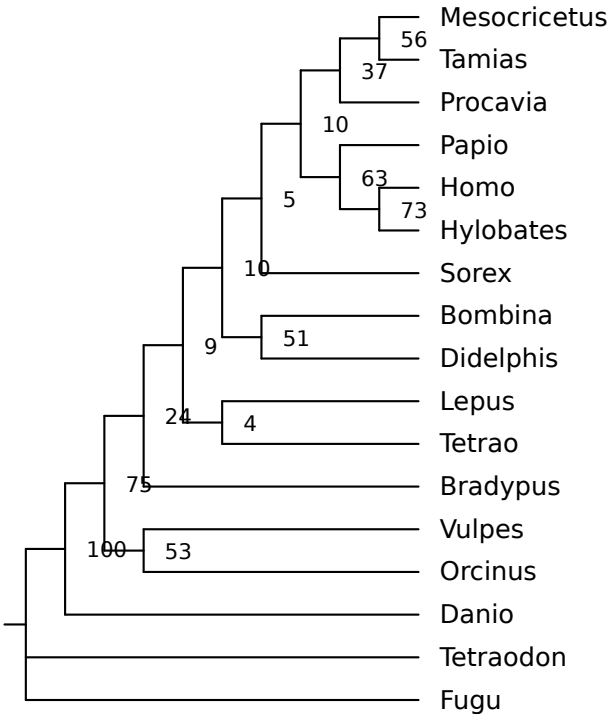
Note that I did not include all three New World monkeys, only `Cebus` and `Allouatta`. This is because it is always possible to define a clade using only two leaves. The result would be the same if I had included all three, though. You can use inner labels too, if there are any:

```
$ 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.

1.3.1 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:

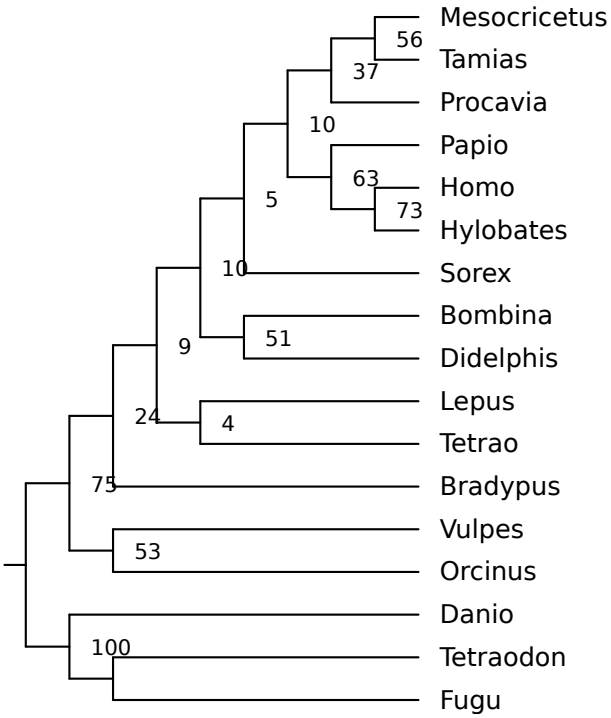
```
$ nw_reroot vrt1cg.nw Fugu Danio
```

But this will fail:

```
nw_reroot: error while loading shared libraries: libnutils.so: cannot open share
d object file: No such file or directory
```

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 -
```



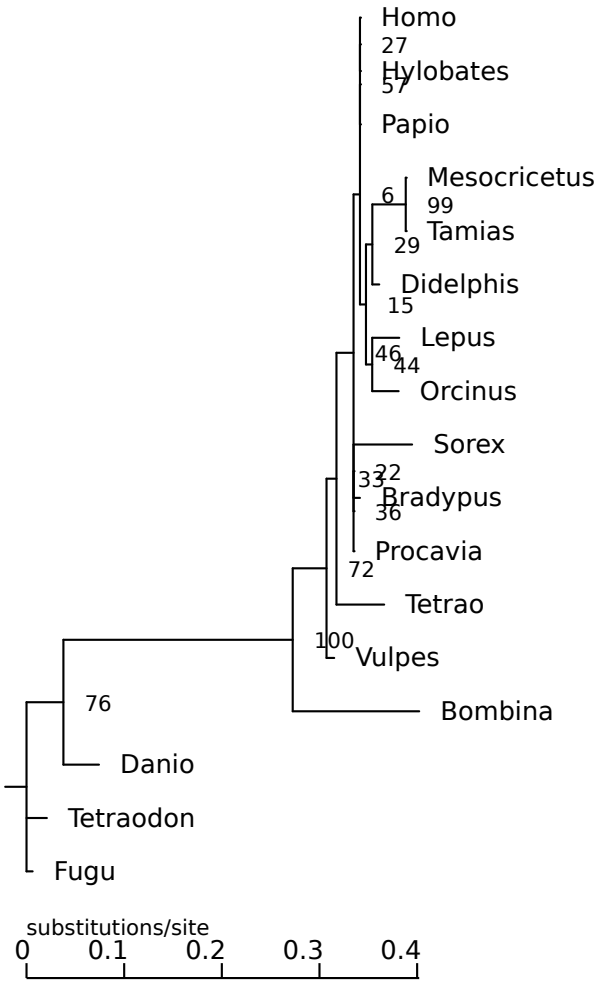
To repeat: all outgroup labels were passed, not just the two normally needed to find the last common ancestor – since, precisely, we can't use the LCA.

1.3.2 Rooting without an (explicit) outgroup

Sometimes it is not clear what node(s) should form the outgroup – perhaps the phylogeny of the group under study is not clear, or someone forgot to include an outgroup in the tree[6]. In that case, one can assume that the root is within the longest branch in the tree, and reroot there. Whether this assumption is reasonable is another question, indeed there are plenty of possible reasons why this may not be the case.

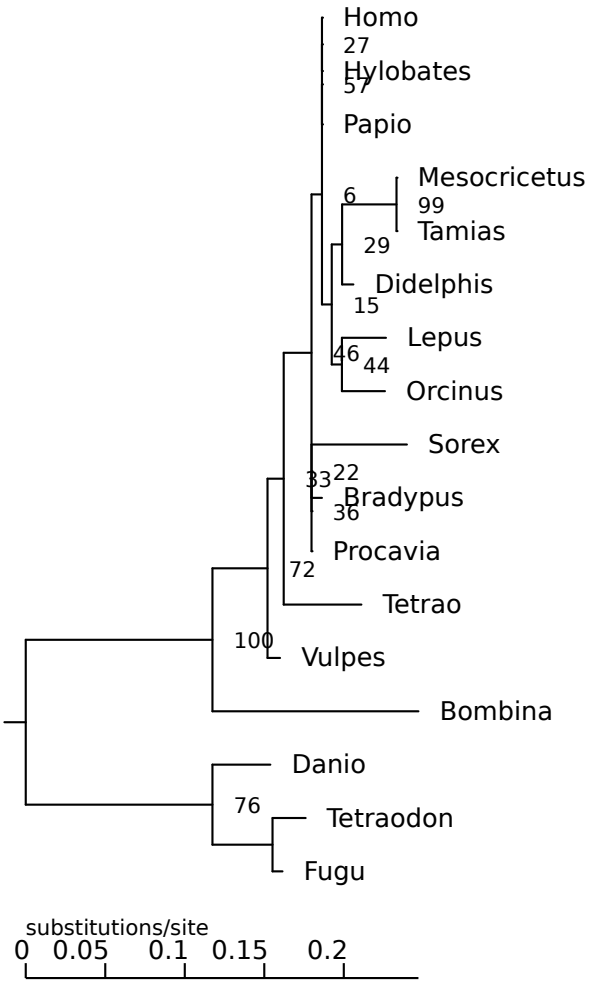
In any case, should you want to reroot on the longest branch, just call `nw_reroot` without specifying any labels. Consider this tree:

[6] This happens regulary in published trees



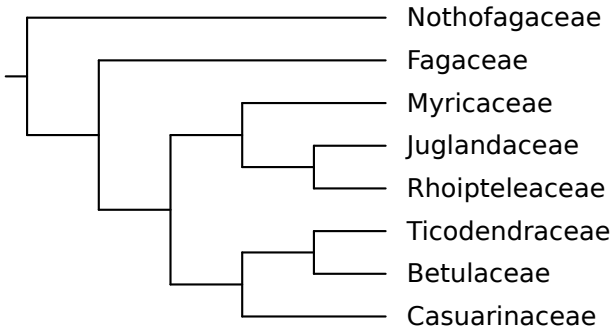
As it happens, rerooting on the longest branch gets it right:

```
$ nw_reroot vrt.nw | nw_display -s -b 'opacity:0;visibility:hidden' -v 25 -
```



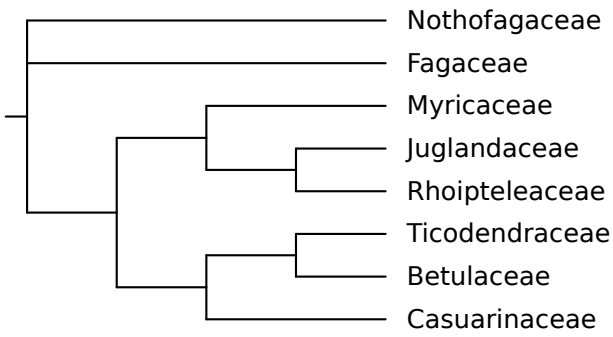
1.3.3 Derooting

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 -
```

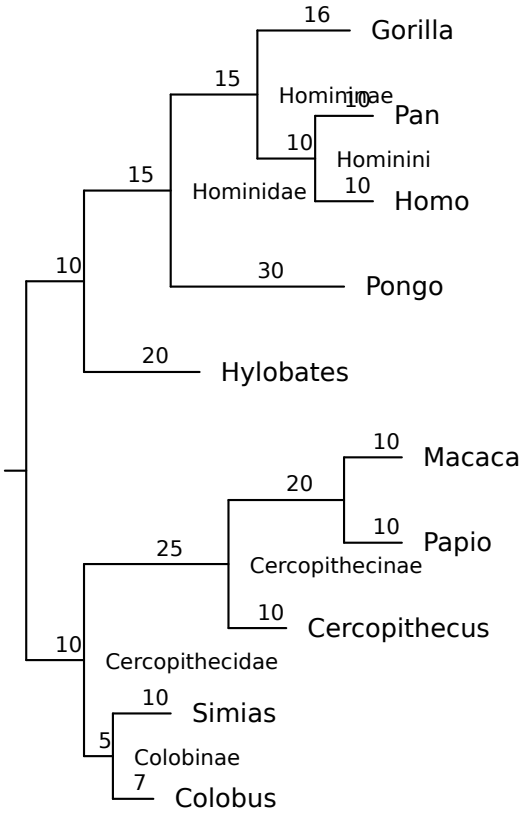


this works as follows. The program finds which of the root's two children (it is assumed to have two, otherwise the tree is already considered unrooted in the above sense) has more children than the other. This is considered the ingroup, and the LCA of the ingroup is spliced out from the tree, attaching 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).

1.4 Extracting Subtrees

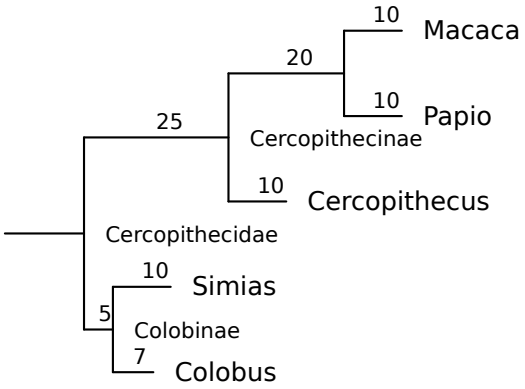
You can extract a clade (AKA subtree) from a tree with `nw_clade`. As usual, a clade is specified by a number of node labels, of which the program finds the last common ancestor, which unequivocally determines the clade (see Appendix **??**). We'll use the catarrhinian tree again for these examples:

