`Neighbors`: Finding the Genomes of Targets and Neighbors

Bernhard Haubold

January 25, 2024

Contents

Chapter 1

Introduction

Diagnostic PCR-markers are designed to amplify all members of a target set of organisms and nothing else. A promising approach to ensure marker specificity is to compare the target genomes to the genomes of the closest distinct relatives, the neighbors. This usually removes the vast majority of non-specific material. The small remainder can then be further tested by *in silico* PCR against, say, the non-redundant collection of nucleotide sequences, `nt`. The published program `fur` implements this comparison between targets and neighbors. Markers constructed from its output can have excellent specificity and sensifivity [2].

Users of programs like `fur` need to know the neighbors of their targets. But how can neighbors be discovered, if they aren't already known? To answer this question, consider the toy taxonomy in Figure 1.1, where the numbers are taxon-IDs that are linked to genome accessions. Let taxa 7 and 4 be our targets. Their most recent common ancestor is 3. This implies there are three additional targets, 3, 5, and 6. The neighbors are the nodes in the subtree rooted on 3's parent, minus the parent and minus the targets. So in our example there are five neighbors, 2, 8, 9, 10, and 11. Notice that we look up all nodes in a subtree, not just the leaves, as genome sequences might be associated with taxa in terminal and internal nodes.

To put this a bit more formally, let m be the most recent common ancestor of the targets. Their neighbors, $\mathcal{N}$, are then computed by subtracting the nodes in m's subtree from the nodes in its parent's subtree, minus the parent

$$\mathcal{N} = s(p(m)) - s(m) - p(m), \tag{1.1}$$

where $s(v)$ returns the nodes in the subtree rooted on v, and $p(v)$ the parent of v. This set subtraction is implemented in the program `neighbors`.

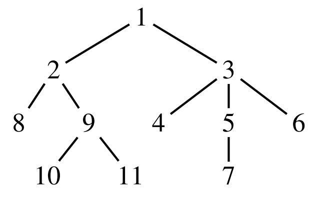

Figure 1.1: Toy taxonomy.

Table 1.1: The ten programs of Neighbors

#	Name	Based on	Function
1	`ants`	taxonomy	list ancestors
2	`climt`	phylogeny	climb tree
3	`dree`	taxonomy	draw tree
4	`fintac`	phylogeny	find target clade
5	`land`	phylogeny	label nodes
6	`makeNeiDb`	taxonomy	make neighbors database
7	`neighbors`	taxonomy	find neighbors (and targets)
8	`outliers`	numbers	find outliers
9	`pickle`	phylogeny	pick clades
10	`taxi`	taxonomy	get taxon-ID for taxon name

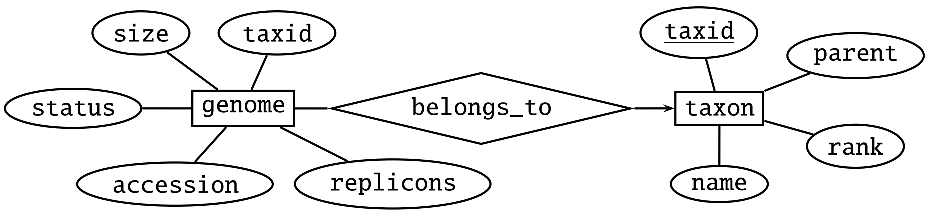

Figure 1.2: Diagram of `neidb`.

The Neighbors package consists of the ten programs listed in Table 1.1. Five of these programs are based on the taxonomy, four on a phylogeny, and one, `outliers`, on numerical data.

The taxonomy is presented as an `sqlite` database, let's call it `neidb`, which is built using the program `makeNeiDb`. As shown in Figure 1.2, `neidb` consists of two tables, `genome` and `taxon`. Each genome comes from an organism identified by its taxon-ID, has a size, a status, a genome accession, and one or more replicons, which are lumped in one attribute. Each genome belongs to a taxon. A taxon has a unique taxon-ID, which serves as primary key. A taxon also has a parent, a rank, and a name. Since several genomes may have been sequenced for a given taxon, the attribute `taxid` in table `genome` is not unique, which is why that table has no primary key.

Once the database is constructed, we can query it. The tutorial shows how to do for *E. coli* O157:H7. It is a notorious food-borne pathogen that can cause severe diarrhea in humans. The program `taxi` gives us the taxon-ID for *E. coli* O157:H7. It is often useful to place this taxon-ID into context, for which we have two programs, `dree` and `andi`. Starting from a taxon-ID, `dree` draws the taxonomic tree rooted on that taxon-ID. Instead of walking from a root towards the leaves, the program `ants` starts from a taxon and walks in the opposite direction toward the universal root. Along this path, `ants` lists all ancestral taxa of our focal taxon. Once we've got our bearings in the taxonomy, we can query it with `neighbors` to get the complete set of target and neighbor genomes currently available.

The genomes returned by `neighbors` form the raw material for marker discovery. However, the classification into targets and neighbors retrieved from the taxonomy often contradicts the phylogeny calculated from the target and neighbor genomes. So we should always compute a phylogeny from our target and neighbor genomes. A program for doing this efficiently is `phylonium` [3].

The phylogenies of targets and neighbors may comprise hundreds of taxa. To help analyze such large phylogenies, Neighbors contains `land` for labeling nodes, `pickle` for picking nodes, `fintac` for finding the target clade, and `climt` for climbing the tree.

Even a clean set of phylogenetic targets might still contain genomes that are outliers in some way, for example with respect to their genome lengths. The program `outliers` helps find such outliers. The tutorial contains the details.

Chapter 2

Programs

2.1 `makeNeiDb`

Neigbor genomes are discovered by querying a database that combines taxonomy and genome information. The program `makeNeiDb` constructs this database. The outline of `makeNeiDb` contains hooks for imports, types, and the logic of the main function.

Prog. 2.1 (`makeNeiDb`)

5a ⟨*makeNeiDb.go* 5a⟩≡

```
package main

import (
            ⟨Imports, Pr. 2.1 6b⟩
)
func main() {
            ⟨Main function, Pr. 2.1 5b⟩
}
```

In the main function, we declare the options, set the usage, parse the options, and construct the database.

5b ⟨*Main function, Pr. 2.1* 5b⟩≡ (5a)

```
⟨Declare options, Pr. 2.1 6a⟩
⟨Set usage, Pr. 2.1 6c⟩
⟨Parse options, Pr. 2.1 6e⟩
⟨Construct database, Pr. 2.1 6g⟩
```

The program takes as input three files of genome information and two files of taxonomy information. Hence we declare five options for these five files. In addition, we declare `-d` to set the database name and `-v` for printing the version.

6a ⟨*Declare options, Pr. 2.1* 6a⟩≡ (5b)
```
var optP = flag.String("p", "prokaryotes.txt", "prokaryote genomes")
var optE = flag.String("e", "eukaryotes.txt", "eukaryote genomes")
var optI = flag.String("i", "viruses.txt", "virus genomes")
var optA = flag.String("a", "names.dmp", "taxonomic names")
var optO = flag.String("o", "nodes.dmp", "node information")
var optD = flag.String("d", "neidb", "database name")
var optV = flag.Bool("v", false, "print version & " +
        "program information")
```

We import `flag`.

6b ⟨*Imports, Pr. 2.1* 6b⟩≡ (5a) 6d ▷
```
"flag"
```

The usage consists of three statements. The actual usage statement, an explanation of the program's purpose, and an example command. In the explanation we cite the data sources.

6c ⟨*Set usage, Pr. 2.1* 6c⟩≡ (5b)
```
u := "makeNeiDb [option]..."
p := "Construct a taxonomy database for discovering neighbor genomes." +
        "\n\tGenomes:  <ftp>/genomes/GENOME_REPORTS/" +
        "((pro|eu)karyotes|viruses).txt" +
        "\n\tTaxonomy: <ftp>/pub/taxonomy/taxdump.tar.gz" +
        "\n\t<ftp>=ftp.ncbi.nlm.nih.gov"
e := "makeNeiDb -d myNew.db"
clio.Usage(u, p, e)
```

We import `clio`.

6d ⟨*Imports, Pr. 2.1* 6b⟩+≡ (5a) ◁ 6b 6f ▷
```
"github.com/evolbioinf/clio"
```

We parse the options and respond to `-v`.

6e ⟨*Parse options, Pr. 2.1* 6e⟩≡ (5b)
```
flag.Parse()
if *optV {
        util.PrintInfo("makeNeiDb")
}
```

We import `util`.

6f ⟨*Imports, Pr. 2.1* 6b⟩+≡ (5a) ◁ 6d 6h ▷
```
"github.com/evolbioinf/neighbors/util"
```

The database construction is delegated to `NewTaxonomyDB`.

6g ⟨*Construct database, Pr. 2.1* 6g⟩≡ (5b)
```
tdb.NewTaxonomyDB(*optO, *optA, *optP, *optE, *optI, *optD)
```

We import `tdb`.

6h ⟨*Imports, Pr. 2.1* 6b⟩+≡ (5a) ◁ 6f
```
"github.com/evolbioinf/neighbors/tdb"
```

We are done with `makeNeiDb`, time to test it.

Testing

The outline for testing `makeNeiDb` has hooks for imports and the testing logic.

7a ⟨*makeNeiDb_test.go* 7a⟩≡
```
package main

import (
          "testing"
          ⟨Testing imports, Pr. 2.1 7c⟩
)

func TestMakeNeiDb(t *testing.T) {
          ⟨Testing, Pr. 2.1 7b⟩
}
```

We construct a set of tests and them in a loop.

7b ⟨*Testing, Pr. 2.1* 7b⟩≡ (7a)
```
var tests []*exec.Cmd
⟨Construct tests, Pr. 2.1 7d⟩
for i, test := range tests {
          ⟨Run test, Pr. 2.1 8a⟩
}
```

We import `exec`.

7c ⟨*Testing imports, Pr. 2.1* 7c⟩≡ (7a) 8b ▷
```
"os/exec"
```

We construct a small database.

7d ⟨*Construct tests, Pr. 2.1* 7d⟩≡ (7b) 7e ▷
```
test := exec.Command("./makeNeiDb",
          "-a", "../data/namesTest.dmp",
          "-d", "test.db",
          "-e", "../data/eukaryotes.txt",
          "-i", "../data/viruses.txt",
          "-o", "../data/nodesTest.dmp",
          "-p", "../data/prokaryotes.txt")
tests = append(tests, test)
```

Now we query the new database.

7e ⟨*Construct tests, Pr. 2.1* 7d⟩+≡ (7b) ◁ 7d
```
test = exec.Command("/usr/bin/sqlite3",
          "test.db",
          "select * from taxon order by taxid")
tests = append(tests, test)
```

We run the test and compare the result we get with the result we want. The results we want are contained in files `r1.txt` and `r2.txt`.

8a ⟨*Run test, Pr. 2.1* 8a⟩≡ (7b)

```
get, err := test.Output()
if err != nil {
          t.Errorf("couldn't run %q", test)
}
f := "r" + strconv.Itoa(i + 1) + ".txt"
want, err := ioutil.ReadFile(f)
if err != nil {
          t.Errorf("couldn't open %q", f)
}
if !bytes.Equal(get, want) {
          t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

8b ⟨*Testing imports, Pr. 2.1* 7c⟩+≡ (7a) ◁7c

```
"strconv"
"io/ioutil"
"bytes"
```

2.2 taxi

The `neighbors` module is based on taxon-IDs. These are difficult to remember, so the program `taxi` takes the user from the name of an organism to its taxon-ID. The input to `taxi` is a scientific name and the taxonomy database. By default, `taxi` carries out a case-insensitive exact match across the full name, but the user can opt for a substring match instead. Regardless of the query type, there may be more than one match, and `taxi` returns them all.

Implementation

The program `taxi` has hooks for imports and the logic of the main function.

Prog. 2.2 (taxi)

9a ⟨*taxi.go* 9a⟩≡
```
package main

import (
            ⟨Imports, Pr. 2.2 9d⟩
)

func main() {
            ⟨Main function, Pr. 2.2 9b⟩
}
```

In the `main` function we set the usage, declare the options, parse the options, get the taxon-IDs, and print them.

9b ⟨*Main function, Pr. 2.2* 9b⟩≡ (9a)
```
⟨Set usage, Pr. 2.2 9c⟩
⟨Declare options, Pr. 2.2 9e⟩
⟨Parse options, Pr. 2.2 10b⟩
⟨Get taxon-IDs, Pr. 2.2 10f⟩
⟨Print taxon-IDs, Pr. 2.2 11a⟩
```

The usage consists of three parts: The actual usage message, an explanation of the purpose of `taxi`, and an example command.

9c ⟨*Set usage, Pr. 2.2* 9c⟩≡ (9b)
```
u := "taxi [option] <scientific-name> <db>"
p := "Take user from scientific name to taxon-ID."
e := "taxi \"homo sapiens\" neidb"
clio.Usage(u, p, e)
```

We import `clio`.

9d ⟨*Imports, Pr. 2.2* 9d⟩≡ (9a) 10a ▷
```
"github.com/evolbioinf/clio"
```

Apart from the version, we declare the substring option.

9e ⟨*Declare options, Pr. 2.2* 9e⟩≡ (9b)
```
var optV = flag.Bool("v", false, "version")
var optS = flag.Bool("s", false, "substring match")
```

We import flag.

10a ⟨*Imports, Pr. 2.2* 9d⟩+≡ (9a) ◁9d 10c▷
```
"flag"
```

We parse the options and respond to the version option (-v) first, as this would stop the program. Then we get the remaining arguments and check there are two of them, the names of the taxon and the database. We store both names, and if the user opted for substring matching, we bookend the taxon name with percent characters.

10b ⟨*Parse options, Pr. 2.2* 10b⟩≡ (9b)
```
flag.Parse()
if *optV { util.PrintInfo("taxi") }
args := flag.Args()
⟨Check arguments, Pr. 2.2 10d⟩
name := args[0]
db := args[1]
if *optS {
        name = fmt.Sprintf("%%%s%%", name)
}
```

We import util and fmt.

10c ⟨*Imports, Pr. 2.2* 9d⟩+≡ (9a) ◁10a 10e▷
```
"github.com/evolbioinf/neighbors/util"
"fmt"
```

If the user didn't provide a taxon and a database, we send a friendly message and quit.

10d ⟨*Check arguments, Pr. 2.2* 10d⟩≡ (10b)
```
m := "please provide a taxon and a database"
if len(args) != 2 {
        fmt.Fprintf(os.Stderr, "%s\n", m)
        os.Exit(-1)
}
```

We import os.

10e ⟨*Imports, Pr. 2.2* 9d⟩+≡ (9a) ◁10c 10g▷
```
"os"
```

We get the taxon-IDs through a method call on the taxonomy database. If we don't find any taxa, we're done.

10f ⟨*Get taxon-IDs, Pr. 2.2* 10f⟩≡ (9b)
```
taxdb := tdb.OpenTaxonomyDB(db)
taxa := taxdb.Taxids(name)
if len(taxa) == 0 {
        return
}
```

We import tdb.

10g ⟨*Imports, Pr. 2.2* 9d⟩+≡ (9a) ◁10e 11b▷
```
"github.com/evolbioinf/neighbors/tdb"
```

We print the taxon-IDs, the parents' IDs, and the corresponding names in a table. We layout the table using a `tabwriter`.

11a ⟨*Print taxon-IDs, Pr. 2.2* 11a⟩≡ (9b)
```
w := tabwriter.NewWriter(os.Stdout, 0, 1, 2, ' ', 0)
defer w.Flush()
fmt.Fprintf(w, "# ID\tParent\tName\n")
for _, taxon := range taxa {
        name := taxdb.Name(taxon)
        p := taxdb.Parent(taxon)
        fmt.Fprintf(w, "  %d\t%d\t%s\n", taxon, p, name)
}
```

We import `tabwriter`.

11b ⟨*Imports, Pr. 2.2* 9d⟩+≡ (9a) ◁ 10g
```
"text/tabwriter"
```

We're done writing `taxi`, time to test it.

Testing

Our testing code for `taxi` contains hooks for imports and the testing logic.

11c ⟨*taxi_test.go* 11c⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Pr. 2.2 11e⟩
)

func TestTaxi(t *testing.T) {
        ⟨Testing, Pr. 2.2 11d⟩
}
```

We construct the tests and iterate over them.

11d ⟨*Testing, Pr. 2.2* 11d⟩≡ (11c)
```
var tests []*exec.Cmd
⟨Construct tests, Pr. 2.2 12a⟩
for i, test := range tests {
        ⟨Run test, Pr. 2.2 12b⟩
}
```

We import `exec`.

11e ⟨*Testing imports, Pr. 2.2* 11e⟩≡ (11c) 12c ▷
```
"os/exec"
```

Our query is "homo sapiens", which we run in default mode and in substring mode.

12a ⟨*Construct tests, Pr. 2.2* 12a⟩≡ (11d)
```
q := "homo sapiens"
d := "../data/neidb"
test := exec.Command("./taxi", q, d)
tests = append(tests, test)
test = exec.Command("./taxi", "-s", q, d)
tests = append(tests, test)
```

We execute the test and compare the result we get to the result we want. The results we want are contained in the files `r1.txt` and `r2.txt`.

12b ⟨*Run test, Pr. 2.2* 12b⟩≡ (11d)
```
get, err := test.Output()
if err != nil { t.Error(err) }
f := "r" + strconv.Itoa(i + 1) + ".txt"
want, err := ioutil.ReadFile(f)
if err != nil { t.Error(err) }
if !bytes.Equal(get, want) {
          t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

12c ⟨*Testing imports, Pr. 2.2* 11e⟩+≡ (11c) ◁11e
```
"strconv"
"io/ioutil"
"bytes"
```

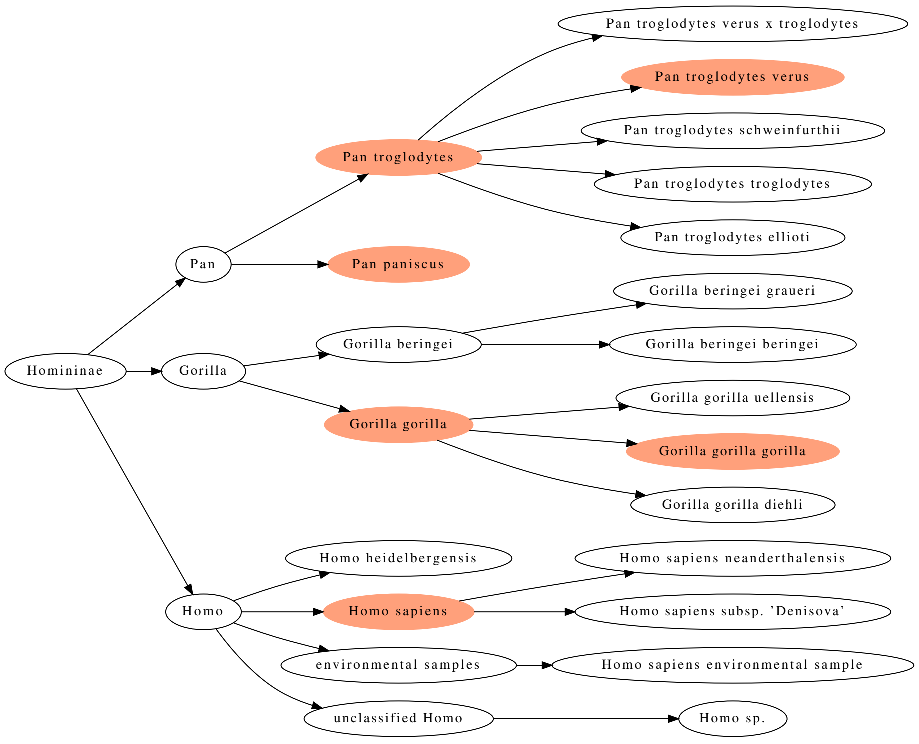

Figure 2.1: Taxonomy of the *Homininae*, taxa with sequenced genomes are colored.

2.3 `dree`

The program `dree` takes as input a taxon-ID and the name of the taxonomy database. It returns the subtree rooted on the focal taxon. The subtree is written in the dot language ready for rendering with `dot`, which is part of the graphviz package. For example, we can draw the tree for the *Homininae*, taxon-ID 207598, which include human, chimp, and gorilla. By default, `dree` labels taxa with their IDs, but with `-n` we get names instead. Taxa with sequenced genomes are colored.

```
$ ./dree -n 207598 neidb | dot -T x11
```

The resulting tree in Figure 2.1 is pretty crowded, so we use `-g` to to reduce it to the taxa with sequenced genomes and their ancestors.