```
$ nw_display -sS catarrhini
```



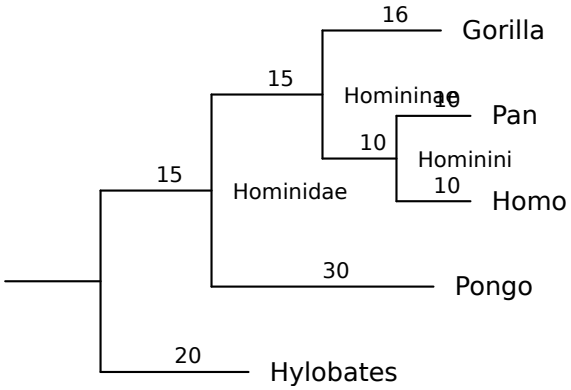
In the simplest case, the clade you want to extract has its own, unique label. This is the case of `Cercopithecidae`, so you can extract the whole cercopithecid subtree (Old World monkeys) using just that label:

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



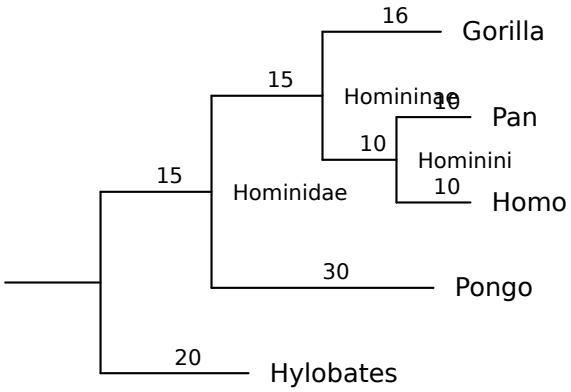
Now suppose I want to extract the apes subtree. These are the Hominidae ("great apes") plus the gibbons (*Hylobates*). But the corresponding node is unlabeled in our tree (it would be `Hominoidea`), so we need to specify (at least) two descendants:

```
$ 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.

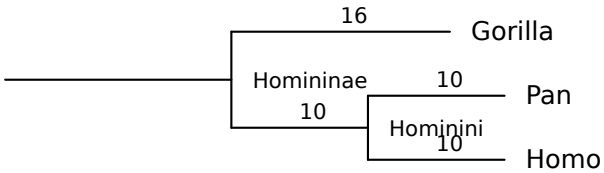
```
$ nw_clade catarrhini Hominidae Hylobates | nw_display -sS -
```



1.4.1 Monophyly

You can check if a set of leaves[7] 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:

```
$ nw_clade -m catarrhini Hylobates Pongo

[no output]
```

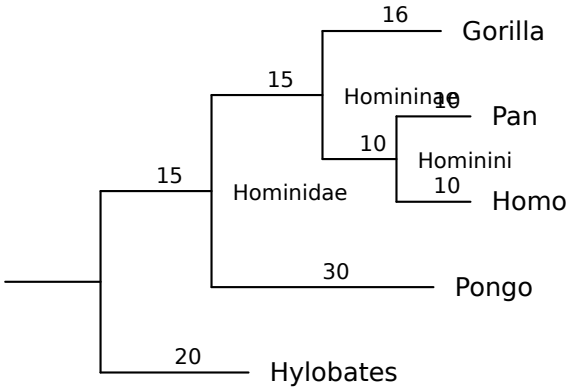
Maybe early hominines split from orangs in South Asia before moving to Africa.

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

1.4.2 Context

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

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

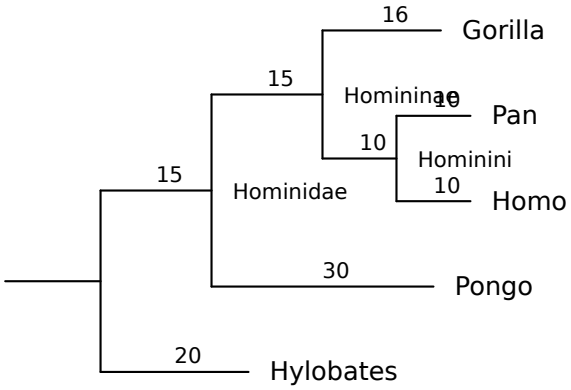


In this case, `nw_clade` computed the LCA of `Gorilla` and `Homo`, "climbed up" two levels, and output the subtree at that point. This is useful when you want to extract a clade with its nearest neighbor(s). I use this when I have several trees in a file and my clade's nearest neighbors aren't always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

1.4.3 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.

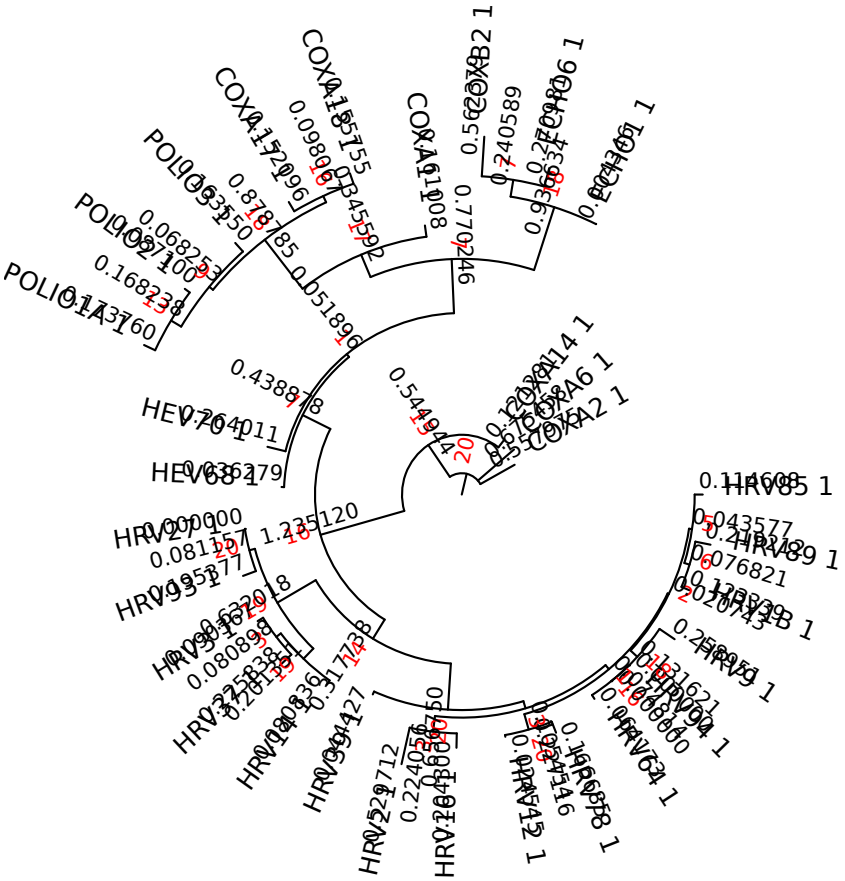
1.4.4 Limits

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

1.5 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.

1.6 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`:

```
nw_display: error while loading shared libraries: libnutils.so: cannot open shar
ed object file: No such file or directory
```

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:

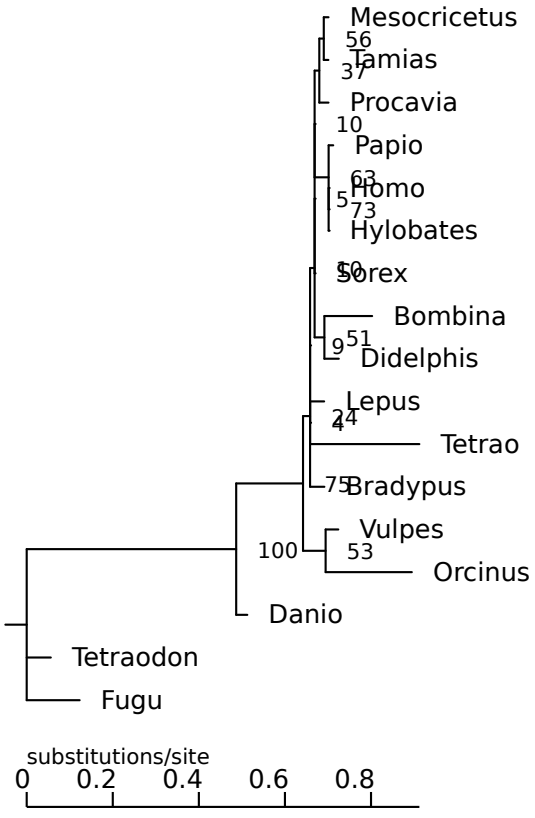
```
$ nw_topology vrt1.nw | nw_display -w 60 -
```

```
nw_display: error while loading shared libraries: libnutils.so: cannot open shar
ed object file: No such file or directory
nw_topology: error while loading shared libraries: libnutils.so: cannot open sha
red object file: No such file or directory
```

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.

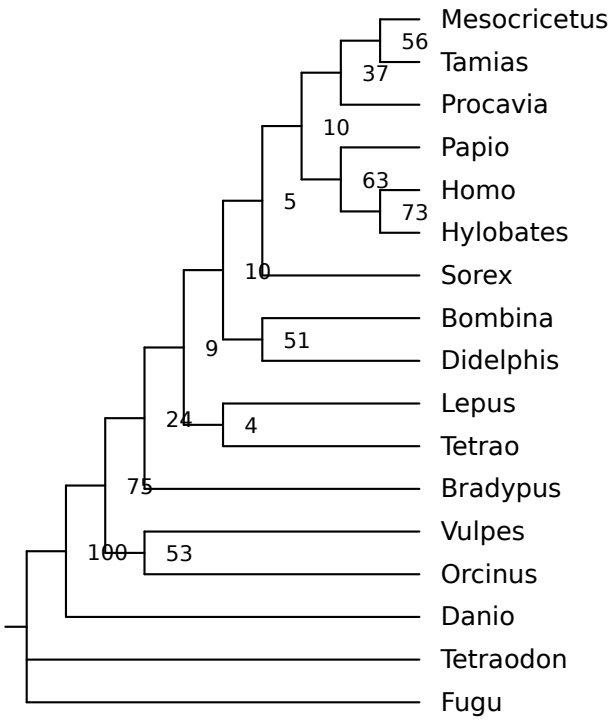
Of course, ASCII trees have low resolution in the first place, so I'll show both trees look in SVG. First the original:

```
$ 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:

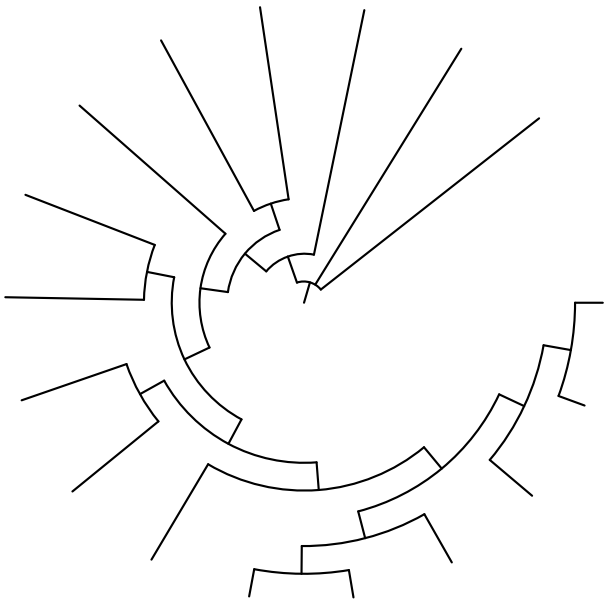
```
$ nw_topology -IL vrt1.nw
```

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

```
nw_topology: error while loading shared libraries: libnutils.so: cannot open sha
red object file: No such file or directory
```

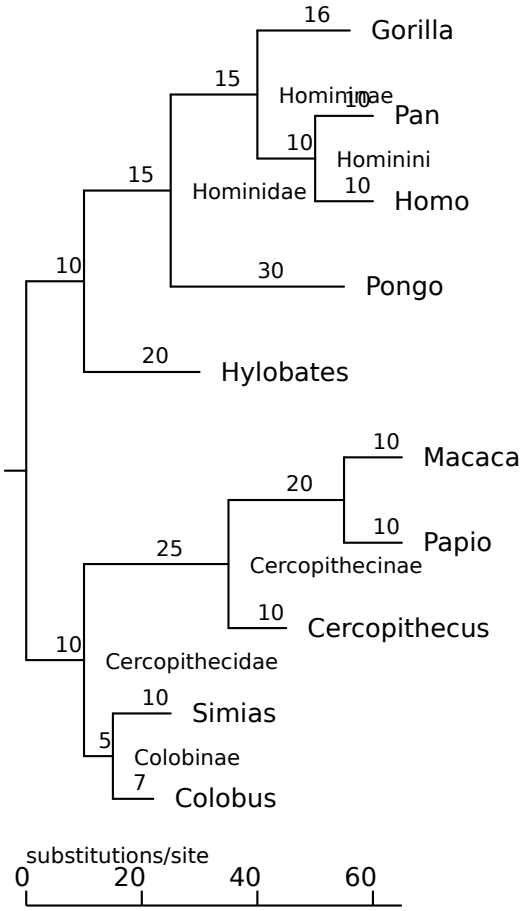
Let's look at it as a radial tree, for a change:

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



1.7 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

nw_distance: error while loading shared libraries: libnutils.so: cannot open sha
red object file: No such file or directory
```

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

nw_distance: error while loading shared libraries: libnutils.so: cannot open sha
red object file: No such file or directory
```

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.

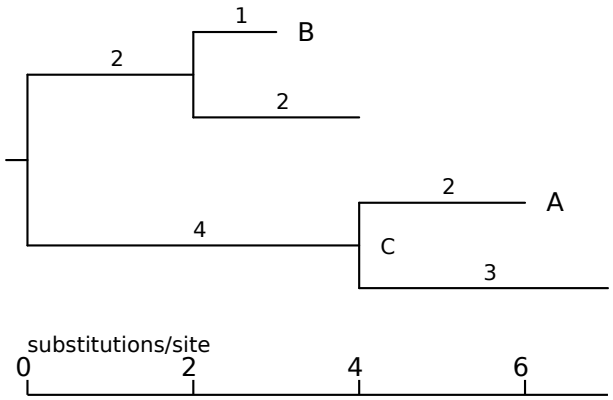
1.7.1 Selection

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
```

```
nw_distance: error while loading shared libraries: libnutils.so: cannot open sha
red object file: No such file or directory
```

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
```

```
nw_distance: error while loading shared libraries: libnutils.so: cannot open sha
red object file: No such file or directory
```

All leaves

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

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

```
nw_distance: error while loading shared libraries: libnutils.so: cannot open sha
red object file: No such file or directory
```

All inner nodes

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

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

```
nw_distance: error while loading shared libraries: libnutils.so: cannot open sha
red object file: No such file or directory
```

All nodes

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

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

```
nw_distance: error while loading shared libraries: libnutils.so: cannot open sha
red object file: No such file or directory
```

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.

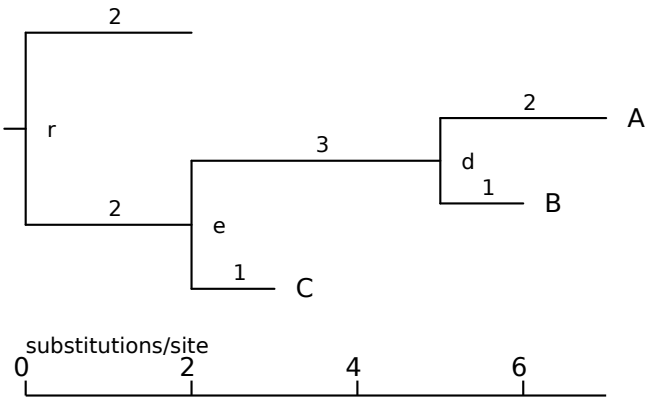
```
$ nw_distance -n dist_sel_xpl.nw A C
```

```
nw_distance: error while loading shared libraries: libnutils.so: cannot open sha
red object file: No such file or directory
```

1.7.2 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
```

```
nw_distance: error while loading shared libraries: libnutils.so: cannot open sha
red object file: No such file or directory
```

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
```

```
nw_distance: error while loading shared libraries: libnutils.so: cannot open sha
red object file: No such file or directory
```

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
```

```
nw_distance: error while loading shared libraries: libnutils.so: cannot open sha
red object file: No such file or directory
```

1.7.3 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
```

```
nw_distance: error while loading shared libraries: libnutils.so: cannot open sha
red object file: No such file or directory
```

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

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

```
nw_distance: error while loading shared libraries: libnutils.so: cannot open sha
red object file: No such file or directory
```

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

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

```
nw_distance: error while loading shared libraries: libnutils.so: cannot open sha
red object file: No such file or directory
```

1.8 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 **??**)
4. branch lengths are discarded

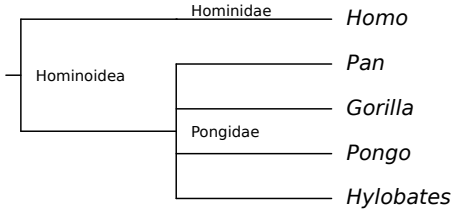
At this point, the modified pattern tree is compared to the modified target, and if the Newick strings are identical, the match is successful.

Example: finding trees with a specified subtree topology

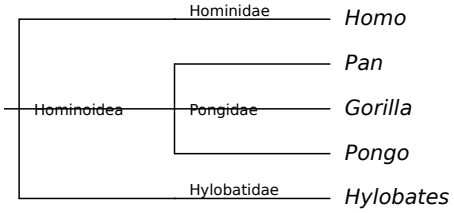
File `hominoidea.nw` contains seven trees corresponding to successive theories about the phylogeny of apes (these were taken from `http://en.wikipedia.org/wiki/Hominoidea`). Let us see which of them group humans and chimpanzees as a sister clade of gorillas (which is the current hypothesis).

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

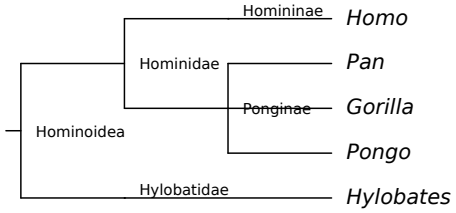
1 (until 1960)



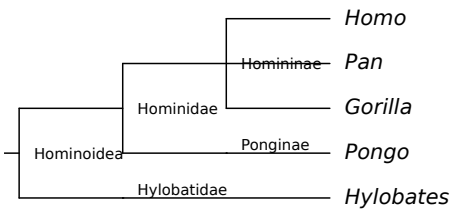
2 (Goodman, 1964)



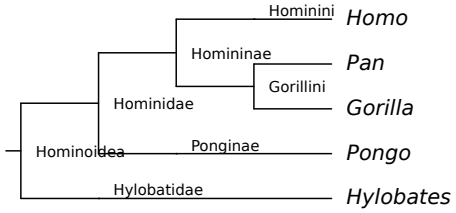
3 (gibbons as outgroup)



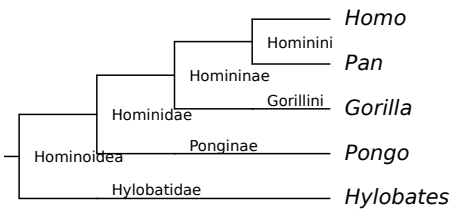
4 (Goodman, 1974: orangs as outgroup)



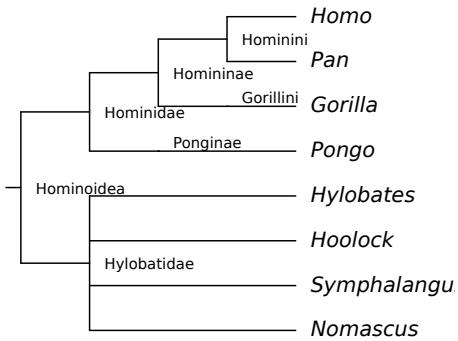
5 (resolving trichotomy)



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 -
```

```
nw_display: error while loading shared libraries: libnutils.so: cannot open shar
ed object file: No such file or directory
nw_match: error while loading shared libraries: libnutils.so: cannot open shared
 object file: No such file or directory
```

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.

1.9 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 **??**)
- 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.

1.9.1 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:

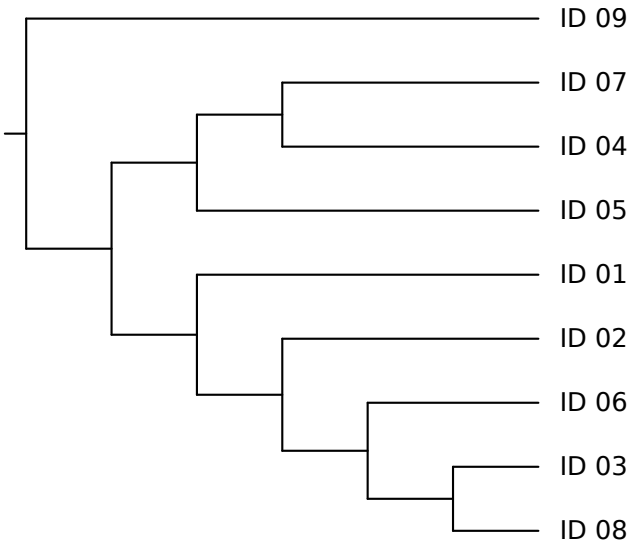
```
 154 259
ID_01     PTTSNSAPAL DAAETGHTSG ...
ID_02     SVSSHSVPAL DAAETGHTSS ...
...
```

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:

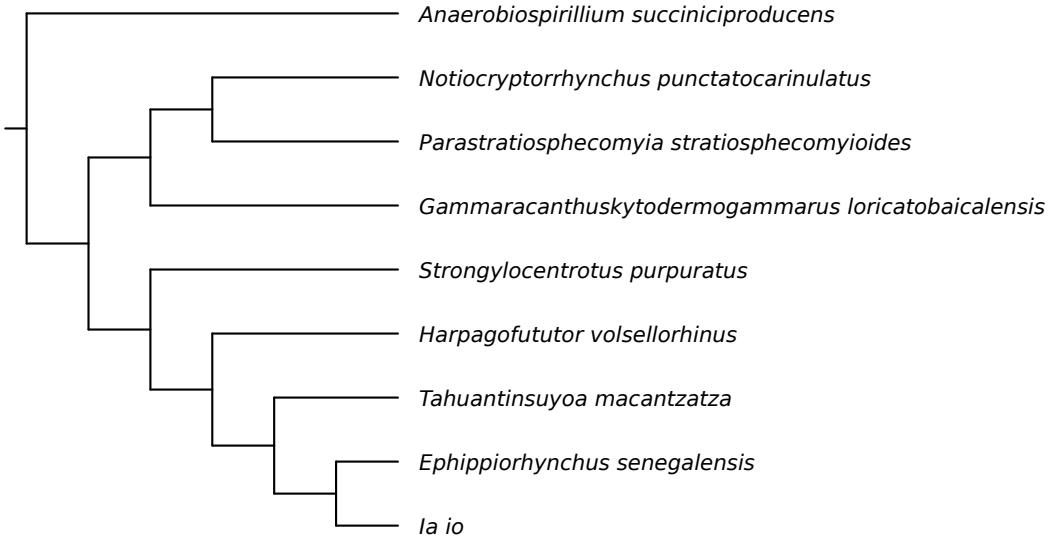
```
$ nw_display -s short_IDs.nw -v 30
```