```
$ ./dree -n -g 207598 neidb | dot -T x11
```

This gives the more legible Figure 2.2.

Implementation

The outline of `dree` has hooks for imports and the logic of the main function.

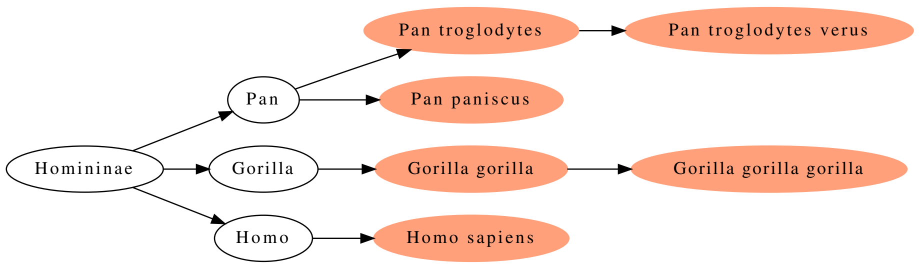

Figure 2.2: Taxonomy of the *Homininae* reduced to taxa with sequenced genomes and their ancestors; taxa with genomes are colored.

Prog. 2.3 (`dree`)

14a ⟨*dree.go* 14a⟩≡

```
package main

import (
              ⟨Imports, Pr. 2.3 14d⟩
)

func main() {
              ⟨Main function, Pr. 2.3 14b⟩
}
```

In the main function we set the usage, declare the options, and parse them. Then we get the subtree we're after, annotate it, and print it.

14b ⟨*Main function, Pr. 2.3* 14b⟩≡ (14a)

```
⟨Set usage, Pr. 2.3 14c⟩
⟨Declare options, Pr. 2.3 15a⟩
⟨Parse options, Pr. 2.3 15c⟩
⟨Get subtree, Pr. 2.3 16d⟩
⟨Annotate subtree, Pr. 2.3 16e⟩
⟨Draw subtree, Pr. 2.3 17a⟩
```

The usage consists of the actual usage message, an explanation of the purpose of `dree`, and an example command.

14c ⟨*Set usage, Pr. 2.3* 14c⟩≡ (14b)

```
u := "dree [-h] [option]... <taxon-ID> <db>"
p := "Get the taxonomy rooted on a specific taxon."
e := "dree -n -g 207598 neidb | dot -T x11"
clio.Usage(u, p, e)
```

We import `clio`.

14d ⟨*Imports, Pr. 2.3* 14d⟩≡ (14a) 15b ▷

```
"github.com/evolbioinf/clio"
```

Apart from the version, we declare two options, one to print the name instead of the default taxon-ID, `-n`, the other to print only taxa with genome sequences, `-g`.

15a ⟨*Declare options, Pr. 2.3* 15a⟩≡ (14b)
```
optV := flag.Bool("v", false, "version")
optN := flag.Bool("n", false,
          "print names instead of taxon-IDs")
optG := flag.Bool("g", false,
          "only taxa with genome sequences")
optL := flag.Bool("l", false, "list taxa")
```

We import `flag`.

15b ⟨*Imports, Pr. 2.3* 14d⟩+≡ (14a) ◁14d 15d▷
```
"flag"
```

We parse the options and respond to `-v`, as this stops `dree`. Then we get the taxon-ID and the database name. Using the database name we open the database connection.

15c ⟨*Parse options, Pr. 2.3* 15c⟩≡ (14b)
```
flag.Parse()
if *optV {
          util.PrintInfo("dree")
}
⟨Get taxon-ID, Pr. 2.3 15e⟩
⟨Get database name, Pr. 2.3 16a⟩
⟨Open database connection, Pr. 2.3 16b⟩
```

We import `util`.

15d ⟨*Imports, Pr. 2.3* 14d⟩+≡ (14a) ◁15b 15f▷
```
"github.com/evolbioinf/neighbors/util"
```

The remaining tokens on the command line are interpreted as taxon-ID and database, in that order. If we don't have two tokens, we bail with a friendly message. We convert the taxonomy-ID to an integer.

15e ⟨*Get taxon-ID, Pr. 2.3* 15e⟩≡ (15c)
```
tokens := flag.Args()
if len(tokens) != 2 {
          fmt.Fprintf(os.Stderr,
                "please provide a taxon-ID and a database\n")
          os.Exit(0)
}
tid, err := strconv.Atoi(tokens[0])
if err != nil {
          log.Fatalf("couldn't convert %q", tokens[0])
}
```

We import `fmt`, `os`, `strconv`, and `log`.

15f ⟨*Imports, Pr. 2.3* 14d⟩+≡ (14a) ◁15d 16c▷
```
"fmt"
"os"
"strconv"
"log"
```

The database name is the second token.

16a ⟨*Get database name, Pr. 2.3* 16a⟩≡ (15c)
```
dbname := tokens[1]
```

We open a connection to the taxonomy database.

16b ⟨*Open database connection, Pr. 2.3* 16b⟩≡ (15c)
```
taxdb := tdb.OpenTaxonomyDB(dbname)
```

We import `tdb`.

16c ⟨*Imports, Pr. 2.3* 14d⟩+≡ (14a) ◁ 15f 17c ▷
```
"github.com/evolbioinf/neighbors/tdb"
```

We get the subtree we're looking for.

16d ⟨*Get subtree, Pr. 2.3* 16d⟩≡ (14b)
```
subtree := taxdb.Subtree(tid)
```

We annotate each node, v, of our subtree with two kinds of information: whether or not v has at least one genome associated with it, and whether or not v has at least one genome in the subtree rooted on it. So we construct two maps for storing this information. Then we mark nodes with genomes and nodes with genomes in subtree.

16e ⟨*Annotate subtree, Pr. 2.3* 16e⟩≡ (14b)
```
hasGenome := make(map[int]bool)
hasGsub := make(map[int]bool)
⟨Mark nodes with genomes, Pr. 2.3 16f⟩
⟨Mark nodes with genomes in subtree, Pr. 2.3 16g⟩
```

We iterate over the nodes and mark those with genomes. These nodes are also the initial set of nodes with genomes in their subtree.

16f ⟨*Mark nodes with genomes, Pr. 2.3* 16f⟩≡ (16e)
```
for _, v := range subtree {
        if len(taxdb.Accessions(v)) > 0 {
                hasGenome[v] = true
                hasGsub[v] = true
        }
}
```

To find the nodes with genomes in their subtrees, we iterate over the nodes and for each node with a genome in it subtree, we propagate this up the tree.

16g ⟨*Mark nodes with genomes in subtree, Pr. 2.3* 16g⟩≡ (16e)
```
for _, v := range subtree {
        if hasGsub[v] {
                u := v
                p := taxdb.Parent(u)
                for u != tid {
                        hasGsub[p] = true
                        u = p
                        p = taxdb.Parent(u)
                }
        }
}
```

By default we draw the subtree graph, but the user might also have opted for listing the taxa in the subtree.

17a ⟨*Draw subtree, Pr. 2.3* 17a⟩≡ (14b)

```
if *optL {
        ⟨List subtree, Pr. 2.3 17b⟩
} else {
        ⟨Draw subtree graph, Pr. 2.3 18a⟩
}
```

We list the subtree in a table consisting of three mandatory columns, taxon-ID, rank, and the number of genomes. In addition, the user might have opted for names. We construct this table with a `tabwriter`. Then we iterate over the nodes and list each one. At the end we flush the tabwriter.

17b ⟨*List subtree, Pr. 2.3* 17b⟩≡ (17a)

```
w := tabwriter.NewWriter(os.Stdout, 0, 1, 2, ' ', 0)
fmt.Fprint(w, "# Taxid\tRank\tGenomes")
if *optN {
        fmt.Fprint(w, "\tName")
}
fmt.Fprint(w, "\n")
for _, v := range subtree {
        ⟨List one node, Pr. 2.3 17d⟩
}
w.Flush()
```

We import `tabwriter`.

17c ⟨*Imports, Pr. 2.3* 14d⟩+≡ (14a) ◁ 16c

```
"text/tabwriter"
```

The user might have opted to list only taxon nodes with genomes and/or to add names to taxa.

17d ⟨*List one node, Pr. 2.3* 17d⟩≡ (17b)

```
n := len(taxdb.Accessions(v))
if !*optG || n > 0 {
        r := taxdb.Rank(v)
        fmt.Fprintf(w, "%d\t%s\t%d", v, r, n)
        if *optN {
                a := taxdb.Name(v)
                fmt.Fprintf(w, "\t%s", a)
        }
        fmt.Fprintf(w, "\n")
}
```

The default node is simply its taxon ID. However, nodes can be annotated with color and a taxon name. For this purpose we declare templates for annotation lines, one for color, “lightsalmon”, the other for the name.

Then we draw the header and footer of the graph, and the actual subtree sandwiched in between. To improve legibility, that subtree is oriented left to right rather than the default top to bottom. For each node v in the subtree we print v and its parent.

18a ⟨*Draw subtree graph, Pr. 2.3* 18a⟩≡ (17a)

```
t1 := "\t%d [color=\"lightsalmon\",style=filled]\n"
t2 := "\t%d [label=\"%s\"]\n"
fmt.Printf("digraph g {\n\trankdir=LR\n")
for _, v := range subtree {
          ⟨Print v and its parent, Pr. 2.3 18b⟩
}
fmt.Printf("}\n")
```

We print a node , v, if one of two conditions holds. Either the user did not restrict the output to genomes-only, or the user did make this restriction and v has a genome in its subtree. If v isn't the root of the subtree, we look up its parent, p. If p isn't identical to v, in other words, if v is not the global root, we also print p and the connecting edge.

18b ⟨*Print v and its parent, Pr. 2.3* 18b⟩≡ (18a)

```
if !*optG || (*optG && hasGsub[v]) {
          ⟨Print v, Pr. 2.3 18c⟩
          if v != tid {
                    p := taxdb.Parent(v)
                    if p != v {
                              fmt.Printf("\t%d -> %d\n", p, v)
                    }
          }
}
```

If the node has a genome attached, or the user asked for names, we print an attribute line for v.

18c ⟨*Print v, Pr. 2.3* 18c⟩≡ (18b)

```
if hasGenome[v] {
          fmt.Printf(t1, v)
}
if *optN {
          fmt.Printf(t2, v, taxdb.Name(v))
}
```

We've finished writing `dree`, now we test it.

Testing

Our outline for testing `dree` contains hooks for imports and the testing logic.

19a ⟨*dree_test.go* 19a⟩≡
```
package main

import (
          "testing"
          ⟨Testing imports, Pr. 2.3 19c⟩
)

func TestDree(t *testing.T) {
          ⟨Testing, Pr. 2.3 19b⟩
}
```

We construct tests and run them in a loop.

19b ⟨*Testing, Pr. 2.3* 19b⟩≡ (19a)
```
var tests []*exec.Cmd
⟨Construct tests, Pr. 2.3 19d⟩
for i, test := range tests {
          ⟨Run test, Pr. 2.3 20b⟩
}
```

We import `exec`.

19c ⟨*Testing imports, Pr. 2.3* 19c⟩≡ (19a) 20c ▷
```
"os/exec"
```

In our tests, we draw the *Homininae* which have taxon-ID 207598. First with no options, then with names, then with genomes, then with list.

19d ⟨*Construct tests, Pr. 2.3* 19d⟩≡ (19b) 20a ▷
```
n := "207598"
d := "../data/neidb"
test := exec.Command("./dree", n, d)
tests = append(tests, test)
test = exec.Command("./dree", "-n", n, d)
tests = append(tests, test)
test = exec.Command("./dree", "-g", n, d)
tests = append(tests, test)
test = exec.Command("./dree", "-l", n, d)
tests = append(tests, test)
```

The list option can be combined with genome, and name, yielding another three combinations to test.

20a ⟨*Construct tests, Pr. 2.3* 19d⟩+≡ (19b) ◁ 19d

```
test = exec.Command("./dree", "-l", "-g", n, d)
tests = append(tests, test)
test = exec.Command("./dree", "-l", "-n", n, d)
tests = append(tests, test)
test = exec.Command("./dree", "-l", "-g", "-n", n, d)
tests = append(tests, test)
```

For each test we compare the output we get with the output we want, which is contained in `r1.txt`, `r2.txt`, and so on.

20b ⟨*Run test, Pr. 2.3* 20b⟩≡ (19b)

```
get, err := test.Output()
if err != nil {
          t.Errorf("couldn't run %q", test)
}
f := "r" + strconv.Itoa(i+1) + ".txt"
want, err := ioutil.ReadFile(f)
if err != nil {
          t.Errorf("couldn't open %q", f)
}
if !bytes.Equal(get, want) {
          t.Errorf("get:\n%s\nwant:\n%s", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

20c ⟨*Testing imports, Pr. 2.3* 19c⟩+≡ (19a) ◁ 19c

```
"strconv"
"io/ioutil"
"bytes"
```

2.4 `ants`

Given a taxon-ID, `ants` returns all ancestors and their taxonomic ranks.

Implementation

The program `ants` contains hooks for imports and the logic of the main function.

Prog. 2.4 (`ants`)

21a ⟨*ants.go* 21a⟩≡
```
package main

import (
        ⟨Imports, Pr. 2.4 21d⟩
)

func main() {
        ⟨Main function, Pr. 2.4 21b⟩
}
```

In the main function we set the usage, declare the options, parse the options, get the ancestors, and print them.

21b ⟨*Main function, Pr. 2.4* 21b⟩≡ (21a)
```
⟨Set usage, Pr. 2.4 21c⟩
⟨Declare options, Pr. 2.4 21e⟩
⟨Parse options, Pr. 2.4 22a⟩
⟨Get ancestors, Pr. 2.4 22e⟩
⟨Print ancestors, Pr. 2.4 23a⟩
```

The usage consists of the actual usage message, an explanation of the purpose of `ants`, and an example command.

21c ⟨*Set usage, Pr. 2.4* 21c⟩≡ (21b)
```
u := "ants [option] <taxon-ID> <db>"
p := "Get a taxon's ancestors."
e := "ants 9606 neidb"
clio.Usage(u, p, e)
```

We import `clio`.

21d ⟨*Imports, Pr. 2.4* 21d⟩≡ (21a) 21f ▷
```
"github.com/evolbioinf/clio"
```

There's only one option, the version.

21e ⟨*Declare options, Pr. 2.4* 21e⟩≡ (21b)
```
var optV = flag.Bool("v", false, "version")
```

We import `flag`.

21f ⟨*Imports, Pr. 2.4* 21d⟩+≡ (21a) ◁ 21d 22b ▷
```
"flag"
```

We parse the options and respond to `-v` as this might stop the program. Then we extract the remaining arguments, check there are two of them, and store the taxon-ID and the database.

22a ⟨*Parse options, Pr. 2.4* 22a⟩≡ (21b)
```
flag.Parse()
if *optV {
          util.PrintInfo("ants")
}
args := flag.Args()
⟨Check arguments, Pr. 2.4 22c⟩
tid, err := strconv.Atoi(args[0])
if err != nil { log.Fatal(err) }
db := args[1]
```

We import `util`, `strconv`, and `log`.

22b ⟨*Imports, Pr. 2.4* 21d⟩+≡ (21a) ◁21f 22d▷
```
"github.com/evolbioinf/neighbors/util"
"strconv"
"log"
```

If the user didn't submit two arguments, something is bound to have gone wrong and we bail with a friendly message.

22c ⟨*Check arguments, Pr. 2.4* 22c⟩≡ (22a)
```
if len(args) != 2 {
          m := "please provide a taxon and a database"
          fmt.Fprintf(os.Stderr, "%s\n", m)
          os.Exit(-1)
}
```

We import `fmt` and `os`.

22d ⟨*Imports, Pr. 2.4* 21d⟩+≡ (21a) ◁22b 22f▷
```
"fmt"
"os"
```

We open a database connection and store the taxon-ID as the first "ancestor". Then we get its parent. As long as the parent differs from the current taxon, we haven't yet reached the root and keep climbing.

22e ⟨*Get ancestors, Pr. 2.4* 22e⟩≡ (21b)
```
taxdb := tdb.OpenTaxonomyDB(db)
var ants []int
ants = append(ants, tid)
a := taxdb.Parent(tid)
for tid != a {
          ants = append(ants, a)
          tid = a
          a = taxdb.Parent(tid)
}
```

We import `tdb`.

22f ⟨*Imports, Pr. 2.4* 21d⟩+≡ (21a) ◁22d 23b▷
```
"github.com/evolbioinf/neighbors/tdb"
```

We print the ancestors and their ranks starting with the most remote. To make the result look nice, we print a table and lay it out with a `tabwriter`.

23a ⟨*Print ancestors, Pr. 2.4* 23a⟩≡ (21b)
```
w := tabwriter.NewWriter(os.Stdout, 0, 1, 2, ' ', 0)
defer w.Flush()
fmt.Fprintf(w, "# Back\tID\tName\tRank\n")
for i := len(ants) - 1; i >= 0; i-- {
          a := ants[i]
          n := taxdb.Name(a)
          r := taxdb.Rank(a)
          fmt.Fprintf(w, "  %d\t%d\t%s\t%s\n", i, a, n, r)
}
```

We import `tabwriter`.

23b ⟨*Imports, Pr. 2.4* 21d⟩+≡ (21a) ◁22f
```
"text/tabwriter"
```

We have finished `ants`, so let's test it.

Testing

The program for testing `ants` has hooks for imports and for the testing logic.

23c ⟨*ants_test.go* 23c⟩≡
```
package main

import (
          "testing"
          ⟨Testing imports, Pr. 2.4 23e⟩
)

func TestAnts(t *testing.T) {
          ⟨Testing, Pr. 2.4 23d⟩
}
```

We construct a set of tests and iterate over them.

23d ⟨*Testing, Pr. 2.4* 23d⟩≡ (23c)
```
var tests []*exec.Cmd
⟨Construct tests, Pr. 2.4 23f⟩
for i, test := range tests {
          ⟨Run test, Pr. 2.4 24a⟩
}
```

We import `exec`.

23e ⟨*Testing imports, Pr. 2.4* 23e⟩≡ (23c) 24b▷
```
"os/exec"
```

A single test should suffice for now, for which we start from *Homo sapiens* (9606).

23f ⟨*Construct tests, Pr. 2.4* 23f⟩≡ (23d)
```
tid := "9606"
db := "../data/neidb"
test := exec.Command("./ants", tid, db)
tests = append(tests, test)
```

When running the test, we compare the result we get with the result we want contained in `r1.txt`.

24a ⟨*Run test, Pr. 2.4* 24a⟩≡ (23d)

```
get, err := test.Output()
if err != nil { t.Error(err) }
f := "r" + strconv.Itoa(i + 1) + ".txt"
want, err := ioutil.ReadFile(f)
if err != nil { t.Error(err) }
if !bytes.Equal(get, want) {
          t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

24b ⟨*Testing imports, Pr. 2.4* 23e⟩+≡ (23c) ◁23e

```
"strconv"
"io/ioutil"
"bytes"
```

2.5 `neighbors`

The program `neighbors` takes as input a set of target taxon-IDs. It finds their most recent common ancestor, and from there calculates two new sets of taxon-IDs, the complete set of targets, which comprises at least the input taxa, and the neighbors. For each taxon in these two sets it also returns the accessions of the corresponding genome sequences.

Implementation

The outline of `neighbors` contains hooks for imports, functions, and the logic of the main function.

Prog. 2.5 (`neighbors`)

25a ⟨*neighbors.go* 25a⟩≡

```
package main

import (
        ⟨Imports, Pr. 2.5 25d⟩
)
⟨Functions, Pr. 2.5 27a⟩
func main() {
        ⟨Main function, Pr. 2.5 25b⟩
}
```

In the main function we set the usage, declare the options, parse the options, and parse the input files.

25b ⟨*Main function, Pr. 2.5* 25b⟩≡ (25a)

```
⟨Set usage, Pr. 2.5 25c⟩
⟨Declare options, Pr. 2.5 26a⟩
⟨Parse options, Pr. 2.5 26c⟩
⟨Parse input files, Pr. 2.5 26g⟩
```

The usage consists of the actual usage message, an explanation of the purpose of `neighbors`, and an example command. In the usage we clarify that the first argument of `neighbors` is the name of the taxonomy database.

25c ⟨*Set usage, Pr. 2.5* 25c⟩≡ (25b)

```
u := "neighbors [-h] [option]... <db> [targets.txt]..."
p := "Given a taxonomy database computed with makeNeiDb and " +
        "a set of target taxon-IDs, find their closest " +
        "taxonomic neighbors."
e := "neighbors neidb targetIds.txt"
clio.Usage(u, p, e)
```

We import `clio`.

25d ⟨*Imports, Pr. 2.5* 25d⟩≡ (25a) 26b ▷

```
"github.com/evolbioinf/clio"
```

Apart from the version, we declare an option to restrict the output to taxa with genomes (`-g`), and an option to just list the genomes (`-l`).

26a ⟨*Declare options, Pr. 2.5* 26a⟩≡ (25b)
```
optV := flag.Bool("v", false, "version")
optG := flag.Bool("g", false, "genome sequences only")
optL := flag.Bool("l", false, "list genomes")
```

We import `flag`.

26b ⟨*Imports, Pr. 2.5* 25d⟩+≡ (25a) ◁25d 26d▷
```
"flag"
```

We parse the options and respond to `-v` as this stops `neighbors`. The remaining token on the command line are taken as file names, the first of which is the mandatory database. If it isn't provided, we kindly request it. Then we open the taxonomy database and remove the database name from the list of input files.

26c ⟨*Parse options, Pr. 2.5* 26c⟩≡ (25b)
```
flag.Parse()
if *optV {
        util.PrintInfo("neighbors")
}
files := flag.Args()
if len(files) == 0 {
        fmt.Fprintf(os.Stderr,
                "please provide a database name\n")
        os.Exit(0)
}
⟨Open taxonomy database, Pr. 2.5 26e⟩
files = files[1:]
```

We import `util`, `fmt`, and `os`.

26d ⟨*Imports, Pr. 2.5* 25d⟩+≡ (25a) ◁26b 26f▷
```
"github.com/evolbioinf/neighbors/util"
"fmt"
"os"
```

We open a connection to the taxonomy database.

26e ⟨*Open taxonomy database, Pr. 2.5* 26e⟩≡ (26c)
```
taxdb := tdb.OpenTaxonomyDB(files[0])
```

We import `tdb`.

26f ⟨*Imports, Pr. 2.5* 25d⟩+≡ (25a) ◁26d 27b▷
```
"github.com/evolbioinf/neighbors/tdb"
```

The input files are parsed using the function `clio.ParseFiles`, which takes as argument the function `parse`. In turn, `parse` takes the database connection, the list option, and the genomes option as arguments.

26g ⟨*Parse input files, Pr. 2.5* 26g⟩≡ (25b)
```
clio.ParseFiles(files, parse, taxdb, *optL, *optG)
```

Inside `parse`, we retrieve the taxonomy database, the list option, and the genomes option. Then we read the taxon-IDs and compute two new sets of taxon-IDs, the targets, and the neighbors. For each element of these sets we also look up the genome accessions. Then we print the combined result.

27a ⟨*Functions, Pr. 2.5* 27a⟩≡ (25a)
```
func parse(r io.Reader, args ...interface{}) {
        taxdb := args[0].(*tdb.TaxonomyDB)
        optL := args[1].(bool)
        optG := args[2].(bool)
        ⟨Read taxon-IDs, Pr. 2.5 27c⟩
        ⟨Compute targets, Pr. 2.5 27e⟩
        ⟨Compute neighbors, Pr. 2.5 28c⟩
        ⟨Look up genomes, Pr. 2.5 29a⟩
        ⟨Print result, Pr. 2.5 29b⟩
}
```

We import `io`.

27b ⟨*Imports, Pr. 2.5* 25d⟩+≡ (25a) ◁26f 27d▷
```
"io"
```

We read the taxon-IDs as a slice of integers from the input file. We ignore blank lines and lines starting with a hash.

27c ⟨*Read taxon-IDs, Pr. 2.5* 27c⟩≡ (27a)
```
var taxa []int
sc := bufio.NewScanner(r)
for sc.Scan() {
        s := sc.Text()
        if s == "" || s[0] == '#' { continue }
        i, err := strconv.Atoi(s)
        if err != nil {
                log.Fatalf("couldn't convert %q", s)
        }
        taxa = append(taxa, i)
}
```

We import `bufio`, `strconv`, and `log`.

27d ⟨*Imports, Pr. 2.5* 25d⟩+≡ (25a) ◁27b 28b▷
```
"bufio"
"strconv"
"log"
```

The targets are nodes of the subtree of the most recent common ancestor of the taxon-IDs we just read. We call this MRCA the target-MRCA, `mrcaT`, and calculate it from the partial taxonomy of the targets. The subtree rooted on the `mrcaT` contains the complete set of targets. This complete set of targets might contain new targets not contained in the taxa submitted. We mark these new targets.

27e ⟨*Compute targets, Pr. 2.5* 27e⟩≡ (27a)
```
mrcaT := taxdb.MRCA(taxa)
targets := taxdb.Subtree(mrcaT)
⟨Mark new targets, Pr. 2.5 28a⟩
```

The new targets are put in a map.

28a ⟨*Mark new targets, Pr. 2.5* 28a⟩≡ (27e)
```
newTargets := make(map[int]bool)
sort.Ints(taxa)
l := len(taxa)
for _, t := range targets {
        i := sort.SearchInts(taxa, t)
        if !(i < l && taxa[i] == t) {
                newTargets[t] = true
        }
}
```

We import `sort`.

28b ⟨*Imports, Pr. 2.5* 25d⟩+≡ (25a) ◁27d 29d▷
```
"sort"
```

The neighbors are the nodes in the subtree of the parent of the target-MRCA, minus the targets and the parent. We call the parent of the target-MRC the analysis-MRCA, `mrcaA`. If the analysis-MRCA has no nodes other than the target(s), we keep moving it up the tree until we've found at least one neighbor.

28c ⟨*Compute neighbors, Pr. 2.5* 28c⟩≡ (27a)
```
var neighbors []int
mrcaA := mrcaT
for len(neighbors) == 0 {
        mrcaA = taxdb.Parent(mrcaA)
        nodes := taxdb.Subtree(mrcaA)
        ⟨Subtract targets from the nodes, Pr. 2.5 28d⟩
}
```

Nodes that are not targets or `mrcaA` must be neighbors.

28d ⟨*Subtract targets from the nodes, Pr. 2.5* 28d⟩≡ (28c)
```
sort.Ints(targets)
l = len(targets)
for _, node := range nodes {
        i := sort.SearchInts(targets, node)
        if !(i < l && node == targets[i]) {
                if node != mrcaA {
                        neighbors = append(neighbors, node)
                }
        }
}
```

We look up the genome accessions that belong to each taxon-ID through a method call on the database. The genomes are stored in a map between taxon-IDs and string slices.

29a ⟨*Look up genomes, Pr. 2.5* 29a⟩≡ (27a)
```
genomes := make(map[int][]string)
for _, t := range targets {
        genomes[t] = taxdb.Accessions(t)
}
for _, t := range neighbors {
        genomes[t] = taxdb.Accessions(t)
}
```

The result consists either of a list of accessions or of the full report.

29b ⟨*Print result, Pr. 2.5* 29b⟩≡ (27a)
```
if optL {
        ⟨List accessions, Pr. 2.5 29c⟩
} else {
        ⟨Print report, Pr. 2.5 30b⟩
}
```

We list the accessions in a table constructed with a `tabwriter`, which we flush at the end. The accessions table has a header consisting of the sample (target or neighbor) and the accession. We first list the targets, then the neighbors.

29c ⟨*List accessions, Pr. 2.5* 29c⟩≡ (29b)
```
w := tabwriter.NewWriter(os.Stdout, 1, 0, 2, ' ', 0)
fmt.Fprintf(w, "# Sample\tAccession\n")
⟨List targets, Pr. 2.5 29e⟩
⟨List neighbors, Pr. 2.5 30a⟩
w.Flush()
```

We import `tabwriter`.

29d ⟨*Imports, Pr. 2.5* 25d⟩+≡ (25a) ◁28b 30d▷
```
"text/tabwriter"
```

For each target we get the list of genome accessions and print each one.

29e ⟨*List targets, Pr. 2.5* 29e⟩≡ (29c)
```
sample := "t"
for _, target := range targets {
        accessions := genomes[target]
        for _, accession := range accessions {
                fmt.Fprintf(w, "%s\t%s\n", sample, accession)
        }
}
```

We list the neighbors like we just listed the targets.

30a ⟨*List neighbors, Pr. 2.5* 30a⟩≡ (29c)

```
sample = "n"
for _, neighbor := range neighbors {
        accessions := genomes[neighbor]
        for _, accession := range accessions {
                fmt.Fprintf(w, "%s\t%s\n", sample, accession)
        }
}
```

We start the report with the taxon-IDs and scientific names of the most recent common ancestors of the targets and of the whole analysis, that is, the MRCA of the targets and neighbors combined. Then we print a table of results consisting of four columns: the taxon type—neighbor or target, the taxon-ID, taxon name, and the genome accessions. Like the replicon list, we typeset again the report table using a tab writer and flush it at the end.

30b ⟨*Print report, Pr. 2.5* 30b⟩≡ (29b)

```
mrcaTname := taxdb.Name(mrcaT)
mrcaAname := taxdb.Name(mrcaA)
fmt.Printf("# MRCA(targets): %d, %s\n", mrcaT, mrcaTname)
fmt.Printf("# MRCA(targets+neighbors): %d, %s\n", mrcaA, mrcaAname)
w := tabwriter.NewWriter(os.Stdout, 1, 0, 2, ' ', 0)
fmt.Fprint(w, "# Type\tTaxon-ID\tName\tGenomes\n")
⟨Print targets, Pr. 2.5 30c⟩
⟨Print neighbors, Pr. 2.5 31a⟩
w.Flush()
```

Targets that were already in the input are marked as type “t”, new targets as “tt”. We print the genome accessions in a single string punctuated by pipe symbols. If the user requested only taxa with genomes, we restrict our output to them.

30c ⟨*Print targets, Pr. 2.5* 30c⟩≡ (30b)

```
for _, target := range targets {
        t := "t"
        if newTargets[target] { t = "tt" }
        g := "-"
        if len(genomes[target]) > 0 {
                g = strings.Join(genomes[target], "|")
        }
        if optG && g == "-" { continue }
        fmt.Fprintf(w, "%s\t%d\t%s\t%s\n", t, target,
                taxdb.Name(target),
                strings.TrimPrefix(g, " "))
}
```

We import strings.

30d ⟨*Imports, Pr. 2.5* 25d⟩+≡ (25a) ◁29d

```
"strings"
```

Neighbors are marked as type “n”. Since the targets were already sorted, we sort the neighbors, too.

31a ⟨*Print neighbors, Pr. 2.5* 31a⟩≡ (30b)
```
sort.Ints(neighbors)
for _, neighbor := range neighbors {
        g := "-"
        if len(genomes[neighbor]) > 0 {
                g = strings.Join(genomes[neighbor], "|")
        }
        if optG && g == "-" { continue }
        n := taxdb.Name(neighbor)
        fmt.Fprintf(w, "n\t%d\t%s\t%s\n", neighbor, n,
                strings.TrimPrefix(g, " "))
}
```

We’ve written `neighbors`, time to test it.

Testing

The outline for testing `neighbors` has hooks for imports and the testing logic.

31b ⟨*neighbors_test.go* 31b⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Pr. 2.5 31d⟩
)

func TestNeighbors(t *testing.T) {
        ⟨Testing, Pr. 2.5 31c⟩
}
```

We construct a set of tests and run each one.

31c ⟨*Testing, Pr. 2.5* 31c⟩≡ (31b)
```
var tests []*exec.Cmd
⟨Construct tests, Pr. 2.5 32a⟩
for i, test := range tests {
        ⟨Run test, Pr. 2.5 32c⟩
}
```

We import `exec`.

31d ⟨*Testing imports, Pr. 2.5* 31d⟩≡ (31b) 32b ▷
```
"os/exec"
```

We run `neighbors` on the full database with four sets of target-IDs, `tid1.txt`, `tid2.txt`, and so on. On `tid4.txt` we also test the list option (`-l`) and the genomes option (`-g`).

32a ⟨*Construct tests, Pr. 2.5* 32a⟩≡ (31c)
```
db := "../data/neidb"
for i := 1; i <= 4; i++ {
          in := "tid" + strconv.Itoa(i) + ".txt"
          test := exec.Command("./neighbors", db, in)
          tests = append(tests, test)
}
test := exec.Command("./neighbors", "-l", db, "tid4.txt")
tests = append(tests, test)
test = exec.Command("./neighbors", "-g", db, "tid4.txt")
tests = append(tests, test)
```

We import `strconv`.

32b ⟨*Testing imports, Pr. 2.5* 31d⟩+≡ (31b) ◁31d 32d▷
```
"strconv"
```

For a given test we compare the result we get with the result we want, which is contained in files `r1.txt`, `r2.txt`, and so on.

32c ⟨*Run test, Pr. 2.5* 32c⟩≡ (31c)
```
get, err := test.Output()
if err != nil {
          t.Errorf("couldn't run %q", test)
}
f := "r" + strconv.Itoa(i+1) + ".txt"
want, err := ioutil.ReadFile(f)
if err != nil {
          t.Errorf("couldn't open %q", f)
}
if !bytes.Equal(get, want) {
          t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `ioutil` and `bytes`.

32d ⟨*Testing imports, Pr. 2.5* 31d⟩+≡ (31b) ◁32b
```
"io/ioutil"
"bytes"
```

2.6 `land`

The program `land` labels the internal nodes of a Newick tree. This makes it easy to subsequently pick individual clades in the tree with `pickle`.

Implementation

`land` contains hooks for imports, functions, and the logic of the main function.

Prog. 2.6 (`land`)

33a ⟨*land.go* 33a⟩≡
```
package main

import (
        ⟨Imports, Pr. 2.6 33d⟩
)
⟨Functions, Pr. 2.6 34d⟩
func main() {
        ⟨Main function, Pr. 2.6 33b⟩
}
```

In the main function of `land` we set the usage, declare the options, parse the options, and parse the input files.

33b ⟨*Main function, Pr. 2.6* 33b⟩≡ (33a)
```
⟨Set usage, Pr. 2.6 33c⟩
⟨Declare options, Pr. 2.6 33e⟩
⟨Parse options, Pr. 2.6 34a⟩
⟨Parse input files, Pr. 2.6 34c⟩
```

The usage consists of the actual usage message, an explanation of the purpose of `land`, and an example command.

33c ⟨*Set usage, Pr. 2.6* 33c⟩≡ (33b)
```
u := "land [option]... [treeFile]..."
p := "Label the internal nodes in Newick trees."
e := "land -p n foo.nwk"
clio.Usage(u, p, e)
```

We import `clio`.

33d ⟨*Imports, Pr. 2.6* 33d⟩≡ (33a) 33f ▷
```
"github.com/evolbioinf/clio"
```

The user can request the version (`-v`), set a label prefix (`-p`), or set a label suffix (`-s`).

33e ⟨*Declare options, Pr. 2.6* 33e⟩≡ (33b)
```
var optV = flag.Bool("v", false, "version")
var optP = flag.String("p", "", "prefix")
var optS = flag.String("s", "", "suffix")
```

We import `flag`.

33f ⟨*Imports, Pr. 2.6* 33d⟩+≡ (33a) ◁ 33d 34b ▷
```
"flag"
```

We parse the options and respond to a request for the version, as this would terminate the program.

34a ⟨*Parse options, Pr. 2.6* 34a⟩≡ (33b)
```
flag.Parse()
if *optV {
	util.PrintInfo("land")
}
```

We import `util`.

34b ⟨*Imports, Pr. 2.6* 33d⟩+≡ (33a) ◁33f 34e▷
```
"github.com/evolbioinf/neighbors/util"
```

The remaining tokens on the command line are interpreted as the names of input files. These are the argument of the function `ParseFiles`. In addition, `ParseFiles` takes as argument the function `parse`, which in turn takes as arguments the prefix and the suffix.

34c ⟨*Parse input files, Pr. 2.6* 34c⟩≡ (33b)
```
files := flag.Args()
clio.ParseFiles(files, parse, *optP, *optS)
```

Inside `parse`, we retrieve the arguments we just passed and iterate over the input file.

34d ⟨*Functions, Pr. 2.6* 34d⟩≡ (33a) 35a▷
```
func parse(r io.Reader, args ...interface{}) {
	⟨Retrieve arguments, Pr. 2.6 34f⟩
	⟨Iterate over input, Pr. 2.6 34g⟩
}
```

We import `io`.

34e ⟨*Imports, Pr. 2.6* 33d⟩+≡ (33a) ◁34b 34h▷
```
"io"
```

We retrieve the prefix and the suffix.

34f ⟨*Retrieve arguments, Pr. 2.6* 34f⟩≡ (34d)
```
pr := args[0].(string)
su := args[1].(string)
```

We scan the trees in the input. Each tree is labeled starting with 1 and printed.

34g ⟨*Iterate over input, Pr. 2.6* 34g⟩≡ (34d)
```
sc := nwk.NewScanner(r)
for sc.Scan() {
	tree := sc.Tree()
	labelTree(tree, 1, pr, su)
	fmt.Println(tree)
}
```

We import `nwk` and `fmt`.

34h ⟨*Imports, Pr. 2.6* 33d⟩+≡ (33a) ◁34e 35c▷
```
"github.com/evolbioinf/nwk"
"fmt"
```

Inside the function `labelTree` we label the nodes in order.

35a ⟨*Functions, Pr. 2.6* 34d⟩+≡ (33a) ◁34d
```
func labelTree(v *nwk.Node, c int, pr, su string) int {
        if v == nil {
                return c
        }
        ⟨Label node, Pr. 2.6 35b⟩
        c = labelTree(v.Child, c, pr, su)
        c = labelTree(v.Sib, c, pr, su)
        return c
}
```

When labeling a node, we leave leaf nodes unchanged, but construct new labels for internal nodes.

35b ⟨*Label node, Pr. 2.6* 35b⟩≡ (35a)
```
l := v.Label
if v.Child != nil {
        l = pr + strconv.Itoa(c) + su
        c++
}
v.Label = l
```

We import `strconv`.

35c ⟨*Imports, Pr. 2.6* 33d⟩+≡ (33a) ◁34h
```
"strconv"
```

We're done writing `land`, let's test it.

Testing

The code for testing `land` has hooks for imports and the testing logic.

35d ⟨*land_test.go* 35d⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Pr. 2.6 36a⟩
)

func TestLand(t *testing.T) {
        ⟨Testing, Pr. 2.6 35e⟩
}
```

We construct a set of tests, iterate over them, and run each one.

35e ⟨*Testing, Pr. 2.6* 35e⟩≡ (35d)
```
var tests []*exec.Cmd
⟨Construct tests, Pr. 2.6 36b⟩
for i, test := range tests {
        ⟨Run test, Pr. 2.6 36c⟩
}
```

We import exec.

36a ⟨*Testing imports, Pr. 2.6* 36a⟩≡ (35d) 36d ▷
```
"os/exec"
```

We construct three tests, one with default options, one where we set a prefix, and one where we set a suffix. Each test works on the tree contained in `test.nwk`.

36b ⟨*Construct tests, Pr. 2.6* 36b⟩≡ (35e)
```
f := "test.nwk"
test := exec.Command("./land", f)
tests = append(tests, test)
test = exec.Command("./land", "-p", "p", f)
tests = append(tests, test)
test = exec.Command("./land", "-s", "s", f)
tests = append(tests, test)
```

For a given tests, we compare the result we get with the result we want. The results we want are contained in files `r1.txt`, `r2.txt`, and `r3.txt`.