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

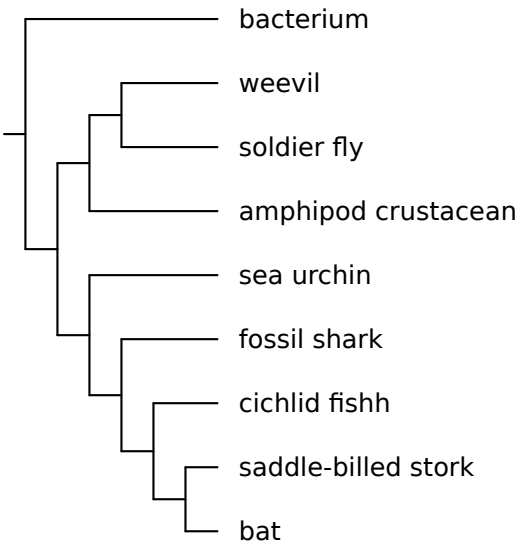
```
$ nw_rename short_IDs.nw id2longname.map \
| nw_display -s -l 'font-size:small;font-style:italic' -w 500 -v 30 -W 6 -
```

(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:

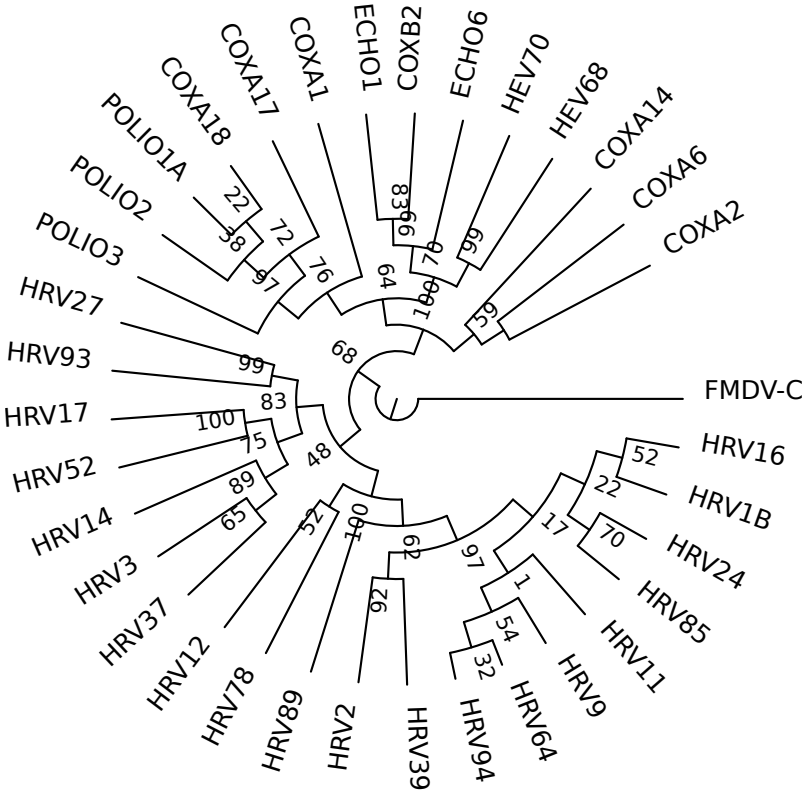
```
$ nw_rename short_IDs.nw id2longname.map \
| nw_rename - longname2english.map \
| nw_display -s -v 30 -W 10 -
```



1.9.2 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 **??**) 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`).

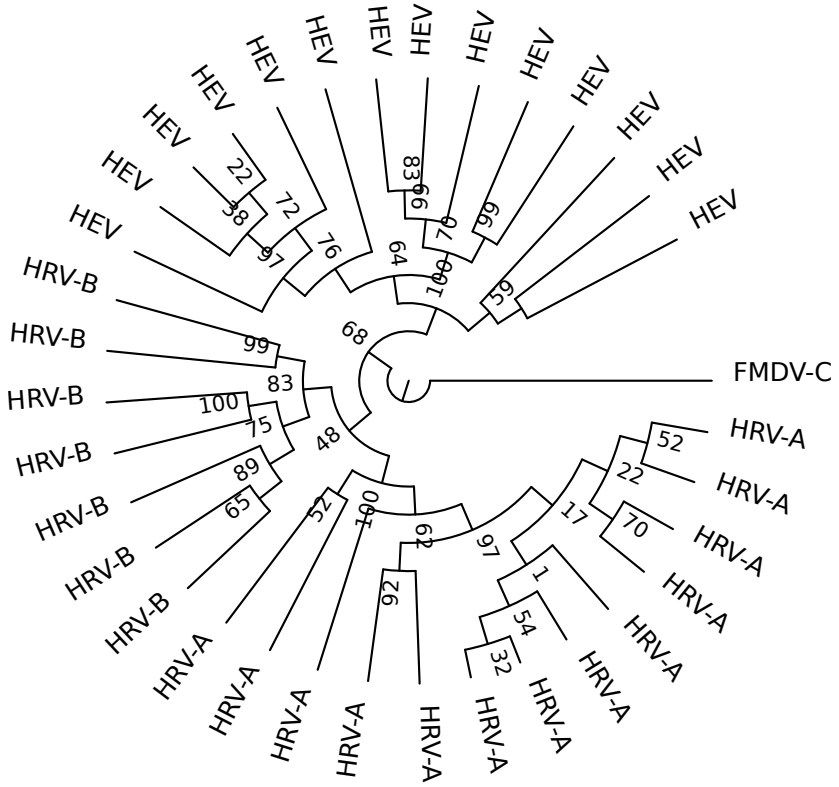
```
$ nw_topology HRV_FMDV.nw | nw_display -sr -w 400 -
```



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 **??**):

```
# 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 -
```

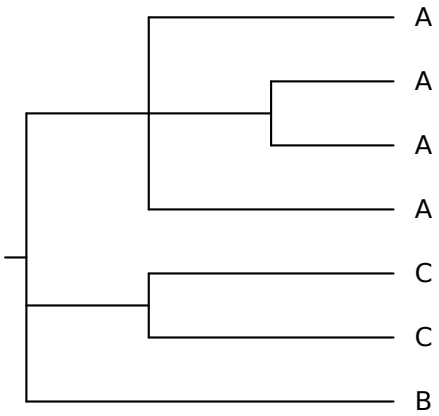


As we can see, it does. This would be even better if we could somehow simplify the tree so that clades of the same species were reduced to a single leaf. And, that's exactly what `nw_condense` does (see below).

1.10 Condensing

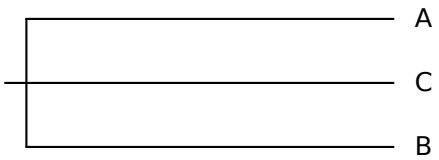
Condensing a tree means reducing its size in a systematic, meaningful way (compare this to *pruning* (**??**) which arbitrarily removes branches, and to *trimming* (**??**) which

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 -
```

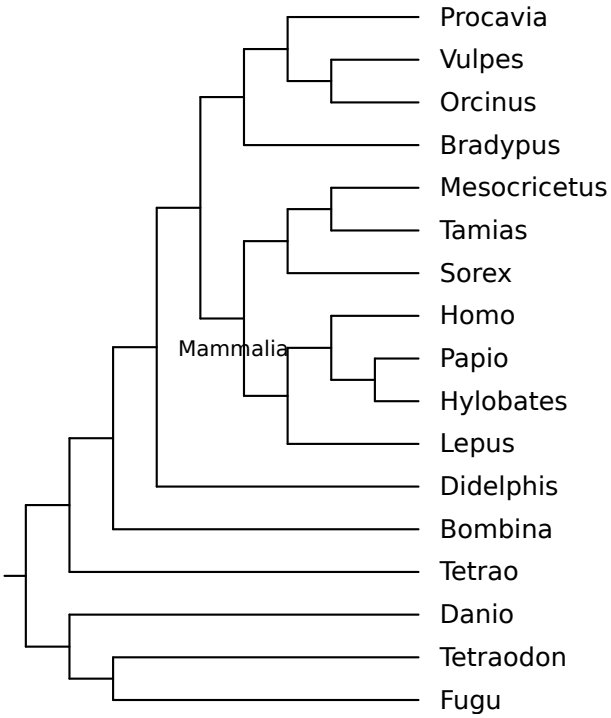


Now the A and B leaves stand for whole clades. The tree is simpler, but the information about the relationships of A, B and C is conserved, while the details of the A and C clades is not. A typical use of this is producing genus trees from species trees (or any higher-level tree from a lower-level one),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

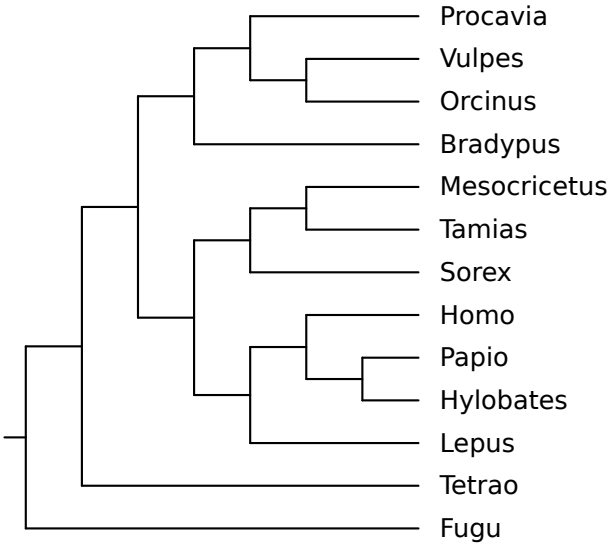
1.11 Pruning

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`:

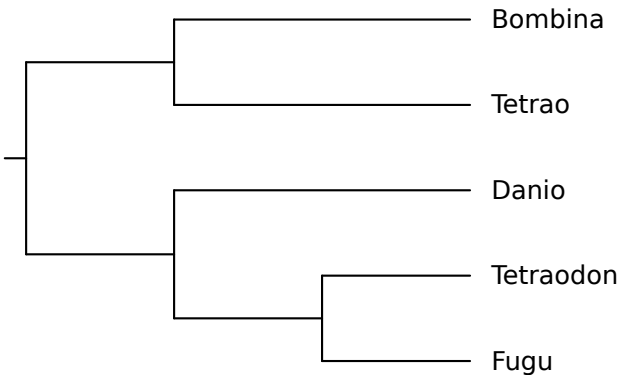
```
$ nw_prune vrt2_top.nw Tetraodon Danio Bombina Didelphis \
| nw_display -s -v 20 -
```



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 -
```



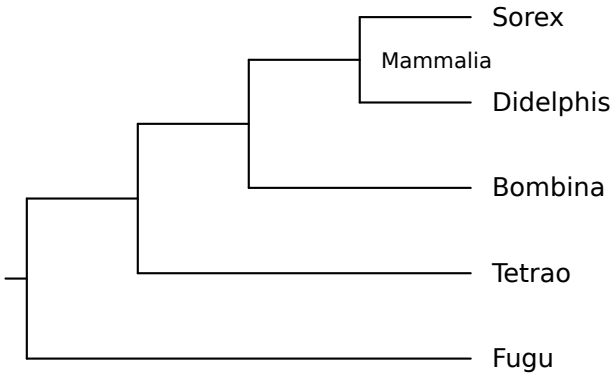
By the way, *Tetrao* and *Tetraodon* are not the same thing, the first is a bird (grouse), the second is a pufferfish.

1.11.1 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.[8].