36c ⟨*Run test, Pr. 2.6* 36c⟩≡ (35e)
```
get, err := test.Output()
if err != nil {
          t.Error(err)
}
f := "r" + strconv.Itoa(i+1) + ".txt"
want, err := ioutil.ReadFile(f)
if err != nil {
          t.Error(err)
}
if !bytes.Equal(get, want) {
          t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

36d ⟨*Testing imports, Pr. 2.6* 36a⟩+≡ (35d) ◁ 36a
```
"strconv"
"io/ioutil"
"bytes"
```

2.7 pickle

In Newick trees with labeled clades, `pickle` picks the clades requested by the user and prints their leaf labels. It can also print the requested clade as a Newick tree. The program also has a “complement” option to get everything but the nodes in the designated clade.

Implementation

`pickle` contains hooks for imports, functions, and the logic of the main function.

Prog. 2.7 (pickle)

37a ⟨*pickle.go* 37a⟩≡
```
package main

import (
          ⟨Imports, Pr. 2.7 37d⟩
)
⟨Functions, Pr. 2.7 38e⟩
func main() {
          ⟨Main function, Pr. 2.7 37b⟩
}
```

In the main function we set the usage, declare the options, parse the options, and parse the input files.

37b ⟨*Main function, Pr. 2.7* 37b⟩≡ (37a)
```
⟨Set usage, Pr. 2.7 37c⟩
⟨Declare options, Pr. 2.7 37e⟩
⟨Parse options, Pr. 2.7 38b⟩
⟨Parse input files, Pr. 2.7 38d⟩
```

The usage consists of three parts, the usage message proper, an explanation of the purpose of `pickle`, and an example command.

37c ⟨*Set usage, Pr. 2.7* 37c⟩≡ (37b)
```
u := "pickle [option]... [foo.nwk]..."
p := "Pick clades in Newick trees."
e := "pickle 3,5 foo.nwk"
clio.Usage(u, p, e)
```

We import `clio`.

37d ⟨*Imports, Pr. 2.7* 37d⟩≡ (37a) 38a ▷
```
"github.com/evolbioinf/clio"
```

Apart from the version option, `-v`, we declare an option for printing the tree picked, `-t`, and an option for complementation, `-c`.

37e ⟨*Declare options, Pr. 2.7* 37e⟩≡ (37b)
```
optV := flag.Bool("v", false, "version")
optT := flag.Bool("t", false, "print tree")
optC := flag.Bool("c", false, "complement")
```

We import `flag`.

38a ⟨*Imports, Pr. 2.7* 37d⟩+≡ (37a) ◁37d 38c▷
```
"flag"
```

We parse the version option, `-v`, and respond to it, as it would terminate `pickle`. The first entry in the argument array is the labels string. If the user hasn't supplied one, we bail with a friendly message. Otherwise we split the labels at commas.

38b ⟨*Parse options, Pr. 2.7* 38b⟩≡ (37b)
```
flag.Parse()
if *optV {
          util.PrintInfo("pickle")
}
args := flag.Args()
if len(args) < 1 {
          fmt.Fprintf(os.Stderr, "please enter a clade identifier\n")
          os.Exit(-1)
}
labels := strings.Split(args[0],",")
```

We import `util`, `fmt`, `os`, and `strings`.

38c ⟨*Imports, Pr. 2.7* 37d⟩+≡ (37a) ◁38a 38f▷
```
"github.com/evolbioinf/neighbors/util"
"fmt"
"os"
"strings"
```

The remaining command line arguments are taken as file names. The files are parsed with `ParseFiles`, which calls `parse` on every input file. The function `parse` in turn takes the slice of labels the tree option, and the complement option as arguments.

38d ⟨*Parse input files, Pr. 2.7* 38d⟩≡ (37b)
```
clio.ParseFiles(args[1:], parse, labels, optT, optC)
```

Inside `parse`, we retrieve the labels and options, iterate over the input trees, and for each tree print a table header and iterate over the labels of the selected nodes.

38e ⟨*Functions, Pr. 2.7* 38e⟩≡ (37a) 40a▷
```
func parse(r io.Reader, args ...interface{}) {
          labels := args[0].([]string)
          optT := args[1].(*bool)
          optC := args[2].(*bool)
          sc := nwk.NewScanner(r)
          for sc.Scan() {
                    origRoot := sc.Tree()
                    ⟨Print table header, Pr. 2.7 39a⟩
                    ⟨Iterate over labels, Pr. 2.7 39b⟩
          }
}
```

We import `io`.

38f ⟨*Imports, Pr. 2.7* 37d⟩+≡ (37a) ◁38c 39e▷
```
"io"
```

The table header starts with a hash and says “Selected clade” if only one was selected, “clades” for plural.

39a ⟨*Print table header, Pr. 2.7* 39a⟩≡ (38e)

```
fmt.Printf("# Selected clade")
if len(labels) > 1 {
          fmt.Printf("s")
}
fmt.Printf("\n")
```

For each label we get a pristine copy of the tree, write a header, find the corresponding clade, complement it if requested, and print it.

39b ⟨*Iterate over labels, Pr. 2.7* 39b⟩≡ (38e)

```
for _, label := range labels {
          t := origRoot.CopyClade()
          ⟨Write clade header, Pr. 2.7 39c⟩
          ⟨Find clade, Pr. 2.7 39d⟩
          if *optC {
                    ⟨Complement clade, Pr. 2.7 40d⟩
          }
          ⟨Print clade, Pr. 2.7 40e⟩
}
```

The clade header starts with a double hash, followed by “complement of”, if appropriate, followed by the label of the focal node.

39c ⟨*Write clade header, Pr. 2.7* 39c⟩≡ (39b)

```
fmt.Printf("## ")
if *optC {
          fmt.Printf("Complement of ")
}
fmt.Printf("%s\n", label)
```

We find the desired clade by converting the tree into a node slice using `tree2slice` and then iterating over the nodes. If we cannot find the node, we exit with message.

39d ⟨*Find clade, Pr. 2.7* 39d⟩≡ (39b)

```
var nodes []*nwk.Node
nodes = tree2slice(t, nodes)
found := false
⟨Iterate over nodes, Pr. 2.7 40c⟩
if !found {
          log.Fatalf("Couldn't find node %q.\n", label)
}
```

We import `log`.

39e ⟨*Imports, Pr. 2.7* 37d⟩+≡ (37a) ◁38f 40b▷

```
"log"
```

`tree2slice` is a recursive function that collects the nodes it encounters into a slice.

40a ⟨*Functions, Pr. 2.7* 38e⟩+≡ (37a) ◁38e 41c▷
```
func tree2slice(v *nwk.Node, ns []*nwk.Node) []*nwk.Node {
          if v == nil { return ns }
          ns = append(ns, v)
          ns = tree2slice(v.Child, ns)
          ns = tree2slice(v.Sib, ns)
          return ns
}
```

We import `nwk`.

40b ⟨*Imports, Pr. 2.7* 37d⟩+≡ (37a) ◁39e
```
"github.com/evolbioinf/nwk"
```

When iterating over the nodes, we look up the first one with a matching label and note that we found it.

40c ⟨*Iterate over nodes, Pr. 2.7* 40c⟩≡ (39d)
```
var clade *nwk.Node
for _, node := range nodes {
          if node.Label == label {
                  clade = node
                  found = true
                  break
          }
}
```

If the clade to be complemented is the root, we set the original tree to nil. Otherwise we remove the clade from the tree.

40d ⟨*Complement clade, Pr. 2.7* 40d⟩≡ (39b)
```
if clade.Parent == nil {
          t = nil
} else {
          clade.RemoveClade()
}
```

To print the clade, we either print its Newick tree or list its leaves.

40e ⟨*Print clade, Pr. 2.7* 40e⟩≡ (39b)
```
if *optT {
          ⟨Print tree, Pr. 2.7 41a⟩
} else {
          ⟨List leaves, Pr. 2.7 41b⟩
}
```

When printing the tree we either start with the global root or with the root of the clade. In the latter case we descend to the child to avoid the sibling nodes and add the root through an opening parenthesis, the label, and the closing semicolon.

41a ⟨*Print tree, Pr. 2.7* 41a⟩≡ (40e)
```
if *optC {
        if t != nil {
                fmt.Printf("%s\n", t)
        }
} else {
        clade = clade.Child
        if clade != nil {
                fmt.Printf("(%s%s;\n", clade, label)
        }
}
```

We delegate listing of the leaves to the function `listLeaves`, which again starts either from the global root or from the clade. If we start at the clade, we need to avoid its siblings, so we actually start at the clade's child.

41b ⟨*List leaves, Pr. 2.7* 41b⟩≡ (40e)
```
if *optC {
        listLeaves(t)
} else {
        listLeaves(clade.Child)
}
```

The function `listLeaves` prints leaf labels recursively.

41c ⟨*Functions, Pr. 2.7* 38e⟩+≡ (37a) ◁40a
```
func listLeaves(v *nwk.Node) {
        if v == nil { return }
        if v.Child == nil {
                fmt.Printf("%s\n", v.Label)
        }
        listLeaves(v.Child)
        listLeaves(v.Sib)
}
```

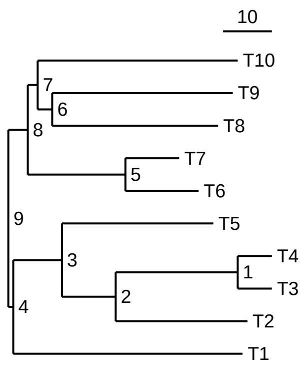

Figure 2.3: Tree for testing the program `pickle`.

We’ve finished `pickle`, so let’s test it.

Testing

Our program for testing `pickle` contains hooks for imports and the testing logic.

42a ⟨*pickle_test.go* 42a⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Pr. 2.7 43b⟩
)

func TestPickle(t *testing.T) {
        ⟨Testing, Pr. 2.7 42b⟩
}
```

We construct a set of tests and then iterate over them.

42b ⟨*Testing, Pr. 2.7* 42b⟩≡ (42a)
```
var tests []*exec.Cmd
⟨Construct tests, Pr. 2.7 43a⟩
for i, test := range tests {
        ⟨Run test, Pr. 2.7 44b⟩
}
```

We begin with four basic tests, all based on the tree in Figure 2.3 contained in the file `test.nwk`. The first test picks clade 7, the second picks clades 7 and 3, the third picks clade 9, which is the root, and the fourth test picks clade 4, which, in contrast to clades 7 and 3, is not at the end of a sibling chain.

43a ⟨*Construct tests, Pr. 2.7* 43a⟩≡ (42b) 43c ▷

```
f := "test.nwk"
test := exec.Command("./pickle", "7", f)
tests = append(tests, test)
test = exec.Command("./pickle", "7,3", f)
tests = append(tests, test)
test = exec.Command("./pickle", "9", f)
tests = append(tests, test)
test = exec.Command("./pickle", "4", f)
tests = append(tests, test)
```

We import `exec`.

43b ⟨*Testing imports, Pr. 2.7* 43b⟩≡ (42a) 44c ▷

```
"os/exec"
```

We repeat the basic tests with the complement switch.

43c ⟨*Construct tests, Pr. 2.7* 43a⟩+≡ (42b) ◁ 43a 43d ▷

```
test = exec.Command("./pickle", "-c", "7", f)
tests = append(tests, test)
test = exec.Command("./pickle", "-c", "7,3", f)
tests = append(tests, test)
test = exec.Command("./pickle", "-c", "9", f)
tests = append(tests, test)
test = exec.Command("./pickle", "-c", "4", f)
tests = append(tests, test)
```

We repeat the basic tests with the tree switch.

43d ⟨*Construct tests, Pr. 2.7* 43a⟩+≡ (42b) ◁ 43c 44a ▷

```
test = exec.Command("./pickle", "-t", "7", f)
tests = append(tests, test)
test = exec.Command("./pickle", "-t", "7,3", f)
tests = append(tests, test)
test = exec.Command("./pickle", "-t", "9", f)
tests = append(tests, test)
test = exec.Command("./pickle", "-t", "4", f)
tests = append(tests, test)
```

Our final set of tests are the four basic tests with the complement and the tree switch.

44a ⟨*Construct tests, Pr. 2.7* 43a⟩+≡ (42b) ◁43d
```
test = exec.Command("./pickle", "-c", "-t", "7", f)
tests = append(tests, test)
test = exec.Command("./pickle", "-c", "-t", "7,3", f)
tests = append(tests, test)
test = exec.Command("./pickle", "-c", "-t", "9", f)
tests = append(tests, test)
test = exec.Command("./pickle", "-c", "-t", "4", f)
tests = append(tests, test)
```

For a given test we compare the result we get with the result we want. The results we want are contained in the files `r1.txt`, `r2.txt`, and so on.

44b ⟨*Run test, Pr. 2.7* 44b⟩≡ (42b)
```
get, err := test.Output()
if err != nil {
          t.Error(err)
}
f := "r" + strconv.Itoa(i+1) + ".txt"
want, err := ioutil.ReadFile(f)
if err != nil {
          t.Error(err)
}
if !bytes.Equal(get, want) {
          t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

44c ⟨*Testing imports, Pr. 2.7* 43b⟩+≡ (42a) ◁43b
```
"strconv"
"io/ioutil"
"bytes"
```

2.8 `outliers`

Given a set of measurements, outliers are values far removed from the data's center. The following description of outliers is taken from the handbook of the American National Institute of Standards and Technology, which is published on the web[1]. The handbook follows the common definition of outliers based on the lower and upper quartile of a distribution, q_l and q_u. Let $r = q_\mathrm{u} - q_\mathrm{l}$ be the interquartile range, then $f_\mathrm{l} = q_\mathrm{l} - 1.5r$ is the *lower inner fence*, $f_\mathrm{u} = q_\mathrm{u} + 1.5r$ the *upper inner fence*. Values outside the interval spanned by the lower and upper inner fences are considered *mild* outliers. We can also define the *lower outer fence* as $F_\mathrm{l} = q_\mathrm{l} - 3r$ and the *upper outer fence* as $F_\mathrm{u} = q_\mathrm{u} + 3r$. Values outside interval spanned by the lower and upper outer fences are considered *extreme* outliers.

The program `outliers` takes as input a list of numbers and returns seven measures of their distribution:

1. the lower outer fence
2. the inner outer fence
3. the first quartile
4. the median
5. the upper quartile
6. the upper inner fence
7. the upper outer fence

It also lists mild and extreme outliers.

Implementation

Our outline of `outliers` contains hooks for imports, functions, and the logic of the main function.

Prog. 2.8 (`outlines`)

45 ⟨*outliers.go* 45⟩≡
```
package main

import (
	⟨Imports, Pr. 2.8 46b⟩
)

⟨Functions, Pr. 2.8 47a⟩
func main() {
	⟨Main function, Pr. 2.8 46a⟩
}
```

[1] `https://www.itl.nist.gov/div898/handbook/prc/section1/prc16.htm`

In the main function we set the program name and its usage, declare the options, parse the options, and parse the input files.

46a ⟨*Main function, Pr. 2.8* 46a⟩≡ (45)
```
util.SetName("outliers")
⟨Set usage, Pr. 2.8 46c⟩
⟨Declare options, Pr. 2.8 46e⟩
⟨Parse options, Pr. 2.8 46g⟩
⟨Scan input files, Pr. 2.8 46h⟩
```

We import `util`.

46b ⟨*Imports, Pr. 2.8* 46b⟩≡ (45) 46d ▷
```
"github.com/evolbioinf/neighbors/util"
```

The usage consists of three parts, the actual usage message, an explanation of the purpose of `outliers` and an example command.

46c ⟨*Set usage, Pr. 2.8* 46c⟩≡ (46a)
```
u := "outliers [option]... [file]..."
p := "List outliers according to the quartile criterion."
e := "outliers foo.dat"
clio.Usage(u, p, e)
```

We import `cio`.

46d ⟨*Imports, Pr. 2.8* 46b⟩+≡ (45) ◁ 46b 46f ▷
```
"github.com/evolbioinf/clio"
```

We declare a single option, the version.

46e ⟨*Declare options, Pr. 2.8* 46e⟩≡ (46a)
```
optV := flag.Bool("v", false, "version")
```

We import `flag`.

46f ⟨*Imports, Pr. 2.8* 46b⟩+≡ (45) ◁ 46d 47b ▷
```
"flag"
```

We parse the options and respond to a request for the version.

46g ⟨*Parse options, Pr. 2.8* 46g⟩≡ (46a)
```
flag.Parse()
if *optV {
          util.Version()
}
```

The remaining arguments on the command line are taken as file names. These are submitted to the function `ParseFiles`, which applies to each file the function `scan`.

46h ⟨*Scan input files, Pr. 2.8* 46h⟩≡ (46a)
```
files := flag.Args()
clio.ParseFiles(files, scan)
```

Inside scan we read the data into a slice of floats. We need at least four data points for an outlier analysis. If we get four or more data points, we carry out the analysis and print the results. Otherwise, we skip the current data set with a friendly warning message.

47a ⟨*Functions, Pr. 2.8* 47a⟩≡ (45) 50c ▷

```
func scan(r io.Reader, args ...interface{}) {
          data := make([]float64, 0)
          ⟨Read data, Pr. 2.8 47c⟩
          if len(data) >= 4 {
                  ⟨Analyse data, Pr. 2.8 48a⟩
                  ⟨Print results, Pr. 2.8 49c⟩
          } else {
                  m := "outliers - Need at least 4 data points " +
                          "for an outlier analysis"
                  fmt.Fprintf(os.Stderr, m)
          }
}
```

We import io and os.

47b ⟨*Imports, Pr. 2.8* 46b⟩+≡ (45) ◁ 46f 47d ▷

```
"io"
"os"
```

We scan the lines of the input file, convert each one into a number, and store that number in a slice of floats.

47c ⟨*Read data, Pr. 2.8* 47c⟩≡ (47a)

```
sc := bufio.NewScanner(r)
for sc.Scan() {
          str := strings.Trim(sc.Text(), " ")
          if len(str) > 0 {
                  d, e := strconv.ParseFloat(str, 64)
                  util.Check(e)
                  data = append(data, d)
          }
}
```

We import bufio, strings, and strconv.

47d ⟨*Imports, Pr. 2.8* 46b⟩+≡ (45) ◁ 47b 48b ▷

```
"bufio"
"strings"
"strconv"
```

We begin the data analysis by sorting the data. Then we determine the median, the lower quartile, the upper quartile, the interquartile range, the inner fences, and the outer fences.

48a ⟨*Analyse data, Pr. 2.8* 48a⟩≡ (47a)
```
sort.Float64s(data)
⟨Determine median, Pr. 2.8 48c⟩
⟨Determine lower quartile, Pr. 2.8 48d⟩
⟨Determin upper quartile, Pr. 2.8 48f⟩
⟨Determine interquartile range, Pr. 2.8 48g⟩
⟨Determine inner fences, Pr. 2.8 49a⟩
⟨Determine outer fences, Pr. 2.8 49b⟩
```

We import `sort`.

48b ⟨*Imports, Pr. 2.8* 46b⟩+≡ (45) ◁47d 48e▷
```
"sort"
```

The median is the value of the midpoint of the data range. I think of it as the second quartile.

48c ⟨*Determine median, Pr. 2.8* 48c⟩≡ (48a)
```
n := len(data)
m := (n+1) / 2
q2 := data[m-1]
if n % 2 == 0 {
          q2 = (q2 + data[m]) / 2.0
}
```

We calculate the lower, or first, quartile.

48d ⟨*Determine lower quartile, Pr. 2.8* 48d⟩≡ (48a)
```
exactQ := float64(n+1) * 0.25
f := math.Floor(exactQ)
l := int(f)
x := math.Remainder(exactQ, f)
q1 := data[l-1] + (data[l]-data[l-1]) * x
```

We import `math`.

48e ⟨*Imports, Pr. 2.8* 46b⟩+≡ (45) ◁48b 49e▷
```
"math"
```

We calculate the upper, or third, quartile.

48f ⟨*Determin upper quartile, Pr. 2.8* 48f⟩≡ (48a)
```
exactQ = float64(n+1) * 0.75
f = math.Floor(exactQ)
l = int(f)
x = math.Remainder(exactQ, f)
q3 := data[l-1] + (data[l]-data[l-1]) * x
```

The interquartile range is the difference between the upper and lower quartile.

48g ⟨*Determine interquartile range, Pr. 2.8* 48g⟩≡ (48a)
```
iq := q3 - q1
```

We determine the lower and the upper inner fence.

49a ⟨*Determine inner fences, Pr. 2.8* 49a⟩≡ (48a)
```
lif := q1 - 1.5 * iq
uif := q3 + 1.5 * iq
```

We determine the lower and upper outer fence.

49b ⟨*Determine outer fences, Pr. 2.8* 49b⟩≡ (48a)
```
lof := q1 - 3.0 * iq
uof := q3 + 3.0 * iq
```

We print the results in three steps; we print the quartiles and fences, collect the outliers, and print them.

49c ⟨*Print results, Pr. 2.8* 49c⟩≡ (47a)
```
⟨Print quartiles and fences, Pr. 2.8 49d⟩
⟨Collect outliers, Pr. 2.8 49f⟩
⟨Print outliers, Pr. 2.8 50b⟩
```

We print the quartiles and outliers in a table that we format with a tabwriter. Once we've written the table, we flush the tabwriter.

49d ⟨*Print quartiles and fences, Pr. 2.8* 49d⟩≡ (49c)
```
w := tabwriter.NewWriter(os.Stdout, 0, 1, 2, ' ', 0)
msg := "#Lower_outer_fence\tLower_inner_fence\t" +
        "Lower_quartile\tMedian\tUpper_quartile\t" +
        "Upper_inner_fence\tUpper_outer_fence"
fmt.Fprintf(w, "%s\n", msg)
fmt.Fprintf(w, "%g\t%g\t%g\t%g\t%g\t%g\t%g\n",
        lof, lif, q1, q2, q3, uif, uof)
w.Flush()
```

We import `fmt` and `tabwriter`.

49e ⟨*Imports, Pr. 2.8* 46b⟩+≡ (45) ◁48e
```
"fmt"
"text/tabwriter"
```

We collect the mild outliers and the extreme outliers.

49f ⟨*Collect outliers, Pr. 2.8* 49f⟩≡ (49c)
```
⟨Collect mild outliers, Pr. 2.8 49g⟩
⟨Collect extreme outliers, Pr. 2.8 50a⟩
```

We iterate over the data points and store the mild outliers in a slice.

49g ⟨*Collect mild outliers, Pr. 2.8* 49g⟩≡ (49f)
```
mouts := make([]float64, 0)
for _, d := range data {
        if (d > lof && d < lif) ||
                d > uif && d < uof {
                mouts = append(mouts, d)
        }
}
```

We iterate over the data points and store the extreme outliers.

50a ⟨*Collect extreme outliers, Pr. 2.8* 50a⟩≡ (49f)
```
eouts := make([]float64, 0)
for _, d := range data {
          if d < lof || d > uof {
                  eouts = append(eouts, d)
          }
}
```

To print the outliers, we call the function `printOutliers`, which takes as arguments the slice of outliers and the type of outlier.

50b ⟨*Print outliers, Pr. 2.8* 50b⟩≡ (49c)
```
printOutliers(mouts, "mild")
printOutliers(eouts, "extreme")
```

Inside `printOutliers`, we print the preamble and the values.

50c ⟨*Functions, Pr. 2.8* 47a⟩+≡ (45) ◁47a
```
func printOutliers(data []float64, kind string) {
          ⟨Print preamble, Pr. 2.8 50d⟩
          ⟨Print values, Pr. 2.8 50e⟩
}
```

In the preamble we distinguish between no outlier, one outlier, and more than one outlier. We also capitalize the beginning of a phrase.

50d ⟨*Print preamble, Pr. 2.8* 50d⟩≡ (50c)
```
n := len(data)
if n == 0 {
          fmt.Printf("No␣%s␣outliers", kind)
} else {
          s := strings.ToUpper(kind[0:1]) + kind[1:]
          fmt.Printf("%s␣outlier", s)
}
if n > 1 {
          fmt.Printf("s")
}
if n > 0 { fmt.Printf(":") }
```

The actual outlier values are printed in a single row.

50e ⟨*Print values, Pr. 2.8* 50e⟩≡ (50c)
```
for _, d := range data {
          fmt.Printf(" %g", d)
}
fmt.Printf("\n")
```

We're finished writing `outliers`, let's test it.

Testing

The outline of our testing code contains hooks for imports and the testing logic.

51a ⟨*outliers_test.go* 51a⟩≡
```
package main

import (
	"testing"
	⟨Testing imports, Pr. 2.8 51c⟩
)

func TestOutliers(t *testing.T) {
	⟨Testing, Pr. 2.8 51b⟩
}
```

We construct a set of tests and iterate over them.

51b ⟨*Testing, Pr. 2.8* 51b⟩≡ (51a)
```
var tests []*exec.Cmd
⟨Construct tests, Pr. 2.8 51d⟩
for i, test := range tests {
	⟨Run test, Pr. 2.8 52a⟩
}
```

We import `exec`.

51c ⟨*Testing imports, Pr. 2.8* 51c⟩≡ (51a) 51e ▷
```
"os/exec"
```

We run five tests, which are only distinguished by their input data. File `test2.txt` was taken from the web page by the American National Institute of Standards and Technology already mentioned[2]. The other data sets are variations.

51d ⟨*Construct tests, Pr. 2.8* 51d⟩≡ (51b)
```
for i := 1; i <= 5; i++ {
	f := "test" + strconv.Itoa(i) + ".txt"
	test := exec.Command("./outliers", f)
	tests = append(tests, test)
}
```

We import `strconv`.

51e ⟨*Testing imports, Pr. 2.8* 51c⟩+≡ (51a) ◁ 51c 52b ▷
```
"strconv"
```

[2] `https://www.itl.nist.gov/div898/handbook/prc/section1/prc16.htm`

When running a test, we compare the result we get with the result we want, which is stored in files `r1.txt`, `r2.txt`, and so on.

52a ⟨*Run test, Pr. 2.8* 52a⟩≡ (51b)
```
get, err := test.Output()
if err != nil {
          t.Error(err)
}
f := "r" + strconv.Itoa(i+1) + ".txt"
want, err := os.ReadFile(f)
if err != nil {
          t.Error(err)
}
if !bytes.Equal(get, want) {
          t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `os` and `bytes`.

52b ⟨*Testing imports, Pr. 2.8* 51c⟩+≡ (51a) ◁51e
```
"os"
"bytes"
```

2.9 fintac

The program `fintac` takes as input a Newick tree and finds the target clade. In the input tree the leaf labels have prefixes that distinguish taxonomic targets from taxonomic neighbors. In addition, the internal nodes of the input tree have unique labels. The program then looks for the node that maximizes the split between targets and neighbors.

To find this node, let n_{t} and n_{n} be the number of taxonomic targets and neighbors. We further call v_{t} and v_{n} the number of targets and neighbors in the clade rooted on v. If the taxonomy always agreed with the phylogeny, we could just look for the node that contains all targets and no neighbors, $v_{\mathrm{t}} = n_{\mathrm{t}}$ and $v_{\mathrm{n}} = 0$. But if taxonomies and phylogenies always agreed, we could restrict our search for targets and neighbors to the output of `neighbors`.

As is well-known, there is often no phylogenetic node that splits the sample into pure taxonomic targets and neighbors. So we define one last quantity, v_{n}^*, the number of neighbors in v's neighborhood, that is, the number of neighbors *outside* of v. Now we can look for the node that maximizes the percent split between targets and neighbors,

$$s(v) = \frac{v_{\mathrm{t}} + v_{\mathrm{n}}^*}{n_{\mathrm{n}} + n_{\mathrm{t}}} \times 100. \qquad (2.1)$$

This quantity ranges from 0, if v contains no targets and all neighbors, to 100, if v contains all targets and no neighbors.

Implementation

The outline of `fintac` has hooks for imports, types, methods, functions, and the logic of the main function.

Prog. 2.9 (dree)

53a ⟨*fintac.go* 53a⟩≡
```
package main

import (
          ⟨Imports, Pr. 2.9 54b⟩
)
⟨Types, Pr. 2.9 55e⟩
⟨Methods, Pr. 2.9 58c⟩
⟨Functions, Pr. 2.9 55a⟩
func main() {
          ⟨Main function, Pr. 2.9 53b⟩
}
```

Inside the main function we set the usage, declare the options, and parse them. Then we parse the input files.

53b ⟨*Main function, Pr. 2.9* 53b⟩≡ (53a)
```
⟨Set usage, Pr. 2.9 54a⟩
⟨Declare options, Pr. 2.9 54c⟩
⟨Parse options, Pr. 2.9 54e⟩
⟨Parse input files, Pr. 2.9 54g⟩
```

The usage consists of three parts, the actual usage message, an explanation of the purpose of fintac, and an example command.

54a ⟨*Set usage, Pr. 2.9* 54a⟩≡ (53b)
```
u := "fintac [option]... [foo.nwk]..."
p := "Find target clade in Newick tree."
e := "fintac foo.nwk"
clio.Usage(u, p, e)
```

We import clio.

54b ⟨*Imports, Pr. 2.9* 54b⟩≡ (53a) 54d ▷
```
"github.com/evolbioinf/clio"
```

Apart from the obligatory version option (-v), we declare an option for listing all splits, instead of the default maximal splits. We also declare an option for the neighbor prefix and the target prefix.

54c ⟨*Declare options, Pr. 2.9* 54c⟩≡ (53b)
```
optV := flag.Bool("v", false, "version")
optA := flag.Bool("a", false, "all splits (default maximal)")
optN := flag.String("n", "n", "neighbor prefix")
optT := flag.String("t", "t", "target prefix")
```

We import flag.

54d ⟨*Imports, Pr. 2.9* 54b⟩+≡ (53a) ◁ 54b 54f ▷
```
"flag"
```

We parse the options and respond to -v, as this stops the program. We also check that the target and neighbor prefixes are distinct and bail with a friendly message otherwise.

54e ⟨*Parse options, Pr. 2.9* 54e⟩≡ (53b)
```
flag.Parse()
if *optV {
          util.PrintInfo("fintac")
}
if *optN == *optT {
          log.Fatal("Please use distinct target " +
                  "and neighbor prefixes.")
}
```

We import util and log.

54f ⟨*Imports, Pr. 2.9* 54b⟩+≡ (53a) ◁ 54d 55b ▷
```
"github.com/evolbioinf/neighbors/util"
"log"
```

The remaining tokens on the command line are interpreted as the names of input files. These are passed to the function ParseFiles, which applies the function parse to each file. The function parse, in turn, takes the options -a, -n, and -t as arguments.

54g ⟨*Parse input files, Pr. 2.9* 54g⟩≡ (53b)
```
files := flag.Args()
clio.ParseFiles(files, parse, optA, optN, optT)
```

Inside `parse`, we retrieve the options, iterate over the trees inside the current file, and analyze each tree.

55a ⟨*Functions, Pr. 2.9* 55a⟩≡ (53a) 56a▷

```
func parse(r io.Reader, args ...interface{}) {
          optA := args[0].(*bool)
          optN := args[1].(*string)
          optT := args[2].(*string)
          sc := nwk.NewScanner(r)
          for sc.Scan() {
                  tree := sc.Tree()
                  ⟨Analyze tree, Pr. 2.9 55c⟩
          }
}
```

We import `io` and `nwk`.

55b ⟨*Imports, Pr. 2.9* 54b⟩+≡ (53a) ◁54f 57b▷

```
"io"
"github.com/evolbioinf/nwk"
```

To analyze a tree, we count nodes, process the counts, and print them.

55c ⟨*Analyze tree, Pr. 2.9* 55c⟩≡ (55a)

⟨*Count nodes, Pr. 2.9* 55d⟩
⟨*Process counts, Pr. 2.9* 57d⟩
⟨*Print counts, Pr. 2.9* 58d⟩

We traverse the tree by calling the function `traverseTree`, which takes as arguments a map, the neighbor prefix, and the target prefix.

55d ⟨*Count nodes, Pr. 2.9* 55d⟩≡ (55c)

```
counts := make(map[int]*Count)
traverseTree(tree, counts, *optN, *optT)
```

A count is understood per node; so it consists of the node label, the label of its parent, the number of neighbors and targets in its subtree, its split as defined in equation (2.1), and its distance to the parent.

55e ⟨*Types, Pr. 2.9* 55e⟩≡ (53a) 58b▷

```
type Count struct {
          label, parent string
          vn, vt int
          sv, dp float64
}
```

The function `traverseTree` recursively traverses the subtree of its input node. For each node it adds a count pre order and analyzes the node post order.

56a ⟨*Functions, Pr. 2.9* 55a⟩+≡ (53a) ◁55a
```
func traverseTree(v *nwk.Node, counts map[int]*Count,
          np, tp string) {
          if v == nil {
                  return
          }
          ⟨Add count, Pr. 2.9 56b⟩
          traverseTree(v.Child, counts, np, tp)
          traverseTree(v.Sib, counts, np, tp)
          ⟨Analyze v, Pr. 2.9 56c⟩
}
```

We create and store a count for each node. In the count we store the label of v and, if v isn't the root, also the label of its parent and the distance to it.

56b ⟨*Add count, Pr. 2.9* 56b⟩≡ (56a)
```
count := new(Count)
count.label = v.Label
if v.Parent != nil {
          count.dp = v.Length
          count.parent = v.Parent.Label
}
counts[v.Id] = count
```

If the current node is a leaf, we check its label, to make sure it is either a target or a neighbor. If the current node is not a leaf, it is an internal node. In that case we also check its label to make sure it actually has a label. In either case we count the targets and neighbors.

56c ⟨*Analyze v, Pr. 2.9* 56c⟩≡ (56a)
```
if v.Child == nil {
          ⟨Check leaf label, Pr. 2.9 56d⟩
} else {
          ⟨Check internal node label, Pr. 2.9 57a⟩
}
⟨Count targets and neighbors, Pr. 2.9 57c⟩
```

If the leaf label is either a target or a neighbor, we set its corresponding counter. Otherwise, something is wrong and we bail with message.

56d ⟨*Check leaf label, Pr. 2.9* 56d⟩≡ (56c)
```
if strings.HasPrefix(v.Label, np) {
          counts[v.Id].vn = 1.0
} else if strings.HasPrefix(v.Label, tp) {
          counts[v.Id].vt = 1.0
} else {
          log.Fatalf("%q is neither target nor neighbor",
                  v.Label)
}
```

We import `strings`.

We make sure all internal nodes are labeled.

57a ⟨*Check internal node label, Pr. 2.9* 57a⟩≡ (56c)
```
if v.Label == "" {
        log.Fatal("please label internal nodes " +
                "using land")
}
```

57b ⟨*Imports, Pr. 2.9* 54b⟩+≡ (53a) ◁55b 58a▷
```
"strings"
```

We count the targets and neighbors by incrementing the corresponding counter in the parent—unless we're a the root.

57c ⟨*Count targets and neighbors, Pr. 2.9* 57c⟩≡ (56c)
```
if v.Parent != nil {
        counts[v.Parent.Id].vt += counts[v.Id].vt
        counts[v.Parent.Id].vn += counts[v.Id].vn
}
```

We now have the raw counts in hand. From these we calculate the splits and sort the counts.

57d ⟨*Process counts, Pr. 2.9* 57d⟩≡ (55c)
⟨*Calculate splits, Pr. 2.9* 57e⟩
⟨*Sort counts, Pr. 2.9* 57g⟩

We calculate splits according to equation (2.1). First, we determine the total counts of targets and neighbors, n_t and n_n. These are the counts of targets and neighbors for the root.

57e ⟨*Calculate splits, Pr. 2.9* 57e⟩≡ (57d) 57f▷
```
nt := counts[tree.Id].vt
nn := counts[tree.Id].vn
```

We iterate over the counts. For each count we calculate the number of neighbors in the neighborhood,

$$v_\mathrm{n}^* = n_\mathrm{n} - v_\mathrm{n}.$$

Then we apply equation (2.1) to compute the split percentage.

57f ⟨*Calculate splits, Pr. 2.9* 57e⟩+≡ (57d) ◁57e
```
for _, count := range counts {
        van := nn - count.vn
        count.sv = float64(count.vt + van) /
                float64(nt + nn) * 100
}
```

We sort the counts by converting them to a slice and casting that to the sortable type `countsSlice`, which we still have to define.

57g ⟨*Sort counts, Pr. 2.9* 57g⟩≡ (57d)
```
cs := make([]*Count, 0)
for _, count := range counts {
        cs = append(cs, count)
}
sort.Sort(countsSlice(cs))
```

We import `sort`.

58a ⟨*Imports, Pr. 2.9* 54b⟩+≡ (53a) ◁57b 58e▷
```
"sort"
```

We declare the type `countsSlice`.

58b ⟨*Types, Pr. 2.9* 55e⟩+≡ (53a) ◁55e
```
type countsSlice []*Count
```

We implement the three methods of the sort interface, `Len`, `Less`, and `Swap`. In `Less` we sort in descending order by split. The order of nodes with the same split is stabilized by sorting in ascending order by label.

58c ⟨*Methods, Pr. 2.9* 58c⟩≡ (53a)
```
func (c countsSlice) Len() int {
	return len(c)
}
func (c countsSlice) Less(i, j int) bool {
	if c[i].sv == c[j].sv {
		return c[i].label < c[j].label
	}
	return c[i].sv > c[j].sv
}
func (c countsSlice) Swap(i, j int) {
	c[i], c[j] = c[j], c[i]
}
```

We print the counts, or rather, the maximal splits in a table, which we construct using a tab writer. If the user asked for all splits we also print the rest.

58d ⟨*Print counts, Pr. 2.9* 58d⟩≡ (55c)
```
w := tabwriter.NewWriter(os.Stdout, 0, 0, 2, ' ', 0)
⟨Print maximal splits, Pr. 2.9 59a⟩
if *optA {
	⟨Print remaining splits, Pr. 2.9 59c⟩
}
w.Flush()
```

We import `os` and `tabwriter`.

58e ⟨*Imports, Pr. 2.9* 54b⟩+≡ (53a) ◁58a 59b▷
```
"os"
"text/tabwriter"
```

We print the splits in six columns, clade, targets, neighbors, parent, distance to parent, and split. We print the headers for these three columns and then the maximal splits. These all have the same split percentage as the top scorer. The root has no parent, so we replace the root's parent label with a hyphen.

59a ⟨*Print maximal splits, Pr. 2.9* 59a⟩≡ (58d)
```
fmt.Fprint(w, "#Clade\tTargets\tNeighbors\tSplit (%)\t" +
        "Parent\tDist(Parent)\n")
i := 0
for ; i < len(cs) && cs[0].sv == cs[i].sv; i++ {
        pl := cs[i].parent
        if pl == "" {
                pl = "-"
        }
        fmt.Fprintf(w, "%s\t%d\t%d\t%.1f\t%s\t%g\n",
                cs[i].label, cs[i].vt, cs[i].vn,
                cs[i].sv, pl, cs[i].dp)
}
```

We import `fmt`.

59b ⟨*Imports, Pr. 2.9* 54b⟩+≡ (53a) ◁58e
```
"fmt"
```

We print the remaining splits.

59c ⟨*Print remaining splits, Pr. 2.9* 59c⟩≡ (58d)
```
for ; i < len(cs); i++ {
        pl := cs[i].parent
        if pl == "" {
                pl = "-"
        }
        fmt.Fprintf(w, "%s\t%d\t%d\t%.1f\t%s\t%g\n",
                cs[i].label, cs[i].vt, cs[i].vn,
                cs[i].sv, pl, cs[i].dp)
}
```

We have finished writing `fintac`, so let's test it.

Testing

Our code for testing `fintac` contains hooks for imports and the testing logic.

59d ⟨*fintac_test.go* 59d⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Pr. 2.9 60b⟩
)

func TestFintac(t *testing.T) {
        ⟨Testing, Pr. 2.9 60a⟩
}
```

We construct a set of tests then run them.

60a ⟨*Testing, Pr. 2.9* 60a⟩≡ (59d)
```
var tests []*exec.Cmd
⟨Construct tests, Pr. 2.9 60c⟩
for i, test := range tests {
	⟨Run test, Pr. 2.9 60e⟩
}
```

We import `exec`.

60b ⟨*Testing imports, Pr. 2.9* 60b⟩≡ (59d) 60f ▷
```
"os/exec"
```

Our first test runs on the test tree with no options set.

60c ⟨*Construct tests, Pr. 2.9* 60c⟩≡ (60a) 60d ▷
```
test := exec.Command("./fintac", "test.nwk")
tests = append(tests, test)
```

In our second and last test we request that all splits are printed.

60d ⟨*Construct tests, Pr. 2.9* 60c⟩+≡ (60a) ◁ 60c
```
test = exec.Command("./fintac", "-a", "test.nwk")
tests = append(tests, test)
```

For a given test we compare the results we get with the results we want, which are stored in files `r1.txt` and `r2.txt`.

60e ⟨*Run test, Pr. 2.9* 60e⟩≡ (60a)
```
get, err := test.Output()
if err != nil {
	t.Error(err)
}
name := "r" + strconv.Itoa(i+1) + ".txt"
want, err := os.ReadFile(name)
if err != nil {
	t.Error(err)
}
if !bytes.Equal(get, want) {
	t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `strconv`, `os`, and `bytes`.

60f ⟨*Testing imports, Pr. 2.9* 60b⟩+≡ (59d) ◁ 60b
```
"strconv"
"os"
"bytes"
```

2.10 `climt`

The program `climt` climbs a tree. It takes as input a phylogenetic tree and a node label and returns the node's chain of ancestors up to the root. For each node it also prints the length of the branch to the ancestor and the cumulative branch length. So its output consists of four columns, *Up*, *Node*, *Branch Length*, and *Cumulative Branch Length*. Like the output of `ants`, the output of `climt` starts at the root and ends at the target node. For example, if we climb in `test.nwk` from node 303, we reach the root in nine steps up the tree. The chain of ancestors consists of parent 295, grand parent 294, and so on, up to node 1, the root:

```
# Up    Node    Branch Length    Cumulative Branch Length
9       1       0                0.0120347
8       77      5.47e-05         0.01198
7       85      0.00406          0.00792
6       271     0.00128          0.00664
5       273     0.000212         0.006428
4       274     0.000425         0.006003
3       293     0.000132         0.005871
2       294     0.000526         0.005345
1       295     0.00488          0.000465
0       303     0.000465         0
```

Instead of climbing up the tree toward the root, the user can also opt to climb down. However, while climbing up goes all the way to the root, climbing down all the way to the leaves could lead to large output that is illegible. So we just climb down by one level to the direct children. Here's an example showing that in `test.nwk` node 295 has two children, node 303 as expected from above, and node 296:

```
# Parent   Children
295        296 303
```

Implementation

The outline of `climt` has hooks for imports, functions, and the logic of the main function.