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 -
```

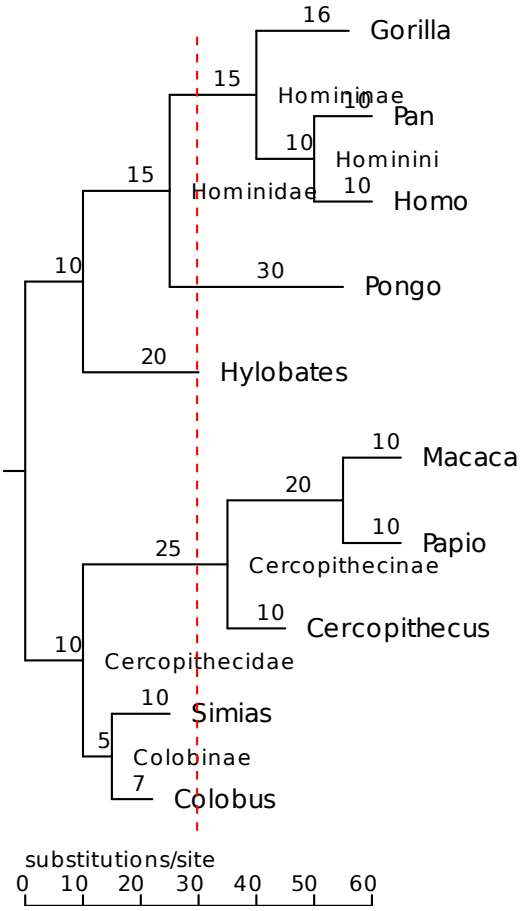


Note that I also passed `Mammalia`, for the reason discussed above: the node with this label would go away if I did not, resulting in a different tree (try it out).

1.12 Trimming trees

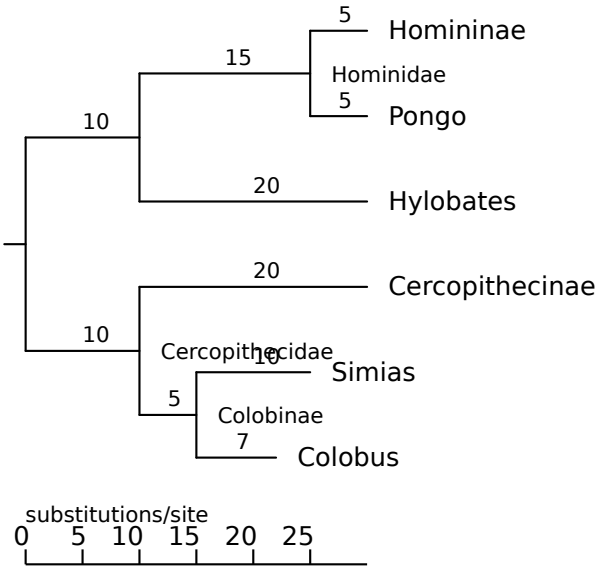
Trimming a tree means cutting the nodes whose depth is larger than a specified threshold. Here is what will happen if I cut the catarrhini tree at depth 30:

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



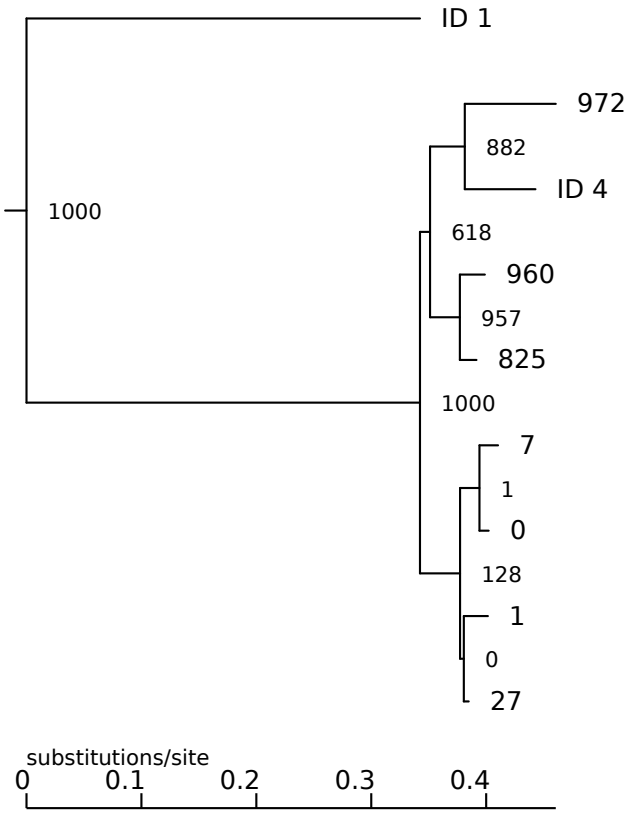
The tree will be "cut" on the red line, and everything right of it will be discarded:

```
$ 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`.

1.13 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[9], the indented form is still valid. For example, this is a tree in compact form, in file `falconiformes`:

[9] except between quotes

```
(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);
```

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.[10] 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);
```

[10] 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 -
```

1.14 Extracting Labels

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

```
$ nw_labels catarrhini
```

```
nw_labels: error while loading shared libraries: libnutils.so: cannot open share
d object file: No such file or directory
```

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

```
$ nw_labels -I catarrhini
```

```
nw_labels: error while loading shared libraries: libnutils.so: cannot open share
d object file: No such file or directory
```

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
```

```
nw_labels: error while loading shared libraries: libnutils.so: cannot open share
d object file: No such file or directory
```

If you just want the root's label, pass `-r`. In conjunction with `nw_clade` (see **??**), 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 **??**:

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

```
nw_labels: error while loading shared libraries: libnutils.so: cannot open share
d object file: No such file or directory
nw_clade: error while loading shared libraries: libnutils.so: cannot open shared
 object file: No such file or directory
```

1.14.1 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):

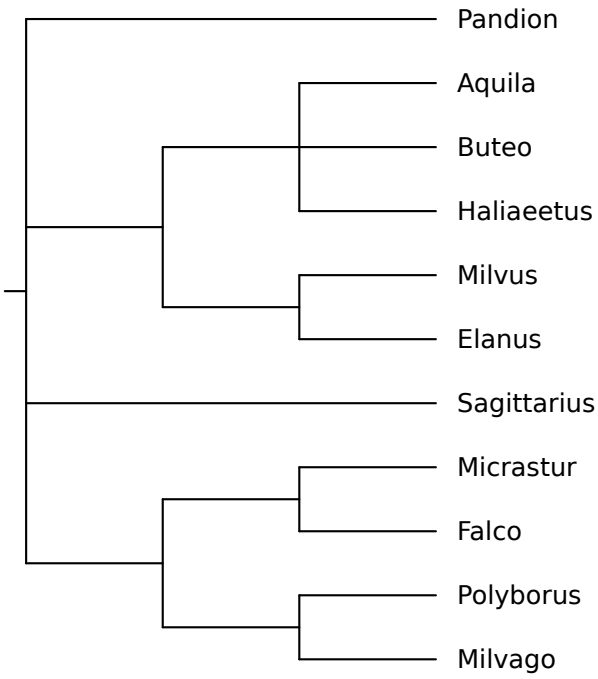
```
$ nw_labels -I catarrhini | wc -l
```

```
nw_labels: error while loading shared libraries: libnutils.so: cannot open share
d object file: No such file or directory
0
```

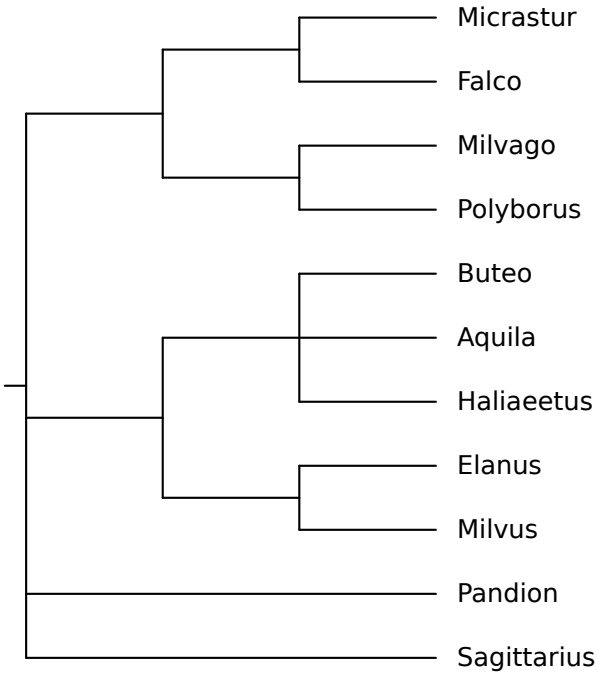
1.15 Ordering Nodes

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[11]:

```
$ 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
```

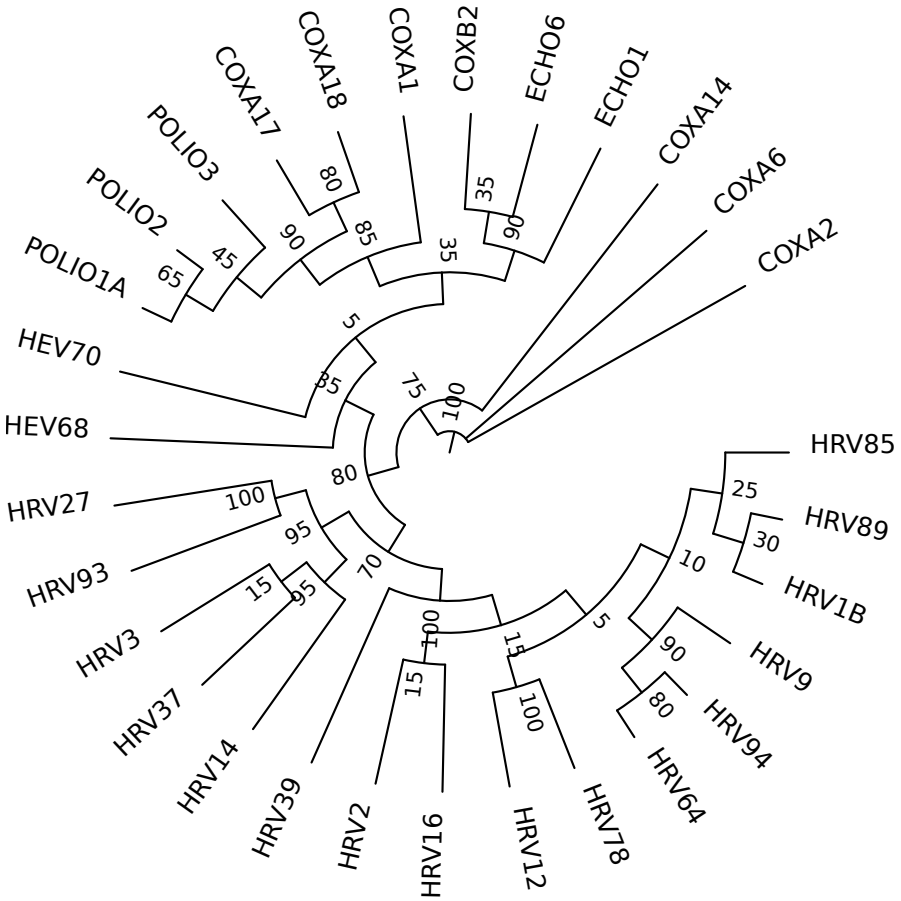
[11] One could also compute a checksum using `md5sum`, etc

```
nw_order: error while loading shared libraries: libnutils.so: cannot open shared
 object file: No such file or directory
nw_order: error while loading shared libraries: libnutils.so: cannot open shared
 object file: No such file or directory
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 **??**).