Prog. 2.10 (`climt`)

61 ⟨*climt.go* 61⟩≡

```
  package main

  import (
            ⟨Imports, Pr. 2.10 62b⟩
  )

  ⟨Functions, Pr. 2.10 63c⟩
  func main() {
            ⟨Main function, Pr. 2.10 62a⟩
  }
```

In the main function we set the name of `climt` and its usage, declare the options, parse the options, and parse the input files.

62a ⟨*Main function, Pr. 2.10* 62a⟩≡ (61)
```
util.SetName("climt")
⟨Set usage, Pr. 2.10 62c⟩
⟨Declare options, Pr. 2.10 62e⟩
⟨Parse options, Pr. 2.10 62g⟩
⟨Parse input files, Pr. 2.10 63a⟩
```

We import `util`.

62b ⟨*Imports, Pr. 2.10* 62b⟩≡ (61) 62d ▷
```
"github.com/evolbioinf/neighbors/util"
```

The usage consists of three parts, the actual usage message, an explanation of the program's purpose, and an example command.

62c ⟨*Set usage, Pr. 2.10* 62c⟩≡ (62a)
```
u := "climt [option]... v [inputFile]..."
p := "Climb a phylogenetic tree starting at node v."
e := "climt someTaxon foo.nwk"
clio.Usage(u, p, e)
```

We import `clio`.

62d ⟨*Imports, Pr. 2.10* 62b⟩+≡ (61) ◁ 62b 62f ▷
```
"github.com/evolbioinf/clio"
```

We declare the obligatory version option. Apart from that, we also allow the user to switch from the default mode of climbing up the tree, that is, toward the root, to climbing down.

62e ⟨*Declare options, Pr. 2.10* 62e⟩≡ (62a)
```
optV := flag.Bool("v", false, "version")
optD := flag.Bool("d", false, "climb down one level")
```

We import `flag`.

62f ⟨*Imports, Pr. 2.10* 62b⟩+≡ (61) ◁ 62d 63b ▷
```
"flag"
```

We parse the options and respond to a request for the version as this stops `climt`.

62g ⟨*Parse options, Pr. 2.10* 62g⟩≡ (62a)
```
flag.Parse()
if *optV {
          util.PrintInfo("climt")
}
```

The next token on the command line is the name of the node from where we start climbing. If the user hasn't provided a node name, we bail with a friendly message. The remaining tokens on the command line are the names of input files. We parse each input file with the function `scan`, which takes as argument the name of the starting node and the "down" option.

63a ⟨*Parse input files, Pr. 2.10* 63a⟩≡ (62a)
```
args := flag.Args()
if len(args) == 0 {
        log.Fatal("please provide a starting node")
}
start := args[0]
files := args[1:]
clio.ParseFiles(files, scan, start, optD)
```

We import `log`.

63b ⟨*Imports, Pr. 2.10* 62b⟩+≡ (61) ◁62f 63d▷
```
"log"
```

Inside `scan` we first retrieve the name of the start node and the "down" option. Then we iterate over the trees in the current file. For each tree we determine its start node and climb from there.

63c ⟨*Functions, Pr. 2.10* 63c⟩≡ (61) 64a▷
```
func scan(r io.Reader, args ...interface{}) {
        start := args[0].(string)
        optD := args[1].(*bool)
        sc := nwk.NewScanner(r)
        for sc.Scan() {
                root := sc.Tree()
                ⟨Find start node, Pr. 2.10 63e⟩
                ⟨Climb tree, Pr. 2.10 64b⟩
        }
}
```

We import `io` and `nwk`.

63d ⟨*Imports, Pr. 2.10* 62b⟩+≡ (61) ◁63b 64c▷
```
"io"
"github.com/evolbioinf/nwk"
```

We search for the start node, v, using the function `findStart`. If we don't find the start node, we exit with return value 1, like `grep` does.

63e ⟨*Find start node, Pr. 2.10* 63e⟩≡ (63c)
```
var v *nwk.Node
findStart(root, &v, start)
if v == nil {
        os.Exit(1)
}
```

The function `findStart` traverses the tree recursively and analyzes each node, v.

64a ⟨*Functions, Pr. 2.10* 63c⟩+≡ (61) ◁63c
```
func findStart(root *nwk.Node, v **nwk.Node, start string) {
          if root == nil {
                  return
          }
          if root.Label == start {
                  *v = root
          }
          findStart(root.Child, v, start)
          findStart(root.Sib, v, start)
}
```

We climb either down or up the tree.

64b ⟨*Climb tree, Pr. 2.10* 64b⟩≡ (63c)
```
if *optD {
          ⟨Climb down, Pr. 2.10 64d⟩
} else {
          ⟨Climb up, Pr. 2.10 65d⟩
}
```

We import `fmt` and `os`.

64c ⟨*Imports, Pr. 2.10* 62b⟩+≡ (61) ◁63d 65a▷
```
"fmt"
"os"
```

When climbing down a tree, we get the children of v and print them.

64d ⟨*Climb down, Pr. 2.10* 64d⟩≡ (64b)
```
⟨Get children, Pr. 2.10 64e⟩
⟨Print children, Pr. 2.10 64f⟩
```

We store the children as a slice of nodes.

64e ⟨*Get children, Pr. 2.10* 64e⟩≡ (64d)
```
children := make([]*nwk.Node, 0)
np := v.Child
for np != nil {
          children = append(children, np)
          np = np.Sib
}
```

If we found any children, we print them as a table, which we format using a `tabwriter`. The table consists of a header and a row listing the children. Having printed the table, we flush the `tabwriter`.

64f ⟨*Print children, Pr. 2.10* 64f⟩≡ (64d)
```
if len(children) > 0 {
          w := tabwriter.NewWriter(os.Stdout, 0,
                  1, 3, ' ', 0)
          ⟨Print children header, Pr. 2.10 65b⟩
          ⟨Print children row, Pr. 2.10 65c⟩
          w.Flush()
}
```

We import the `tabwriter`.

65a ⟨*Imports, Pr. 2.10* 62b⟩+≡ (61) ◁64c 66c▷
```
"text/tabwriter"
```

In the header of our table we distinguish between one or more children.

65b ⟨*Print children header, Pr. 2.10* 65b⟩≡ (64f)
```
fmt.Fprint(w, "# Parent\tChild")
if len(children) > 1 {
          fmt.Fprint(w, "ren")
}
fmt.Fprint(w, "\n")
```

The child row consists of the name of the parent, followed by the names of its children separated by blanks.

65c ⟨*Print children row, Pr. 2.10* 65c⟩≡ (64f)
```
fmt.Fprintf(w, "%s\t", start)
for i, child := range children {
          if i > 0 {
                  fmt.Fprint(w, " ")
          }
          fmt.Fprintf(w, "%s", child.Label)
}
fmt.Fprint(w, "\n")
```

When climbing the tree, we encounter the nodes in the order child/parent, but we would like to print them in the order parent/child, so that on the screen the root is *up*. To get the printing order from the climbing order, we first collect all the ancestors in a slice. Then we calculate the cumulative length of their branches, before we print the ancestor table, again using a `tabwriter`.

65d ⟨*Climb up, Pr. 2.10* 65d⟩≡ (64b)
```
⟨Collect ancestors, Pr. 2.10 65e⟩
cumLen := v.UpDistance(root)
w := tabwriter.NewWriter(os.Stdout, 0, 1, 3, ' ', 0)
⟨Print ancestor header, Pr. 2.10 65f⟩
⟨Print ancestor table, Pr. 2.10 66a⟩
w.Flush()
```

Like the children, we collect the ancestors in a slice of nodes.

65e ⟨*Collect ancestors, Pr. 2.10* 65e⟩≡ (65d)
```
ancestors := make([]*nwk.Node, 0)
np := v
for np != nil {
          ancestors = append(ancestors, np)
          np = np.Parent
}
```

The ancestor table consists of four columns, the steps *up* the tree, the node label, the branch length, and the cumulative branch length.

65f ⟨*Print ancestor header, Pr. 2.10* 65f⟩≡ (65d)
```
fmt.Fprint(w, "# Up\tNode\tBranch Length\t" +
          "Cumulative Branch Length\n")
```

When printing the table of ancestors, we inverse the climbing order, so the root is in the first row and the starting node in the last. While iterating over the ancestors in inverse order, we repeatedly subtract the length of the upcoming branch from the current value of the cumulative length. The imprecision of floating point representation means that we cannot be sure that addition and subtraction are exactly reversible. To ensure that in our table we revisit the exact branch lengths of the values we originally put in, we round the cumulative lengths before printing them.

66a ⟨*Print ancestor table, Pr. 2.10* 66a⟩≡ (65d)
```
  n := len(ancestors)
  for i := n-1; i >= 0; i-- {
          ⟨Round cumulative length, Pr. 2.10 66b⟩
          ⟨Print cumulative length, Pr. 2.10 66d⟩
          if i > 0 {
                  cumLen -= ancestors[i-1].Length
          }
  }
```

We round a float, f, to s significant digits by multiplying by 10^s, rounding, and dividing again by 10^s,

$$f \leftarrow \frac{\text{round}(f \times 10^s)}{10^s}. \tag{2.2}$$

The question is, which value should we use for s? Since `float64` provides approximately 15 decimal digits of precision [1, p. 56], we use $s = 15$.

As a further complication, we might end up rounding to a negative zero. Since a negative zero is not less than zero, we can't test for negativity by testing for < 0. Instead, we use the dedicated function `Signbit`, which returns true for negative arguments, including negative zero.

66b ⟨*Round cumulative length, Pr. 2.10* 66b⟩≡ (66a)
```
  f := cumLen
  s := 15.0
  x := math.Pow(10, s)
  f = math.Round(f*x) / x
  if math.Signbit(f) {
          f *= -1.0
  }
```

We import `math`.

66c ⟨*Imports, Pr. 2.10* 62b⟩+≡ (61) ◁65a
```
  "math"
```

We print the steps back the label of the current ancestor, and its branch length, and the rounded cumulative branch length we just calculated.

66d ⟨*Print cumulative length, Pr. 2.10* 66d⟩≡ (66a)
```
  fmt.Fprintf(w, "%d\t%s\t%g\t%g\n",
          i,
          ancestors[i].Label,
          ancestors[i].Length,
          f)
```

We're done writing `climt`, so let's test it.

Testing

The outline of our testing code for `climt` has hooks for imports and the logic of the main function.

67a ⟨*climt_test.go* 67a⟩≡
```
package main

import (
          "testing"
          ⟨Testing imports, Pr. 2.10 67c⟩
)

func TestClimt(t *testing.T) {
          ⟨Testing, Pr. 2.10 67b⟩
}
```

We construct a set of tests and run each one in a loop.

67b ⟨*Testing, Pr. 2.10* 67b⟩≡ (67a)
```
tests := make([]*exec.Cmd, 0)
⟨Construct tests, Pr. 2.10 67d⟩
for i, test := range tests {
          ⟨Run test, Pr. 2.10 68a⟩
}
```

We import `exec`.

67c ⟨*Testing imports, Pr. 2.10* 67c⟩≡ (67a) 68b ▷
```
"os/exec"
```

We run our tests on the tree in file `test.nwk` with starting node 303. Using these two inputs, we construct two tests, one with only default options, the other with the "down" option, `-d`.

67d ⟨*Construct tests, Pr. 2.10* 67d⟩≡ (67b)
```
f := "test.nwk"
s := "303"
test := exec.Command("./climt", s , f)
tests = append(tests, test)
test = exec.Command("./climt", "-d", s, f)
tests = append(tests, test)
```

For a given test we compare the result we get with the result we want, which is stored in files `r1.txt` and `r2.txt`.

68a ⟨*Run test, Pr. 2.10* 68a⟩≡ (67b)

```
get, err := test.Output()
if err != nil {
          t.Error(err)
}
f := "r" + strconv.Itoa(i+1) + ".txt"
want, err := os.ReadFile(f)
if err != nil {
          t.Error(err)
}
if !bytes.Equal(get, want) {
          t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `strconv`, `os`, and `bytes`.

68b ⟨*Testing imports, Pr. 2.10* 67c⟩+≡ (67a) ◁67c

```
"strconv"
"os"
"bytes"
```

Chapter 3

Packages

3.1 tdb

The package tdb provides code for constructing and navigating tax.db (Figure 1.2). Its outline has hooks for imports, types, methods, and functions.

Package 3.1 (tdb)

70a ⟨*tdb.go* 70a⟩≡
```
// Package tdb constructs and queries the taxonomy database.
package tdb

import (
        ⟨Imports, Pa. 3.1 70c⟩
)
⟨Types, Pa. 3.1 70b⟩
⟨Methods, Pa. 3.1 78e⟩
⟨Functions, Pa. 3.1 70d⟩
```

The package tdb has two central types: TaxonDB wraps the relational database shown in Figure 1.2 and Taxonomy is the taxonomic tree.

70b ⟨*Types, Pa. 3.1* 70b⟩≡ (70a) 72d ▷
```
type TaxonomyDB struct {
        db *sql.DB
}
```

We import sql. In addition, we import a driver for sqlite3, go-sqlite3, using a blank import.

70c ⟨*Imports, Pa. 3.1* 70c⟩≡ (70a) 71b ▷
```
"database/sql"
_ "github.com/mattn/go-sqlite3"
```

3.1.1 NewTaxonomyDB

The function NewTaxonomyDB takes as parameters the names of the five input files from which we construct the database, and the name of the database. It opens these files, opens a new database, constructs the database, closes the files, and closes the database.

70d ⟨*Functions, Pa. 3.1* 70d⟩≡ (70a) 78d ▷
```
// NewTaxonomyDB takes as parameters the names
// of the five data files and the database name,
// and constructs the database from them.
func NewTaxonomyDB(nodes, names, prokaryotes,
        eukaryotes, viruses, dbName string) {
        ⟨Open files, Pa. 3.1 71a⟩
        ⟨Open database, Pa. 3.1 71c⟩
        ⟨Construct database, Pa. 3.1 71g⟩
        ⟨Close database, Pa. 3.1 78c⟩
        ⟨Close files, Pa. 3.1 73d⟩
}
```

We open the five input tables.

71a ⟨*Open files, Pa. 3.1* 71a⟩≡ (70d)
```
of := util.Open(nodes)
af := util.Open(names)
pf := util.Open(prokaryotes)
ef := util.Open(eukaryotes)
vf := util.Open(viruses)
```

We import `util`.

71b ⟨*Imports, Pa. 3.1* 70c⟩+≡ (70a) ◁70c 71d▷
```
"github.com/evolbioinf/neighbors/util"
```

Before opening the database, we check whether it already exists. If we cannot open the database, we abort.

71c ⟨*Open database, Pa. 3.1* 71c⟩≡ (70d)
```
⟨Does database exist? Pa. 3.1 71e⟩
db, err := sql.Open("sqlite3", dbName)
if err != nil { log.Fatal(err) }
```

We import `log`.

71d ⟨*Imports, Pa. 3.1* 70c⟩+≡ (70a) ◁71b 71f▷
```
"log"
```

If we are asked to build a new database on top of an existing one, the user has probably made a mistake, so we bow out.

71e ⟨*Does database exist? Pa. 3.1* 71e⟩≡ (71c)
```
_, err := os.Stat(dbName)
if err == nil {
          fmt.Fprintf(os.Stderr, "database %s already exists\n",
                  dbName)
          os.Exit(1)
}
```

We import `os` and `fmt`.

71f ⟨*Imports, Pa. 3.1* 70c⟩+≡ (70a) ◁71d 73c▷
```
"os"
"fmt"
```

To construct the database, we construct the input tables, and load their counterparts in the database.

71g ⟨*Construct database, Pa. 3.1* 71g⟩≡ (70d)
```
⟨Construct tables, Pa. 3.1 71h⟩
⟨Load tables, Pa. 3.1 72c⟩
```

As illustrated in Figure 1.2, we construct two tables, `taxon` and `genome`.

71h ⟨*Construct tables, Pa. 3.1* 71h⟩≡ (71g)
```
⟨Construct table taxon, Pa. 3.1 72a⟩
⟨Construct table genome, Pa. 3.1 72b⟩
```

Table `taxon` has the attributes `taxid`, `parent`, `name`, and `rank`. Taxon-IDs are integers, names are text. `taxid` is also the primary key. As there are currently 2.4 million taxa, and this number is bound to grow, we ensure that queries are quick by using indexes. The primary key, `taxid`, is automatically indexed and we construct an index for `parent`. We might be tempted to also index the remaining two attributes, `name` and `rank`. However, these are text and `sqlite` does not make use of indexes on text columns. You can check this on an existing database by prefacing a query with `explain query plan`. So we leave it at our index for `parent`.

72a ⟨*Construct table* `taxon`*, Pa. 3.1* 72a⟩≡ (71h)
```
sqlStmt := `create table taxon (
taxid int, parent int, name text, rank text,
primary key(taxid));
create index taxon_parent_idx on taxon(parent);`
if _, err := db.Exec(sqlStmt); err != nil {
        log.Fatal(err)
}
```

Table `genome` has attributes `taxid`, `size`, `replicons`, `accession`, and `status`. To stress that every genome belongs to exactly one taxon, we declare `taxid` as foreign key into `taxon`. We also index the numerical attributes `taxid` and `size`.

72b ⟨*Construct table* `genome`*, Pa. 3.1* 72b⟩≡ (71h)
```
sqlStmt = `create table genome (
taxid int, size real, replicons text,
         accession text, status text,
foreign key(taxid) references taxon(taxid));
create index genome_taxid_idx on genome(taxid);
create index genome_size_idx on genome(size);`
if _, err := db.Exec(sqlStmt); err != nil {
        log.Fatal(err)
}
```

Since `genome` has a foreign key referring to `taxon`, we load `taxon` before `genome`.

72c ⟨*Load tables, Pa. 3.1* 72c⟩≡ (71g)
⟨*Load table* `taxon`*, Pa. 3.1* 72e⟩
⟨*Load table* `genome`*, Pa. 3.1* 74c⟩

To load `taxon`, we simulate a join of `nodes.dmp` and `names.dmp` on taxon-ID. For this we create the type `taxon` with the same four fields as the four attributes in Figure 1.2.

72d ⟨*Types, Pa. 3.1* 70b⟩+≡ (70a) ◁70b 74d▷
```
type taxon struct {
        taxid, parent int
        name, rank string
}
```

We read the taxa into a map and insert them into the table `taxon`.

72e ⟨*Load table* `taxon`*, Pa. 3.1* 72e⟩≡ (72c)
```
taxa := make(map[int]*taxon)
```
⟨*Read taxa, Pa. 3.1* 73a⟩
⟨*Insert taxa, Pa. 3.1* 74a⟩

To read the taxa, we first parse the nodes file, then the names file.

73a ⟨*Read taxa, Pa. 3.1* 73a⟩≡ (72e)
  ⟨*Parse nodes file, Pa. 3.1* 73b⟩
  ⟨*Parse names file, Pa. 3.1* 73e⟩

We traverse the nodes file using a scanner. The first column contains the taxon-ID, the second the parent's ID, the third the taxon's rank. Columns are delimited by `\t|\t` and we convert the strings denoting IDs to integers before storing them.

73b ⟨*Parse nodes file, Pa. 3.1* 73b⟩≡ (73a)
```
scanner := bufio.NewScanner(of)
for scanner.Scan() {
          row := scanner.Text()
          fields := strings.SplitN(row, "\t|\t", 4)
          t := new(taxon)
          t.taxid, err = strconv.Atoi(fields[0])
          if err != nil { log.Fatal(err) }
          t.parent, err = strconv.Atoi(fields[1])
          if err != nil { log.Fatal(err) }
          t.rank = fields[2]
          taxa[t.taxid] = t
}
```

We import `bufio`, `strings`, and `strconv`.

73c ⟨*Imports, Pa. 3.1* 70c⟩+≡ (70a) ◁71f 79c▷
```
"bufio"
"strings"
"strconv"
```

We are done with the nodes file and close it.

73d ⟨*Close files, Pa. 3.1* 73d⟩≡ (70d) 73f▷
```
of.Close()
```

The names file contains the taxon-ID in the first column and the name in the second. Now, there are several kinds of names, scientific, common, and more. We store the scientific names. To parse the file, we again use a scanner and split the columns at `\t|\t`.

73e ⟨*Parse names file, Pa. 3.1* 73e⟩≡ (73a)
```
scanner = bufio.NewScanner(af)
for scanner.Scan() {
          row := scanner.Text()
          fields := strings.Split(row, "\t|\t")
          id, err := strconv.Atoi(fields[0])
          if err != nil { log.Fatal(err) }
          if fields[3][:3] == "sci" {
                  taxa[id].name = fields[1]
          }
}
```

We are done with the names file and close it.

73f ⟨*Close files, Pa. 3.1* 73d⟩+≡ (70d) ◁73d 75c▷
```
af.Close()
```

Having read the taxa from the input files, we insert them in the `taxon` table. We do this with a transaction, which takes a statement and prepares it, before we insert the individual rows.

74a ⟨*Insert taxa, Pa. 3.1* 74a⟩≡ (72e)
```
tx, err := db.Begin()
if err != nil { log.Fatal(err) }
sqlStmt = "insert into taxon(taxid, parent, name, rank) " +
          "values(?, ?, ?, ?)"
stmt, err := tx.Prepare(sqlStmt)
if err != nil { log.Fatal(err) }
⟨Insert rows into table taxon, Pa. 3.1 74b⟩
tx.Commit()
stmt.Close()
```

Each entry in the `taxa` map corresponds to a row in the table.

74b ⟨*Insert rows into table* `taxon`*, Pa. 3.1* 74b⟩≡ (74a)
```
for _, v := range taxa {
          _, err = stmt.Exec(v.taxid, v.parent, v.name, v.rank)
          if err != nil { log.Fatal(err) }
}
```

The next table to load is `genome`. Like the taxa, we read the genomes from input and insert them.

74c ⟨*Load table* `genome`*, Pa. 3.1* 74c⟩≡ (72c)
```
⟨Read genomes, Pa. 3.1 74e⟩
⟨Insert genomes, Pa. 3.1 78a⟩
```

We read each genome into a struct that holds the four attributes shown in Figure 1.2, `taxid`, `replicons`, `accession`, `status`, and `size`.

74d ⟨*Types, Pa. 3.1* 70b⟩+≡ (70a) ◁72d
```
type genome struct {
          taxid int
          replicons, accession, status string
          size float64
}
```

We save genomes in a slice and read them from the three files that correspond to the three deep divisions of life, prokaryotes, eukaryotes, and viruses.

74e ⟨*Read genomes, Pa. 3.1* 74e⟩≡ (74c)
```
var genomes []genome
⟨Read prokaryote genomes, Pa. 3.1 75a⟩
⟨Read eukaryote genomes, Pa. 3.1 76a⟩
⟨Read virus genomes, Pa. 3.1 77a⟩
```

Table 3.1: Columns in the table of prokaryote genomes supplied by the NCBI.

1	Organism/Name	9	Replicons	17	Center
2	TaxID	10	WGS	18	BioSample Accession
3	BioProject Accession	11	Scaffolds	19	Assembly Accession
4	BioProject ID	12	Genes	20	Reference
5	Group	13	Proteins	21	FTP Path
6	SubGroup	14	Release Date	22	Pubmed ID
7	Size (Mb)	15	Modify Date	23	Strain
8	GC%	16	Status		

In the prokaryote genomes file, we skip the header, split the columns at tabs, generate the genomes, and save them.

75a ⟨*Read prokaryote genomes, Pa. 3.1* 75a⟩≡ (74e)
```
fn := pf.Name()
scanner = bufio.NewScanner(pf)
var gen genome
for scanner.Scan() {
          row := scanner.Text()
          if row[0] == '#' { continue }
          fields := strings.Split(row, "\t")
          ⟨Generate a prokaryote genome, Pa. 3.1 75b⟩
          genomes = append(genomes, gen)
}
```

Table 3.1 shows the columns in the table of prokaryote genomes. We can see that the taxon ID is in the second column, the size in Mb in the seventh, the replicons in the nineth, the status in the 16th, and the assembly accessions in the 19th. Unfortunately, it is possible that there is no genome size, in which case we set it to -1. It is also possible that the row is incomplete, in which case we skip it and send a warning.

75b ⟨*Generate a prokaryote genome, Pa. 3.1* 75b⟩≡ (75a)
```
if len(fields) < 19 {
          fmt.Fprintf(os.Stderr,
                  "skipping truncated line in %q\n", fn)
          continue
}
gen.taxid, err = strconv.Atoi(fields[1])
if err != nil { log.Fatal(err) }
gen.size, err = strconv.ParseFloat(fields[6], 64)
if err != nil { gen.size = -1.0 }
gen.replicons = fields[8]
gen.status = fields[15]
gen.accession = fields[18]
```

We close the prokaryote genomes file.

75c ⟨*Close files, Pa. 3.1* 73d⟩+≡ (70d) ◁73f 76c▷
```
pf.Close()
```

Table 3.2: Columns in the table of eukaryote genomes supplied by the NCBI.

1	Organism/Name	8	GC%	14	Proteins
2	TaxID	9	Assembly Accession	15	Release Date
3	BioProject Accession	10	Replicons	16	Modify Date
4	BioProject ID	11	WGS	17	Status
5	Group	12	Scaffolds	18	Center
6	SubGroup	13	Genes	19	BioSample Accession
7	Size (Mb)				

Similarly, we read the file for eukaryote genomes.

76a ⟨*Read eukaryote genomes, Pa. 3.1* 76a⟩≡ (74e)
```
fn = ef.Name()
scanner = bufio.NewScanner(ef)
for scanner.Scan() {
          row := scanner.Text()
          if row[0] == '#' { continue }
          fields := strings.Split(row, "\t")
          ⟨Generate a eukaryote genome, Pa. 3.1 76b⟩
          genomes = append(genomes, gen)
}
```

Table 3.2 shows the columns in the table of eukaryote genomes. The taxon ID is in the second column, the size in the seventh, the assembly accession in the nineth, the replicons the tenth, and the status in the 17th.

76b ⟨*Generate a eukaryote genome, Pa. 3.1* 76b⟩≡ (76a)
```
if len(fields) < 10 {
          fmt.Fprintf(os.Stderr,
                  "skipping truncated line in %q\n", fn)
          continue
}
gen.taxid, err = strconv.Atoi(fields[1])
if err != nil { log.Fatal(err) }
gen.size, err = strconv.ParseFloat(fields[6], 64)
if err != nil { gen.size = -1.0 }
gen.accession = fields[8]
gen.replicons = fields[9]
gen.status = fields[16]
```

We close the eukaryote genomes file.

76c ⟨*Close files, Pa. 3.1* 73d⟩+≡ (70d) ◁75c 77c▷
```
ef.Close()
```

Table 3.3: Columns in the table of viral genomes supplied by the NCBI.

1	Organism/Name	6	SubGroup	11	Genes
2	TaxID	7	Size (Kb)	12	Proteins
3	BioProject Accession	8	GC%	13	Release Date
4	BioProject ID	9	Host	14	Modify Date
5	Group	10	Segments	15	Status

We scan the file of viral genomes.

77a ⟨*Read virus genomes, Pa. 3.1* 77a⟩≡ (74e)
```
fn = vf.Name()
scanner = bufio.NewScanner(vf)
for scanner.Scan() {
        row := scanner.Text()
        if row[0] == '#' { continue }
        fields := strings.Split(row, "\t")
        ⟨Generate a viral genome, Pa. 3.1 77b⟩
        genomes = append(genomes, gen)
}
```

Table 3.3 shows the columns in the table of viral genomes. Again, the taxon ID is in the second column and the size in the seventh. However, this time the size is in kb, which we convert to Mb. Also, there are no “Replicons”, nor is there an “Assembly Accession”. Instead, we have “Segments” in column 10, which we use both as attributes `replicons` and `accession` in our database. The status is in column 15.

77b ⟨*Generate a viral genome, Pa. 3.1* 77b⟩≡ (77a)
```
if len(fields) < 10 {
        fmt.Fprintf(os.Stderr,
                "skipping truncated line in %q", fn)
        continue
}
gen.taxid, err = strconv.Atoi(fields[1])
if err != nil { log.Fatal(err) }
gen.size, err = strconv.ParseFloat(fields[6], 64)
if err != nil { gen.size = -1.0 }
if gen.size > 0 { gen.size /= 1000.0 }
gen.replicons = fields[9]
gen.accession = fields[9]
gen.status = fields[14]
```

We close the virus genomes file.

77c ⟨*Close files, Pa. 3.1* 73d⟩+≡ (70d) ◁76c
```
vf.Close()
```

To insert the genomes into the `genomes` table, we use a transaction like we did for the `taxon` table.

78a ⟨*Insert genomes, Pa. 3.1* 78a⟩≡ (74c)
```
tx, err = db.Begin()
if err != nil { log.Fatal(err) }
sqlStmt = "insert into genome(taxid, replicons," +
          "size, accession, status) " +
          "values(?, ?, ?, ?, ?)"
stmt, err = tx.Prepare(sqlStmt)
if err != nil { log.Fatal(err) }
⟨Insert rows into table genome, Pa. 3.1 78b⟩
tx.Commit()
stmt.Close()
```

We insert each entry in the `genomes` slice into table `genome`.

78b ⟨*Insert rows into table* `genome`*, Pa. 3.1* 78b⟩≡ (78a)
```
for _, g := range genomes {
          _, err = stmt.Exec(g.taxid, g.replicons,
                  g.size, g.accession, g.status)
          if err != nil { log.Fatal(err) }
}
```

The database is constructed, so we close it.

78c ⟨*Close database, Pa. 3.1* 78c⟩≡ (70d)
```
db.Close()
```

3.1.2 OpenTaxonomyDB

The function `OpenTaxonomyDB` opens an existing taxonomy database and returns a pointer to it.

78d ⟨*Functions, Pa. 3.1* 70d⟩+≡ (70a) ◁70d 84b▷
```
// OpenTaxonomyDB opens an existing taxonomy database and returns a
// pointer to it.
func OpenTaxonomyDB(name string) *TaxonomyDB {
          db := new(TaxonomyDB)
          var err error
          db.db, err = sql.Open("sqlite3", name)
          if err != nil { log.Fatal(err) }
          return db
}
```

3.1.3 Close

The method `Close` closes a taxonomy database.

78e ⟨*Methods, Pa. 3.1* 78e⟩≡ (70a) 79a▷
```
// Close closes the taxonomy database.
func (t *TaxonomyDB) Close() {
          t.db.Close()
}
```

3.1.4 Replicons

The method Replicons takes as parameter a taxon-ID and returns a slice of replicons.

We query for replicons and then store them in the string slice we return.

79a ⟨*Methods, Pa. 3.1* 78e⟩+≡ (70a) ◁78e 80a▷

```
func (t *TaxonomyDB) Replicons(tid int) []string {
          var reps []string
          ⟨Query for replicions, Pa. 3.1 79b⟩
          ⟨Store replicons, Pa. 3.1 79d⟩
          return reps
}
```

We generate the query and execute it, which returns a set of table rows. The rows are later closed again.

79b ⟨*Query for replicions, Pa. 3.1* 79b⟩≡ (79a)

```
tmpl := "select replicons from genome where taxid=%d " +
          "and replicons <> '-'"
q := fmt.Sprintf(tmpl, tid)
rows, err := t.db.Query(q)
if err != nil {
          log.Fatal(err)
}
defer rows.Close()
```

We import fmt.

79c ⟨*Imports, Pa. 3.1* 70c⟩+≡ (70a) ◁73c

```
"fmt"
```

We append each replicon to our slice of replicons.

79d ⟨*Store replicons, Pa. 3.1* 79d⟩≡ (79a)

```
s := ""
for rows.Next() {
          err := rows.Scan(&s)
          if err != nil {
                  log.Fatal(err)
          }
          reps = append(reps, s)
}
```

3.1.5 Accessions

The method Accessions takes as parameter a taxon-ID and returns a slice of assembly accessions.

We query for accessions and then store them in the string slice we return.

80a ⟨*Methods, Pa. 3.1* 78e⟩+≡ (70a) ◁79a 80d▷
```
func (t *TaxonomyDB) Accessions(tid int) []string {
          var accessions []string
          ⟨Query for accessions, Pa. 3.1 80b⟩
          ⟨Store accessions, Pa. 3.1 80c⟩
          return accessions
}
```

We generate the query and execute it, which returns a set of table rows. The rows are later closed again.

80b ⟨*Query for accessions, Pa. 3.1* 80b⟩≡ (80a)
```
tmpl := "select accession from genome where taxid=%d"
q := fmt.Sprintf(tmpl, tid)
rows, err := t.db.Query(q)
if err != nil {
          log.Fatal(err)
}
defer rows.Close()
```

We append each accession to our slice of replicons.

80c ⟨*Store accessions, Pa. 3.1* 80c⟩≡ (80a)
```
accession := ""
for rows.Next() {
          err := rows.Scan(&accession)
          if err != nil {
                  log.Fatal(err)
          }
          accessions = append(accessions, accession)
}
```

3.1.6 Name

The method Name takes as argument a taxon ID and returns the taxon's name. We construct the query, execute it, and extract the name.

80d ⟨*Methods, Pa. 3.1* 78e⟩+≡ (70a) ◁80a 81d▷
```
// Name returns a taxon's name.
func (t *TaxonomyDB) Name(taxon int) string {
          n := ""
          ⟨Construct name query, Pa. 3.1 81a⟩
          ⟨Execute name query, Pa. 3.1 81b⟩
          ⟨Extract name, Pa. 3.1 81c⟩
          return n
}
```

We construct the query from a string template.

81a ⟨*Construct name query, Pa. 3.1* 81a⟩≡ (80d)
```
tmpl := "select name from taxon where taxid=%d"
q := fmt.Sprintf(tmpl, taxon)
```

We execute the query, which might throw an error. We also close the results table once we're done with it.

81b ⟨*Execute name query, Pa. 3.1* 81b⟩≡ (80d)
```
rows, err := t.db.Query(q)
if err != nil {
          log.Fatal(err)
}
defer rows.Close()
```

Our results table contains only the requested single name. We scan this and catch potential errors.

81c ⟨*Extract name, Pa. 3.1* 81c⟩≡ (80d)
```
rows.Next()
err = rows.Scan(&n)
if err != nil {
          log.Fatal(err)
}
```

3.1.7 Rank

The method Rank takes as argument a taxon ID and returns the taxon's name. We construct the query, execute it, and extract the name.

81d ⟨*Methods, Pa. 3.1* 78e⟩+≡ (70a) ◁80d 82b▷
```
func (t *TaxonomyDB) Rank(taxon int) string {
          rank := ""
          ⟨Construct rank query, Pa. 3.1 81e⟩
          ⟨Execute rank query, Pa. 3.1 81f⟩
          ⟨Extract rank, Pa. 3.1 82a⟩
          return rank
}
```

We construct the rank query from a string template.

81e ⟨*Construct rank query, Pa. 3.1* 81e⟩≡ (81d)
```
tmpl := "select rank from taxon where taxid=%d"
q := fmt.Sprintf(tmpl, taxon)
```

We execute the rank query, which might throw an error. We also close the results table once we're done with it.

81f ⟨*Execute rank query, Pa. 3.1* 81f⟩≡ (81d)
```
rows, err := t.db.Query(q)
if err != nil {
          log.Fatal(err)
}
defer rows.Close()
```

Our results table contains only the requested single rank. We scan this and catch potential errors.

82a ⟨*Extract rank, Pa. 3.1* 82a⟩≡ (81d)

```
rows.Next()
err = rows.Scan(&rank)
if err != nil {
          log.Fatal(err)
}
```

3.1.8 Parent

The method `Parent` takes as argument a taxon ID and returns the taxon ID of its parent. We construct the query, execute it, and extract the parent.

82b ⟨*Methods, Pa. 3.1* 78e⟩+≡ (70a) ◁81d 83a▷

```
// Parent returns a taxon's parent.
func (t *TaxonomyDB) Parent(c int) int {
          p := 0
          ⟨Construct parent query, Pa. 3.1 82c⟩
          ⟨Execute parent query, Pa. 3.1 82d⟩
          ⟨Extract parent, Pa. 3.1 82e⟩
          return p
}
```

We construct the parent query from a string template.

82c ⟨*Construct parent query, Pa. 3.1* 82c⟩≡ (82b)

```
tmpl := "select parent from taxon where taxid=%d"
q := fmt.Sprintf(tmpl, c)
```

We execute the query to get the results table. Query execution might throw an error, which we catch. We also make sure the results table is eventually closed again.

82d ⟨*Execute parent query, Pa. 3.1* 82d⟩≡ (82b)

```
rows, err := t.db.Query(q)
if err != nil {
          log.Fatal(err)
}
defer rows.Close()
```

Our results table contains at most one row with one entry, the parent. We extract this and catch possible errors.

82e ⟨*Extract parent, Pa. 3.1* 82e⟩≡ (82b)

```
rows.Next()
err = rows.Scan(&p)
if err != nil {
          log.Fatal(err)
}
```

3.1.9 Children

The method `Children` takes as argument a taxon ID and returns its children. We construct the children query, execute it, and extract the children. The children are stored in an integer slice, which we construct at the start of the method and return at the end.

83a ⟨*Methods, Pa. 3.1* 78e⟩+≡ (70a) ◁82b 84a▷

```
// Children returns a taxon's children.
func (t *TaxonomyDB) Children(p int) []int {
          c := make([]int, 0)
          ⟨Construct children query, Pa. 3.1 83b⟩
          ⟨Execute children query, Pa. 3.1 83c⟩
          ⟨Extract children, Pa. 3.1 83d⟩
          return c
}
```

Like the parent query, we construct the children query from a string template.

83b ⟨*Construct children query, Pa. 3.1* 83b⟩≡ (83a)

```
tmpl := "select taxid from taxon where parent=%d"
q := fmt.Sprintf(tmpl, p)
```

We execute the children query to get the results table, catch errors, and eventually close the results table again.

83c ⟨*Execute children query, Pa. 3.1* 83c⟩≡ (83a)

```
rows, err := t.db.Query(q)
if err != nil {
          log.Fatal(err)
}
defer rows.Close()
```

We copy the children into the slice we've prepared for this purpose.

83d ⟨*Extract children, Pa. 3.1* 83d⟩≡ (83a)

```
x := 0
for rows.Next() {
          err = rows.Scan(&x)
          if err != nil {
                  log.Fatal(err)
          }
          c = append(c, x)
}
```

3.1.10 Subtree

The method Subtree returns all taxa in a subtree, including its root. It does this by calling the private function traverseSubtree.

84a ⟨*Methods, Pa. 3.1* 78e⟩+≡ (70a) ◁83a 84c▷

```
// Subtree returns the taxa in the subtree rooted on the given taxon.
func (t *TaxonomyDB) Subtree(r int) []int {
          taxa := make([]int, 0)
          taxa = traverseSubtree(t, r, taxa)
          return taxa
}
```

traverseSubtree is a recursive function, where we take care to avoid the infinite loop we would get if we included the root in the recursion.

84b ⟨*Functions, Pa. 3.1* 70d⟩+≡ (70a) ◁78d

```
func traverseSubtree(t *TaxonomyDB, r int, taxa []int) []int {
          taxa = append(taxa, r)
          ch := t.Children(r)
          for _, c := range ch {
                  if c != r {
                          taxa = traverseSubtree(t, c, taxa)
                  }
          }
          return taxa
}
```

3.1.11 Taxids

Given a taxon name, Taxids returns the corresponding taxon-IDs. We construct and execute the query for taxon-IDs, extract the IDs, and return them.

84c ⟨*Methods, Pa. 3.1* 78e⟩+≡ (70a) ◁84a 85c▷

```
// Taxids matches the name of a taxon and returns the corresponding
// taxon-IDs.
func (t *TaxonomyDB) Taxids(name string) []int {
          taxids := make([]int, 0)
          ⟨Construct taxids query, Pa. 3.1 84d⟩
          ⟨Execute taxids query, Pa. 3.1 85a⟩
          ⟨Extract taxids, Pa. 3.1 85b⟩
          return taxids
}
```

We construct the query for taxon-IDs.

84d ⟨*Construct taxids query, Pa. 3.1* 84d⟩≡ (84c)