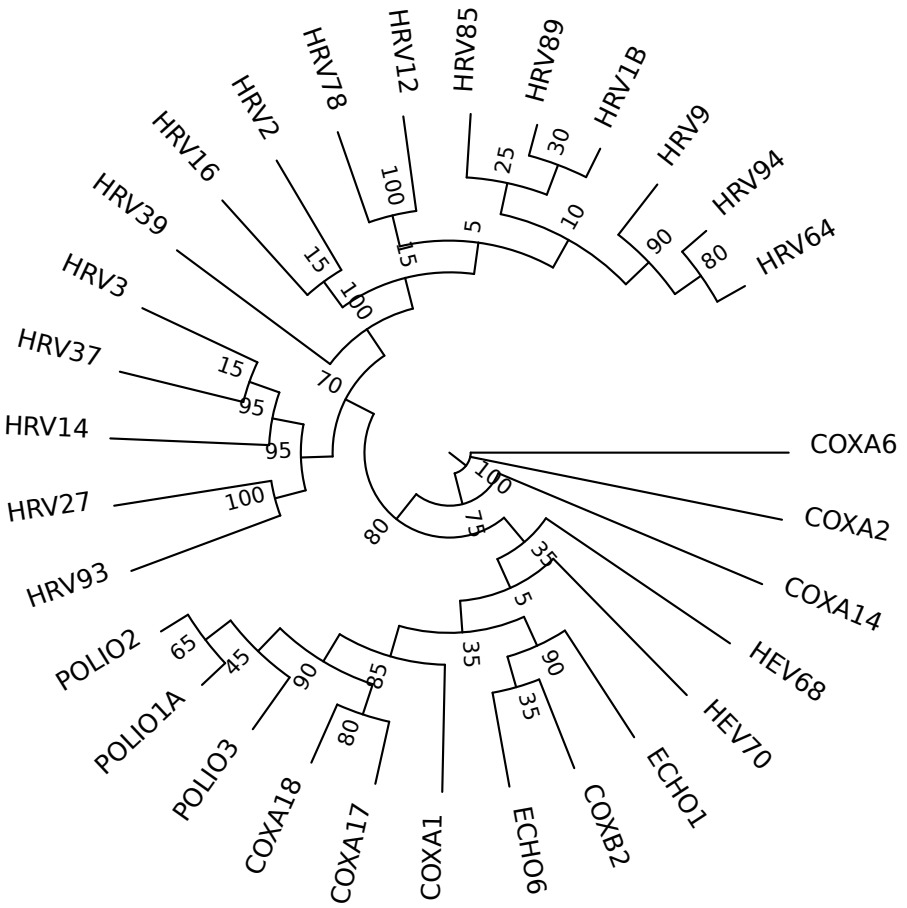
1.15.1 Variants

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:



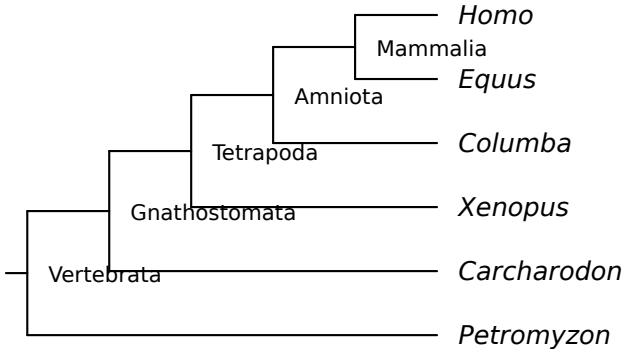
Here is the same tree, reordered by number of descendants: light nodes appear before (clockwise) heavy nodes:

```
$ 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 [**?**]) to the following mistakes:

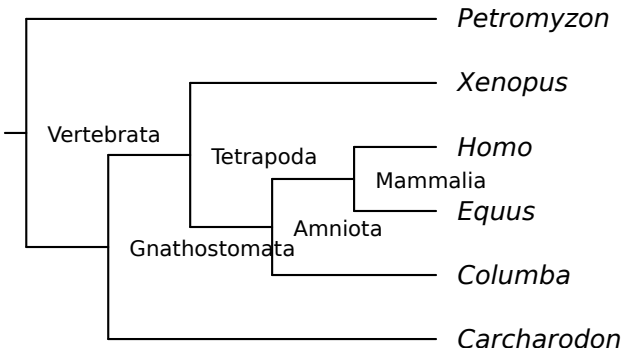


- there is a "chain of being" with "higher" and "lower" organisms, with (surprise!) humans at the top; "higher" can be interpreted in various ways, including "more perfect", or "more evolved" or even morally superior. This is known as the *scala naturæ* fallacy.
- there is a "main line" that progressively 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 [**?**]. 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:

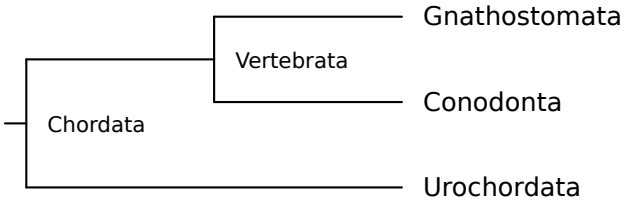
```
$ nw_order -c d scala.nw \
| nw_display -s -v 30 -l 'font-style:italic' -
```



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.

1.16 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

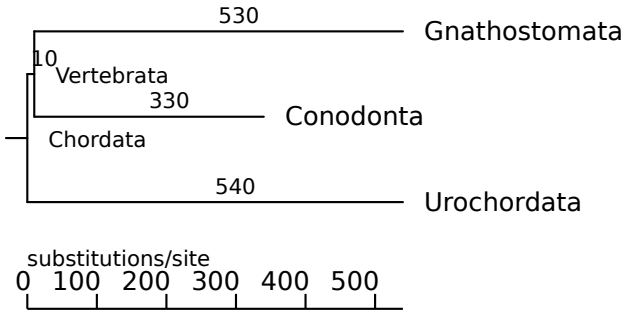
We can use the "branch length" field of Newick to specify ages, like this:

```
$ 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[12]. Note that the `Gnathostomata` leaf has no age: this means that there are still living gnathostomes (such as you and I[13]); 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.

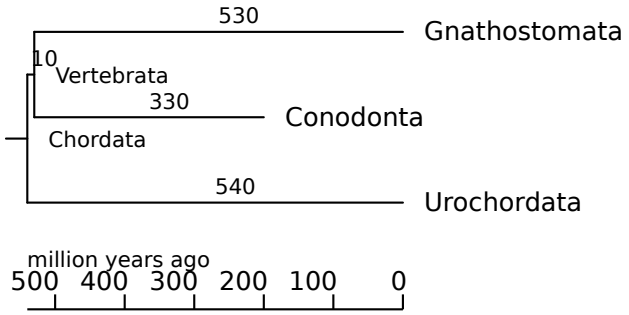
Now, if we were to display this tree without further ado, it would be nonsense. We have to convert the ages into durations, and this is the function of `nw_duration`:

```
$ 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' -
```

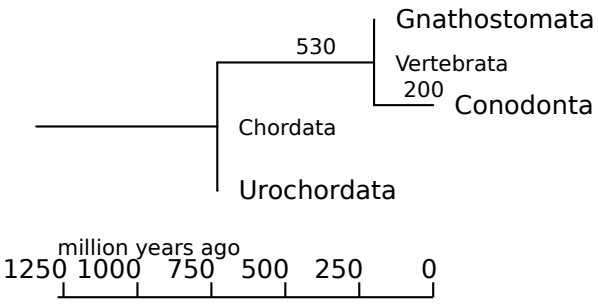


[12] Of course there are other branches in the vertebrate lineage, but they are not shown in this tree

[13] 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
```

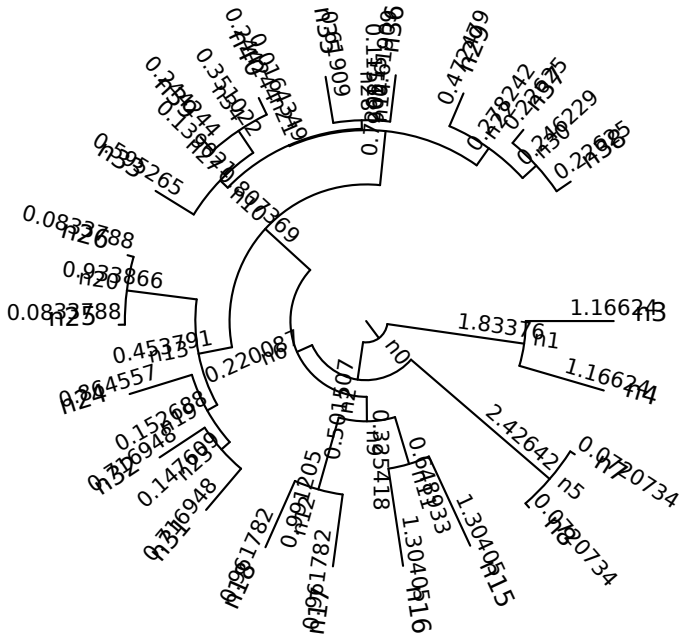


Now it looks as though only the conodonts are still alive, while the gnathostomes and urochordates each had brief flashes of existence 200 and 730 million years ago, respectively. Don't show this to a palaeontologist.

1.17 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.

1.18 Stream editing

1.18.1 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 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.[14] Sed is specialized for editing text streams, and `grep` is designed for precisely the line-finding task in question.[15] 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 any task. Sed will lie in between. The Newick Utilities 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 three programs 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 has no external dependency. Finally, I understand that Scheme is not the only solution as an embedded language, and that many people (myself included) find learning it a bit of a challenge. Therefore, I tried the same approach with Lua[16] (`http://www.lua.org`), which is designed as an embeddable language, is even smaller than Guile, and by most accounts easier to learn.[17]. The result, `nw_luaed`, is probably the best so far: as powerful as `nw_sched`, while smaller,

[14] Ok, Perl was *initially* designed for processing text – it's the Practical Extraction and Report Language, after all – but it has long grown out of this initial specialization.

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

[16] In case you're curious, the reason I tried Scheme before Lua is simply that I heard about them in that order.

[17] And, in my experience, easier to embed in a C program, but your experience may differ. In particular, I could provide all of `nw_luaed`'s functionality without writing a single line of Lua code, whereas `nw_sched` relies on a few dozen lines of embedded Scheme code that have to be parsed and interpreted on each run. But that may very possibly just reflect my poor Scheme/Guile skills. Furthermore, I can apparently run `nw_luaed` through Valgrind's (`http://www.valgrind.org`) memcheck utility without problems (I do

faster and easier to use. For this reason, I will probably not develop `nw_sched` much more, but I won't drop it altogether either, not soon at any rate.

1.18.2 The General Idea

`nw_ed`, `nw_sched`, and `nw_luaed` work in the same way: they iterate over the nodes in a specific order (Newick order by default), and for each node they evaluate a logical expression provided by the user. If the expression is true, they perform a user-specified action. By default, the (possibly modified) tree is printed at the end of the run.

Where the programs differ is the way the expression and action are expressed; as well as the expressive power of the languages used; some of them can also perform actions at the start or end of the run, or of each input tree. These are summarized in the table below.

	`nw_ed`	`nw_sched`	`nw_luaed`
language	own	Scheme	Lua
programming constructs	no	Scheme's	Lua's
functions	fixed	arbitrary[a]	arbitrary[a]
depends on	nothing	GNU Guile	Lua library
pre- & post-tree code	no	yes	yes
pre- & post-run code	no	yes	yes

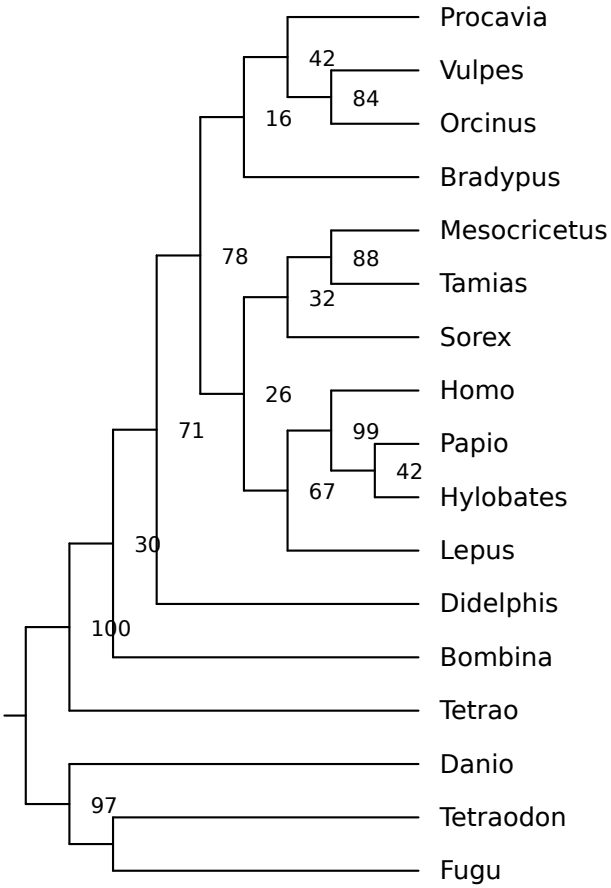
[a] *i.e.*, user can define their own

1.18.3 `nw_luaed`

Although `nw_luaed` is the most recent of the three, we'll cover it first because if this one does what you need it's unlikely you'll need the others. 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
```

this with all the programs in the utils), but with `nw_sched` I get tons of error messages. But it may be that I don't get how to manage memory with Guile



Let's extract all well-supported clades, using a support value of 95% or more as the criterion for being well-supported. In our jargon, the *condition* would be that a node i) have a support value in the first place (some nodes don't, *e.g.* the root and the LCA of (Fugu,Tetraodon)), and ii) that this value be no less than 95. The *action* would simply be to print out the tree rooted at the current node.

```
$ nw_luaed -n vrt2cg.nw 'b ~= nil and b >= 95' 's()' \
| nw_display -w 65 -
```

```
nw_luaed: error while loading shared libraries: libnutils.so: cannot open shared
 object file: No such file or directory
nw_display: error while loading shared libraries: libnutils.so: cannot open shar
ed object file: No such file or directory
```

Note that the `(Papio,(Hylobates,Homo))` clade appears twice – once on its own, and once within a larger clades (the tetrapods). This is because both clades meet the condition – the first has support value 99, the second has 100. There is a way (see Examples below) of only showing non-overlapping clades, which results in the "deepest" of two overlapping clades to be printed.
As always, the first argument to the program is a tree file, `vrt2cg.nw` in this example.

The second argument, `b ~= nil and b >= 95`, is the condition. In this case, it is just the conjunction (`and`) of two expressions `b ~= nil` and `b >= 95`. The former checks that the node's support value (variable `b`) is defined (*i.e.*, not `nil`); the latter checks that the support value is no less than 95. Note that the checks occur in that order, and that if `b` isn't defined, the second check isn't even performed, as it is meaningless.

name	(Lua) type	meaning (refers to the current node)
a	integer	number of ancestors
b	number	support value (or nil)
c	integer	number of children (direct descendants)
D	integer	total number of descendants (includes children)
d	number	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	number	parent edge length
N	node	the current node itself
r	Boolean	true iff node is the root

Table 1.1: Predefined variables in nw_luaed. 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.

The third argument, s(), is the action: it specifies what to do when a node meets the condition – in this case, call function s, which just prints the tree rooted at the current node.

Conditions

Conditions are Boolean expressions usually involving node properties which are available as predefined variables. As the program "visits" each node in turn, the variables are set to the current node's properties. These predefined variables have short names, to keep expressions concise. They are shown in table **??**.

The condition being just a Boolean expression written in Lua, all the logical operators of the Lua language can be used (indeed, any valid Lua snippet can be used, provided it evaluates to a Boolean), and you can use parentheses to override operator precedence or for clarity.
Here are some examples of nw_luaed conditions:

expression	selects:
l (lowercase ell)	all leaves
l and a <= 3	leaves with 3 ancestors or less
i and (b ~= nil) and (b >= 95)	internal nodes with support $\geq$ 95%
i and (b ~= nil) and (b < 50)	unsupported nodes (less than 50%)
not r	all nodes except the root
c > 2	multifurcating nodes

Notes:

- If it is certain that all nodes do have support, checks such as b ~= nil can be omitted.
- if an action must be performed on every node, just pass true as the condition.

Actions

Actions are arbitrary Lua expressions. These will typically involve printing out data or altering node properties or even tree structure. nw_luaed predefines a few functions

code	effect	modifies tree?
o	splice out the node	yes
s	print the subtree rooted at the node	no
u	delete ("unlink") the node (and all descendants)	yes

Table 1.2: Predefined actions in `nw_luaed`. The names are one letter long for convenience when passing the action on the command line. When called without an argument, these functions operate on the current node (*i.e.*, `s()` is the same as `s(N)` (where `N` means the current node – see table **??**).

name	type	mode	meaning
`len`, `L`	number	rw	parent edge's length
`lbl`	string	rw	label
`b`	number	ro	support value
`par`	node	ro	parent
`first_child`, `fc`	node	ro	first child
`last_child`, `lc`	node	ro	last child
`children_count`, `c`	integer	ro	number of children
`kids`	table	ro	list of children nodes

Table 1.3: Node properties accessible from Lua. rw: read-write, ro: read only. Some fields have both a short and a long name, the former is intended for use on the command line (where space is at a premium), the latter is for use in scripts (but both can be used anywhere). Note that when referring to the *current* node, the predefined variables (see table **??**) are even more concise, *e.g.* `N.len` or `N.L` can be written just `L`, but they are read-only.

for such purposes (table **??**), and you can of course write your own (unless the function is very short, this is easier if you pass the Lua code in a file, see **??**).

`nw_sched` defines a "node" type, and the current node is always accessible as variable `N` (other nodes can be obtained through node properties, see below). Node properties can be accessed as fields in a Lua table. Table **??** lists the available node fields. So for example the parent of the current node is expressed by `N.par`; doubling its length could be `N.par.len = N.par.len * 2`.

Lua script in a file

Sometimes the command line is too short to comfortably write the condition and action. In this case, one can put the Lua code in a file, which is passed to `nw_luaed` via option `-f`. The file can contain any Lua code, but some function names are special: they function as "hooks", that is, they are called on predefined occasions. Table **??** shows the hook names and when they are called.

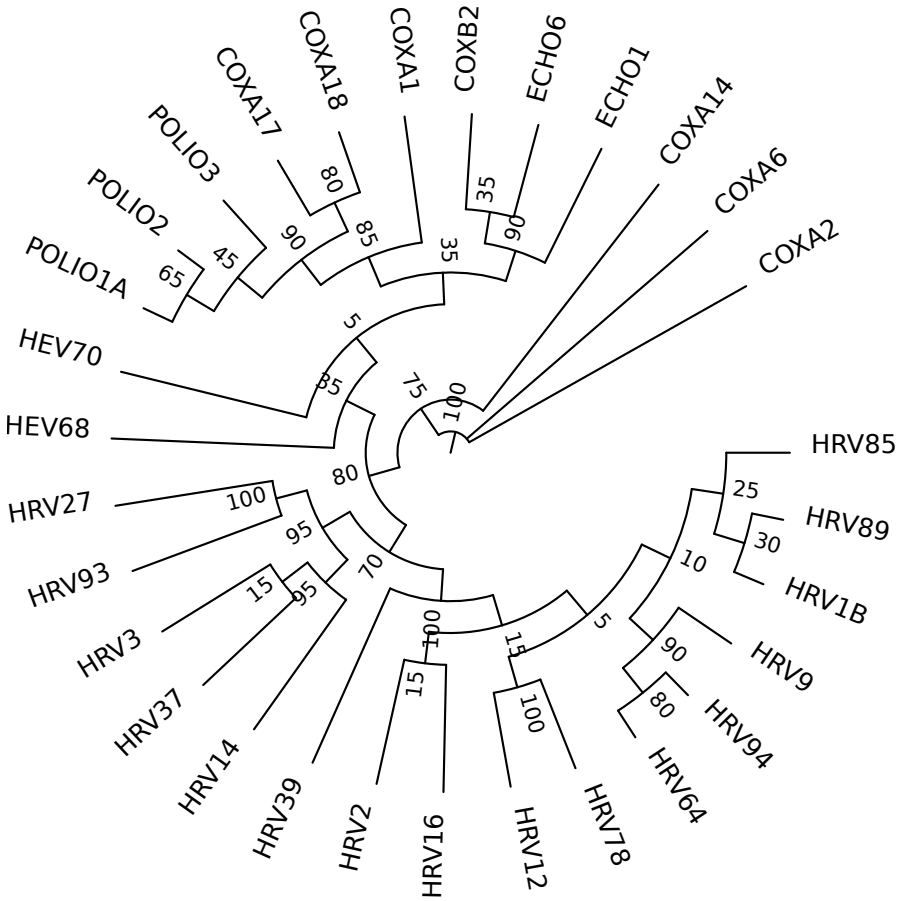
See "tree counter" and "numbering nodes" below for examples of use of `-f`.

Examples

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`:

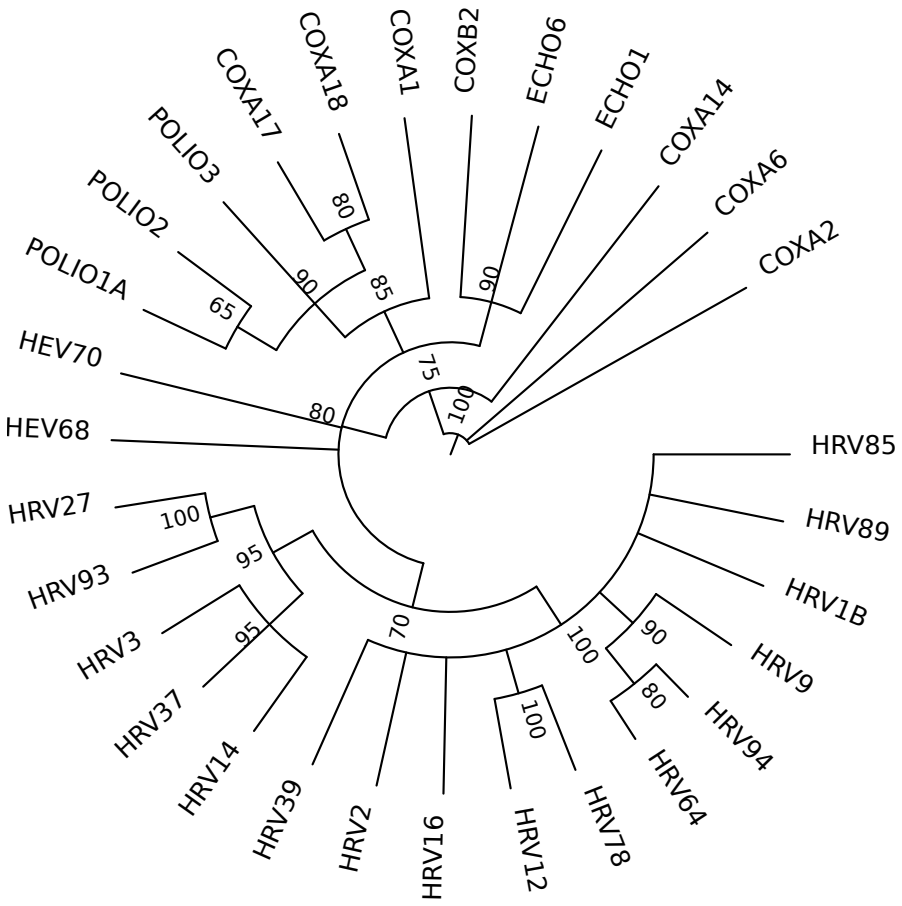
hook name	called…
`start_run`	before processing any tree
`start_tree`	for each tree, before processing
`node`	for each node
`stop_tree`	for each tree, after processing
`stop_run`	after processing all trees

Table 1.4: Hooks defined by `nw_luaed`. If a function named `start_tree` is defined, it will be called once per tree, before the tree is processed; etc. If a hook is not defined, no action is performed on the corresponding occasion. Strictly speaking, `start_run` is not really necessary, as the file is evaluated before the run anyway, but it seems cleaner to provide a start-of-the-run hook as well.



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 use `nw_luaed`'s `o` function to "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_luaed HRV_cg.nw 'i and 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.

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.[18] To format[19] the lengths to two decimal places, do the following:

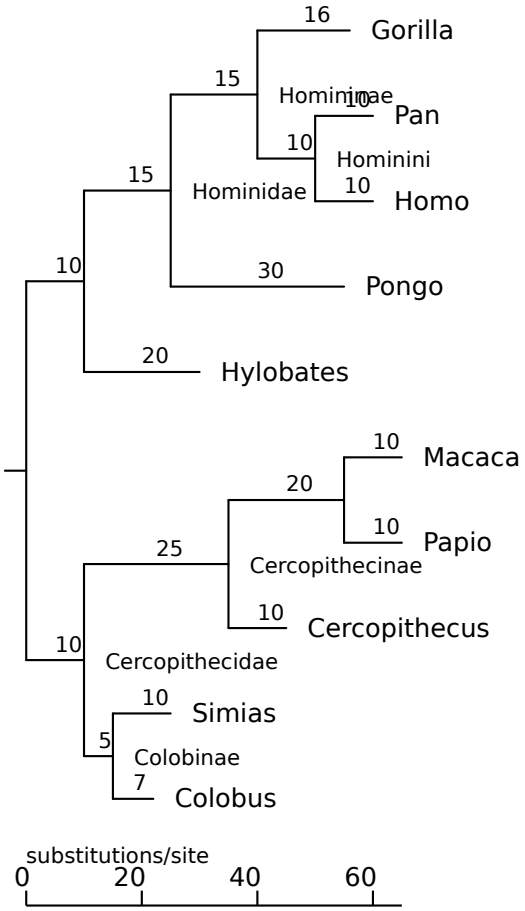
```
$ nw_luaed HRV.nw 'L ~= nil' 'N.len = string.format("%.2f",L)' \
| nw_indent - | tail

nw_luaed: error while loading shared libraries: libnutils.so: cannot open shared
 object file: No such file or directory
```

Multiplying lengths by a constant 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
```

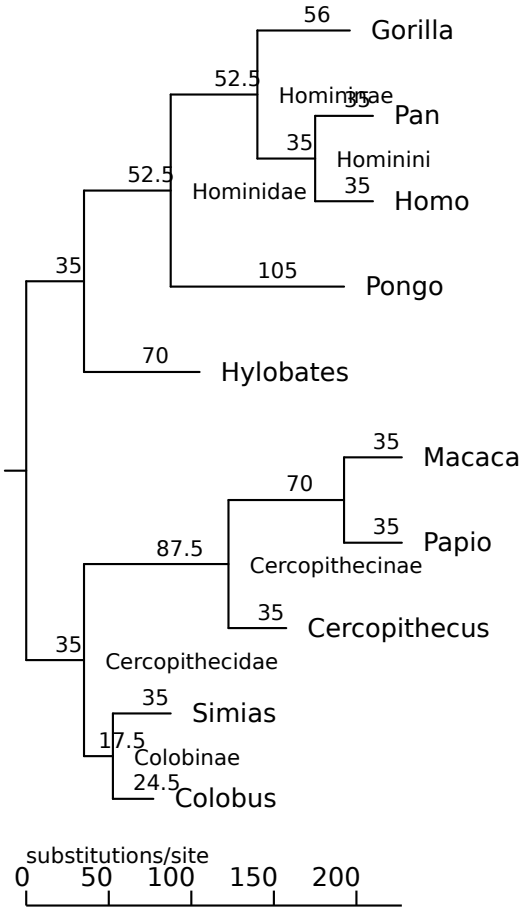
[18] the *first* ten lines contain only opening parentheses.

[19] `nw_sched` automatically loads the `format` module so that the full-fledged `format` function is available.



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

```
$ nw_luaed catarrhini 'not r' 'N.len = 3.5 * N.len' | nw_display -s -
```



Implementing other Newick Utilities `nw_luaed` 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_luaed` rather than the original, since `nw_luaed` will be slower (after all, it has to start the Lua interpreter, parse the Lua expressions, etc.). But these "equivalents" can serve as illustration.

program	`nw_luaed` equivalent
`nw_labels`	`'lbl ~= "" ' 'print(lbl)'`
`nw_labels -I`	`'l and lbl ~= "" ' 'print(lbl)'`
`nw_topology`	`true 'N.len = "" '`
`nw_topology -I`	`true 'N.len = ""; if not l then N.lbl = "" end'`

The `lbl ~= ""` condition in the `nw_labels` replacements is checked because the original `nw_labels` does not print empty labels. In the `nw_topology` replacement, the check for node type (`l`) is done in the action rather than the condition, because there is some code that is performed for every node and some additional code only for non-leaves.

A tree counter As you know by now, the Newick Utilities are able to process files that contain any number of trees. But just how many trees are there in a file? If you're certain that there is exactly one tree per line, you just use `wc -l`. But the Newick format allows trees to span more than one line, or conversely there may be more than

one tree per line; moreover there may be blank lines. All these conspire to yield wrong tree counts. To solve this, we write a tree counter in Lua, and pass it to `nw_luaed`. Here is the counter:

```
$ cat count_trees.lua
```

```
function start_run()
        count = 0
end

function stop_tree()
        count = count + 1
end

function stop_run()
        print(count)
end
```

As you can see, I've defined three of the five possible hooks. Before any tree is processed, `start_run` is called, which defines variable `count` and initializes it to zero. After each tree is processed (actually, no processing is done, since the `node` hook is not defined), function `stop_tree` is called, which increments the counter. And after the last tree has been processed, the `stop_run` hook is called, which just prints out the count.

Here it is in action. First, the simple case of one tree per line:

```
$ wc -l forest
```

```
4 forest
```

```
$ nw_luaed -n -f count_trees.lua forest
```

```
nw_luaed: error while loading shared libraries: libnutils.so: cannot open shared
 object file: No such file or directory
```

Right. Now how about this one: these are the same trees as in `forest`, but all on a single line:

```
$ wc -l jungle
```

```
1 jungle
```

```
$ nw_luaed -n -f count_trees.lua jungle
```

```
nw_luaed: error while loading shared libraries: libnutils.so: cannot open shared
 object file: No such file or directory
```

`nw_luaed` is not fooled! And this is the opposite case – an indented tree, which has one *node* per line:

```
$ nw_indent catarrhini | wc -l
```

```
31
```

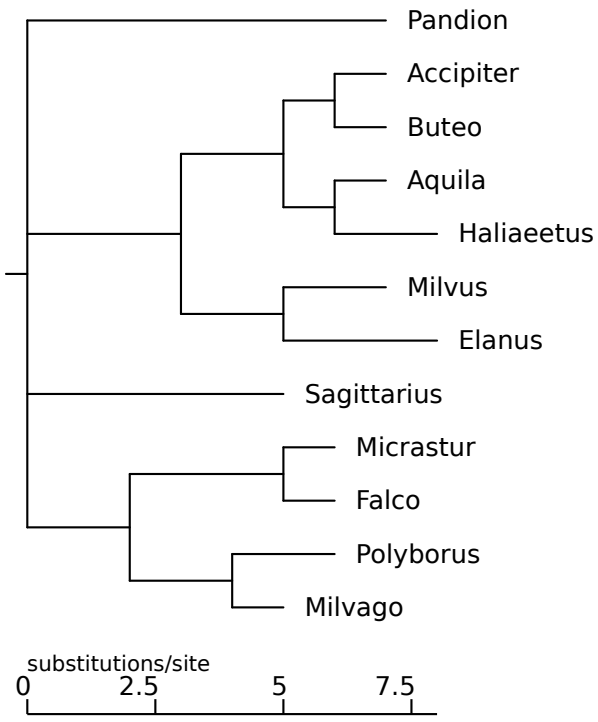
```
$ nw_indent catarrhini | nw_luaed -n -f count_trees.lua -
```

```
nw_luaed: error while loading shared libraries: libnutils.so: cannot open shared
 object file: No such file or directory
```

There's no confusing our tree counter, it seems. Note that in future versions I might well make this unnecessary by supplying a predefined variable which counts the input trees, akin to Awk's `NR`.

Numbering inner nodes I was once handed a tree with the task of numbering the inner nodes, starting close to the leaves and ending at the root.[20] Here is a tree with unlabeled inner nodes (I hide the branch lengths lest they obscure the inner node labels, which will also be numeric):

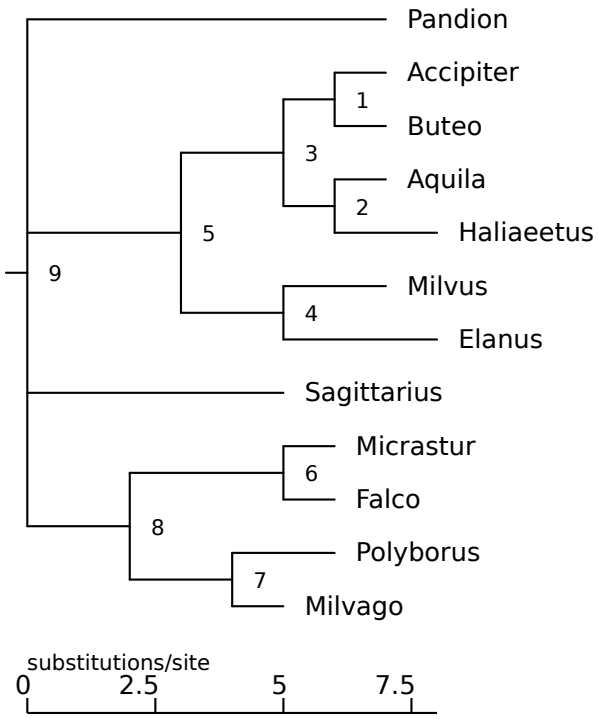
```
$ nw_display -s -v 25 -b 'opacity:0' falconiformes
```



A solution is the following `nw_luaed` script:

```
$ cat number_inodes.lua

-- Benoit's problem: number all inner nodes.

function start_tree() n = 0 end

function node()
        if not l then n = n + 1; N.lbl = n end
end

$ nw_luaed -f number_inodes.lua falconiformes \
| nw_display -s -v 25 -b 'opacity:0' -
```

[20] Thanks to Benoît Defaucheux for this example.



Extracting deep, well-supported clades In the first example of this section (**??**), we extracted well-supported clades, but there was an overlap because one well-supported clade was a subclade of another. We may want to extract only the "deepest" clades that meet the condition, in other words, once a node has been found to match, its descendants should not be processed. This is the purpose of option `-o`. For this option to be useful, though, the tree must be processed from the root to the leaves, which is the opposite of the default (namely, Newick order). To override this, we pass option `-r` ("reverse"):

```
$ nw_luaed -orn vrt2cg.nw 'b ~= nil and b >= 95' 's()' \
| nw_display -w 65 -

nw_luaed: error while loading shared libraries: libnutils.so: cannot open shared
 object file: No such file or directory
nw_display: error while loading shared libraries: libnutils.so: cannot open shar
ed object file: No such file or directory
```

All overlap has now disappeared: the `(Papio,(Hylobates,Homo))` clade is no longer printed on its own.

Future

I intend to develop `nw_luaed` further. Among the items in my TODO list are a few new predefined variables (number of records, root of the tree, more powerful structure-altering functions, etc).

1.18.4 nw_ed

Note: it is likely that `nw_luaed` (**??**) will be more useful than `nw_ed`. See also section **??** for a general intro to the stream editing programs. This section gives a minimal description of `nw_ed`, without

The two parameters of `nw_ed` (besides the input file) are the condition and the action.

Conditions

Conditions are logical expressions involving node properties, 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. 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

I have no plans to implement any other actions, as this can be done easily with nw_luaed (or nw_sched).

1.18.5 nw_sched

Note: it is likely that nw_luaed (**??**) will be more convenient than nw_sched. See also section **??** for a general intro to the stream editing programs. This section gives a minimal description of nw_sched, with no motivation and only a few examples (see **??** for more).
As mentioned above, nw_sched works like nw_luaed, but uses Scheme instead of Lua. Accordingly, the condition and action are passed as a Scheme expression. The Scheme language 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_luaed.

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

nw_display: error while loading shared libraries: libnutils.so: cannot open shar
ed object file: No such file or directory
```

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 (`&`)[21] of the expressions `(def? 'b)` and `(>= 95 b)`. The former checks that variable `b` (bootstrap support) is defined[22], 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 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.

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

[22] 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?`.

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[23]:

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

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

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.

Future

I do not plan to develop `nw_sched` any more, because in my opinion `nw_luaed` is better. I will probably drop it eventually, but not immediately.