```
q := "select taxid from taxon where name like '%s'"
q = fmt.Sprintf(q, name)
```

We execute the query for taxon-IDs and catch potential errors. We also make sure the result table is eventually closed again.

85a ⟨*Execute taxids query, Pa. 3.1* 85a⟩≡ (84c)
```
rows , err := t.db.Query(q)
if err != nil {
          log.Fatal(err)
}
defer rows.Close()
```

We extract and store the taxon-IDs.

85b ⟨*Extract taxids, Pa. 3.1* 85b⟩≡ (84c)
```
taxid := 0
for rows.Next() {
          err = rows.Scan(&taxid)
          if err != nil {
                  log.Fatal(err)
          }
          taxids = append(taxids, taxid)
}
```

3.1.12 `MRCA`

The method `MRCA` takes as input a slice of taxon IDs and returns their most recent common ancestor. For example, in Figure 1.1, the most recent common ancestor of taxa 4 and 7 is 3. We begin by checking the IDs supplied and then search for the MRCA. If we haven't found one, we return -1.

85c ⟨*Methods, Pa. 3.1* 78e⟩+≡ (70a) ◁84c
```
func (t *TaxonomyDB) MRCA(ids []int) int {
          mrca := -1
          ⟨Check IDs, Pa. 3.1 85d⟩
          ⟨Search for MRCA, Pa. 3.1 86a⟩
          return mrca
}
```

If the ID list is empty, something went wrong and we throw an error. If the ID list contains a single entry, that is the MRCA.

85d ⟨*Check IDs, Pa. 3.1* 85d⟩≡ (85c)
```
if len(ids) == 0 {
          log.Fatal("Empty ID list in tdb.MRCA")
} else if len(ids) == 1 {
          return ids[0]
}
```

There is a rich history of MRCA queries in computer science [4, ch. 3]. However, instead of using the general solution to the problem, I construct a simpler solution that involves climbing to the common ancestor of a set of taxa. If these taxa are closely related, the climb is short.

To be more precise, in each step of our climb, we increment a descendant counter for each node. If one of these counters is equal to the number of input taxa, the corresponding node is the MRCA. Otherwise, we climb to the parents and repeat. So we construct a variable to count descendants and variables for parents and children. Then we iterate over the parents.

86a ⟨*Search for MRCA, Pa. 3.1* 86a⟩≡ (85c)

```
⟨Construct counter for descendants, Pa. 3.1 86b⟩
⟨Construct variables for parents and children, Pa. 3.1 86c⟩
⟨Iterate over parents, Pa. 3.1 86d⟩
```

We store descendants in a map between a taxon ID, which is an integer, and the number of its descendants, another integer. The initial number of descendants of the taxa supplied is one.

86b ⟨*Construct counter for descendants, Pa. 3.1* 86b⟩≡ (86a)

```
desc := make(map[int]int)
for _, id := range ids {
        desc[id] = 1
}
```

We store the parent and child taxa as integer slices. The children slice is initialized to the taxon IDs supplied.

86c ⟨*Construct variables for parents and children, Pa. 3.1* 86c⟩≡ (86a)

```
parents := make([]int, 0)
children := make([]int, 0)
for _, id := range ids {
        children = append(children, id)
}
```

As long as we have more than a single child, i. e. haven't reached the root yet, we climb to the next set of parents and we replace the children by the parents.

86d ⟨*Iterate over parents, Pa. 3.1* 86d⟩≡ (86a)

```
for len(children) > 1 {
        ⟨Climb to parents, Pa. 3.1 86e⟩
        ⟨Replace parents by children, Pa. 3.1 87a⟩
}
```

For each parent we reach, we check whether the number of its descendants is equal to the number of taxa submitted. If yes, we've found the MRCA and return it.

86e ⟨*Climb to parents, Pa. 3.1* 86e⟩≡ (86d)

```
for _, child := range children {
        parent := t.Parent(child)
        desc[parent] += desc[child]
        if desc[parent] == len(ids) {
                mrca = parent
                break
        }
        parents = append(parents, parent)
}
```

If the MRCA hasn't been found yet, we replace the children by the parent and reset the parent slice to empty.

87a ⟨*Replace parents by children, Pa. 3.1* 87a⟩≡ (86d)
```
if mrca == -1 {
        children = children[:0]
        for _, parent := range parents {
                children = append(children, parent)
        }
        parents = parents[:0]
} else {
        break
}
```

3.1.13 Testing

The outline of our code for testing `tdb` has a hook for the testing logic.

87b ⟨*tdb_test.go* 87b⟩≡
```
package tdb

import (
        "testing"
)

func TestTdb(t *testing.T) {
        ⟨Testing, Pa. 3.1 87c⟩
}
```

We test the function `NewTaxonomyDB`, and the methods `Replicons`, `Name`, `Parent`, `Children`, `Subtree`, and `Taxids`.

87c ⟨*Testing, Pa. 3.1* 87c⟩≡ (87b)
```
⟨Test NewTaxonomyDB, Pa. 3.1 87d⟩
⟨Test Replicons, Pa. 3.1 88b⟩
⟨Test Name, Pa. 3.1 88c⟩
⟨Test Parent, Pa. 3.1 88d⟩
⟨Test Children, Pa. 3.1 88e⟩
⟨Test Subtree, Pa. 3.1 89a⟩
⟨Test Taxids, Pa. 3.1 89b⟩
```

We build a small database, `taxSmall.db`.

87d ⟨*Test* `NewTaxonomyDB`*, Pa. 3.1* 87d⟩≡ (87c) 88a ▷
```
p := "../data/"
no := p + "nodesTest.dmp"
na := p + "namesTest.dmp"
pr := p + "prokaryotes.txt"
eu := p + "eukaryotes.txt"
vi := p + "viruses.txt"
d1 := p + "taxSmall.db"
NewTaxonomyDB(no, na, pr, eu, vi, d1)
```

We extract the full taxonomy from the small database and check it has 11 entries.

88a ⟨*Test* NewTaxonomyDB, *Pa. 3.1* 87d⟩+≡ (87c) ◁87d
```
taxdb := OpenTaxonomyDB(d1)
subtree := taxdb.Subtree(1)
if len(subtree) != 11 {
          t.Errorf("get %d rows, want 11", len(subtree))
          for _, s := range subtree {
                  println(s)
          }
}
```

We open a complete taxonomy, submit three taxon-IDs to the method Replicons, and print out the result.

88b ⟨*Test* Replicons, *Pa. 3.1* 88b⟩≡ (87c)
```
d2 := p + "tax.db"
taxdb = OpenTaxonomyDB(d2)
tid := 866775
reps := taxdb.Replicons(tid)
get := reps[0]
want := "chromosome:NC_015278.1/CP002512.1"
if get != want {
          t.Errorf("get: %q; want: %q", get, want)
}
```

We look up the name of taxon, 9606, *Homo sapiens*.

88c ⟨*Test* Name, *Pa. 3.1* 88c⟩≡ (87c)
```
tid = 9606
want = "Homo sapiens"
get = taxdb.Name(tid)
if get != want {
          t.Errorf("get: %q; want: %q", get, want)
}
```

The parent of *Homo sapiens* is *Homo* with taxon-ID 9605.

88d ⟨*Test* Parent, *Pa. 3.1* 88d⟩≡ (87c)
```
w := 9605
g := taxdb.Parent(tid)
if g != w {
          t.Errorf("get parent: %d; want: %d", g, w)
}
```

Homo sapiens (9606) has two children.

88e ⟨*Test* Children, *Pa. 3.1* 88e⟩≡ (87c)
```
w = 2
g = len(taxdb.Children(tid))
if g != w {
          t.Errorf("get %d children; want %d", g, w)
}
```

The *Hominidae* subtree contains 26 nodes.

89a ⟨*Test* `Subtree`*, Pa. 3.1* 89a⟩≡ (87c)
```
tid = 207598
w = 26
taxa := taxdb.Subtree(tid)
g = len(taxa)
if g != w {
          t.Errorf("get %d nodes in subtree; want %d", g, w)
}
```

To test `Taxids`, we get the ten taxa whose names contain *homo sapiens*, of which there are 11. Notice the lower-case “h” in *homo*, which SQL matches to the upper-case “H” in the actual names.

89b ⟨*Test* `Taxids`*, Pa. 3.1* 89b⟩≡ (87c)
```
w = 11
taxa = taxdb.Taxids("%homo sapiens%")
g = len(taxa)
if g != w {
          t.Errorf("get %d taxa for homo sapiens; want %d", g, w)
}
```

To test `MRCA`, we construct several tests and run them in a loop.

89c ⟨*Test* `MRCA`*, Pa. 3.1* 89c⟩≡
  ⟨*Construct tests of* `MRCA`*, Pa. 3.1* 89d⟩
  ⟨*Run tests of* `MRCA`*, Pa. 3.1* 90⟩

We look for the MRCAs of the following five sets of taxa in the tree shown in Figure 1.1:

- $\{4, 7\}$
- $\{4\}$
- $\{2, 7\}$
- $\{2, 2\}$
- $\{4, 7, 6\}$

89d ⟨*Construct tests of* `MRCA`*, Pa. 3.1* 89d⟩≡ (89c)
```
taxa := make([][]int, 0)
var res []int
taxa = append(taxa, []int{4, 7})
res = append(res, 3)
taxa = append(taxa, []int{4})
res = append(res, 4)
taxa = append(taxa, []int{2, 7})
res = append(res, 1)
taxa = append(taxa, []int{2, 2})
res = append(res, 2)
taxa = append(taxa, []int{4, 7, 6})
res = append(res, 3)
```

We loop over the tests.

90 ⟨*Run tests of* `MRCA`*, Pa. 3.1* 90⟩≡ (89c)

```
for i, taxon := range taxa {
          get := tax.MRCA(taxon)
          want := res[i]
          if get != want {
                  t.Errorf("get: %d\nwant: %d\n", get, want)
          }
}
```

3.2 util

The package util collects utility functions. Its outline provides hooks for imports, variables, and functions.

Package 3.2 (util)

91a ⟨*util.go* 91a⟩≡
```
// Package util provides utility functions for the programs
// indexNeighbors and neighbors.
package util

import (
        ⟨Imports, Pa. 3.2 91d⟩
)
⟨Variables, Pa. 3.2 91c⟩
⟨Functions, Pa. 3.2 91b⟩
```

3.2.1 PrintInfo

PrintInfo prints program information and exits.

91b ⟨*Functions, Pa. 3.2* 91b⟩≡ (91a) 92a ▷
```
func PrintInfo(program string) {
        author := "Bernhard Haubold"
        email := "haubold@evolbio.mpg.de"
        license := "Gnu General Public License, " +
                "https://www.gnu.org/licenses/gpl.html"
        clio.PrintInfo(program, version, date,
                author, email, license)
        os.Exit(0)
}
```

We declare the variables version and date, which ought to be injected at compile time.

91c ⟨*Variables, Pa. 3.2* 91c⟩≡ (91a) 92f ▷
```
var version, date string
```

We import clio and os.

91d ⟨*Imports, Pa. 3.2* 91d⟩≡ (91a) 92b ▷
```
"github.com/evolbioinf/clio"
"os"
```

3.2.2 Open

Open opens a file with error checking.

92a ⟨*Functions, Pa. 3.2* 91b⟩+≡ (91a) ◁91b 92c▷
```
func Open(file string) *os.File {
          f, err := os.Open(file)
          if err != nil {
                  fmt.Fprintf(os.Stderr, "couldn't open %s\n", file)
                  os.Exit(1)
          }
          return f
}
```

We import `fmt` and `os`.

92b ⟨*Imports, Pa. 3.2* 91d⟩+≡ (91a) ◁91d 92d▷
```
"fmt"
"os"
```

Check

Check checks an error and aborts if it isn't nil.

92c ⟨*Functions, Pa. 3.2* 91b⟩+≡ (91a) ◁92a 92e▷
```
func Check(err error) {
          if err != nil {
                  log.Fatal(err)
          }
}
```

We import `log`.

92d ⟨*Imports, Pa. 3.2* 91d⟩+≡ (91a) ◁92b
```
"log"
```

3.2.3 SetName

The function `SetName` sets the name of the program. It stores the name in a global variable and prepares the `log` package to print that name in the event of an error message.

92e ⟨*Functions, Pa. 3.2* 91b⟩+≡ (91a) ◁92c 93a▷
```
func SetName(n string) {
          name = n
          s := fmt.Sprintf("%s: ", n)
          log.SetPrefix(s)
          log.SetFlags(0)
}
```

We declare the global string variable `name`.

92f ⟨*Variables, Pa. 3.2* 91c⟩+≡ (91a) ◁91c
```
var name string
```

3.2.4 `Version`

The function `Version` prints the version and other information about the program and exits. `Version` simply wraps a call to `PrintInfo`.

93a ⟨*Functions, Pa. 3.2* 91b⟩+≡ (91a) ◁92e

```
func Version() {
        PrintInfo(name)
}
```

We are done with the `util` package, time to test it.

3.2.5 Testing

Our testing code for `util` contains hooks for imports and the logic of the testing function.

93b ⟨*util_test.go* 93b⟩≡

```
package util

import (
        "testing"
        ⟨Testing imports, Pa. 3.2 93d⟩
)

func TestUtil(t *testing.T) {
        ⟨Testing, Pa. 3.2 93c⟩
}
```

There is only one function we can sensibly test, `Open`. So we open a test file and read the string "success" from it.

93c ⟨*Testing, Pa. 3.2* 93c⟩≡ (93b)

```
f := Open("r.txt")
defer f.Close()
sc := bufio.NewScanner(f)
if !sc.Scan() {
        t.Error("scan failed")
}
get := sc.Text()
want := "success"
if get != want {
        t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `bufio`.

93d ⟨*Testing imports, Pa. 3.2* 93d⟩≡ (93b)

```
"bufio"
```

Chapter 4

Tutorial

Introduction

The Neighbors package provides tools for finding the target and neighbor genomes used to identify regions that are ubiquitous in the targets and absent from the neighbors. These target regions are a good starting point for constructing diagnostic markers. In this Tutorial I demonstrate the application of the nine programs in the Neighbors package listed in Table 1.1. We begin by constructing a local relational database from the NCBI taxonomy using the program `makeNeiDb`. Then we query the database with `taxid` to find the taxon-ID for our target organism, serotype O157:H7 of *Escherichia coli*. The programs `ants` and `dree` allow us to explore the taxonomic context of our target organism. Based on the target taxon-ID, we carry out the search for target and neighbor genomes using `neighbors`.

Any genomes suggested by `neighbors` are best subjected to phylogeny reconstruction before we extract marker candidates from them. The prerequisite phylogeny reconstruction is done with `phylonium`[1]. The resulting tree can be very large and its analysis is supported by three additional programs from the Neighbors package, `land` for labeling the nodes in a phylogeny, `pickle` for picking a particular clade in a phylogeny, and `fintac` for finding the target clade. Actual marker discovery is done with `fur`[2].

Construct Database

A dump of the current NCBI taxonomy database is posted as a tar ball at

`ftp.ncbi.nlm.nih.gov/pub/taxonomy/taxdump.tar.gz`

The corresponding genome reports for prokaryotes, eukaryotes, and viruses are posted at

```
ftp.ncbi.nlm.nih.gov/genomes/GENOME_REPORTS/prokaryotes.txt
ftp.ncbi.nlm.nih.gov/genomes/GENOME_REPORTS/eukaryotes.txt
ftp.ncbi.nlm.nih.gov/genomes/GENOME_REPORTS/viruses.txt
```

The database dump and the genome reports are updated regularly. To ensure the stability of this Tutorial, we use files downloaded on 21st September 2023:

95a ⟨*db.sh* 95a⟩≡ 95b ▷
```
  wget guanine.evolbio.mpg.de/neighbors/neidb_210923.tgz
```

We unpack the database files, delete the tar file, and change into the new directory `neidb_210923`.

95b ⟨*db.sh* 95a⟩+≡ ◁ 95a 96a ▷
```
  tar -xvzf neidb_210923.tgz
  rm neidb_210923.tgz
  cd neidb_210923
```

[1]`https://github.com/evolbioinf/phylonium`
[2]`https://github.com/evolbioinf/fur`

We can now construct the taxonomy database, `neidb`, using `makeNeiDb`. This takes approximately 43s. Then we return to the parent directory, move the database `neidb` there, and remove the directory used in its construction.

96a ⟨*db.sh* 95a⟩+≡ ◁95b
```
  makeNeiDb
  cd ../
  mv neidb_210923/neidb .
  rm -r neidb_210923
```

Query Database

The next step is to find the taxon-IDs for one or more target organisms. Our target organism is the bactrium *E. coli* O157:H7, a cause of severe diarrhea in humans. We look up its taxon-ID using `taxi` in substring mode (`-s`). For each taxon we get the ID, the parent ID, and the name.

96b ⟨*query.sh* 96b⟩≡ 96c▷
```
  taxi -s "O157:H7" neidb | head
```

```
# ID       Parent  Name
  1446642  83334   Escherichia coli O157:H7 str. 2011EL-2112
  991906   83334   Escherichia coli O157:H7 str. 611
  1446651  83334   Escherichia coli O157:H7 str. 2011EL-2313
...
```

Our `taxi` query returns 137 distinct taxa.

96c ⟨*query.sh* 96b⟩+≡ ◁96b 96d▷
```
  taxi -s "O157:H7" neidb |
      tail -n +2 |
      wc -l
```

In order to find out the taxon that summarizes all O157:H7 strains, we sort and count the parent taxa to find that 136 of our 137 taxa have parent 83334, which in turn has parent 562.

96d ⟨*query.sh* 96b⟩+≡ ◁96c 96e▷
```
  taxi -s O157:H7 neidb |
      tail -n +2 |
      awk '{print $2}' |
      sort |
      uniq -c
```

```
    1 562
  136 83334
```

We can use `dree` to visualize the taxonomic tree rooted on taxon 83334. The output of `dree` is in dot notation, which we render into the tree shown in Figure 4.1 using the program `dot` from the graphviz package. Figure 4.1 is too cramped for reading, but we can see that the tree has only two levels and that most of its nodes are colored, which means they are linked to genome sequences.

96e ⟨*query.sh* 96b⟩+≡ ◁96d 98a▷
```
  dree 83334 neidb
```

Figure 4.1: The taxonomic tree of *E. coli* O157:H7 drawn with `dree`; taxa with sequenced genomes are shown in color.

We saw previously that the parent of 83334 is taxon 562. To find out its identity, we use `ants` to print the ancestors of 83334. It turns out that taxon 562 is the species *E. coli*.

98a ⟨*query.sh* 96b⟩+≡ ◁96e 98b▷
```
  ants 83334 neidb

# Back  ID      Name                      Rank
  9     1       root                      no rank
  8     131567  cellular organisms        no rank
  7     2       Bacteria                  superkingdom
  6     1224    Proteobacteria            phylum
  5     1236    Gammaproteobacteria       class
  4     91347   Enterobacterales          order
  3     543     Enterobacteriaceae        family
  2     561     Escherichia               genus
  1     562     Escherichia coli          species
  0     83334   Escherichia coli O157:H7  serotype
```

In the tree returned by `dree`, the number of genomes per node is reduced to presence/absence. To get the distribution of genomes across a subtree, we can list its nodes.

98b ⟨*query.sh* 96b⟩+≡ ◁98a 98c▷
```
  dree -l 83334 neidb | head

# Taxid  Rank      Genomes
83334    serotype  190
1446642  strain    1
991906   strain    0
...
```

We add taxon names to the list.

98c ⟨*query.sh* 96b⟩+≡ ◁98b 98d▷
```
  dree -l -n 83334 neidb | head

# Taxid  Rank      Genomes  Name
83334    serotype  190      Escherichia coli O157:H7
1446642  strain    1        Escherichia coli O157:H7 str. ...
991906   strain    0        Escherichia coli O157:H7 str. 611
...
```

There are 329 genomes for serotype O157:H7.

98d ⟨*query.sh* 96b⟩+≡ ◁98c 98e▷
```
  dree -n -l 83334 neidb |
      tail -n +2 |
      awk '{s += $3}END{print s}'
```

We can reverse-sort the taxon list by the number of genomes to see that it ranges from 190 to 0.

98e ⟨*query.sh* 96b⟩+≡ ◁98d 99a▷
```
  dree -n -l 83334 neidb |
      tail -n +2 |
      sort -k 3 -n -r
```

```
83334    serotype  190      Escherichia coli O157:H7
155864   strain    7        Escherichia coli O157:H7 str. EDL933
1343836  strain    2        Escherichia coli O157:H7 str. F8092B
1286877  strain    2        Escherichia coli O157:H7 str. TW14313
997825   strain    1        Escherichia coli O157:H7 str. 1125
...
1240385  strain    1        Escherichia coli O157:H7 str. ...
991907   strain    0        Escherichia coli O157:H7 str. 262
991906   strain    0        Escherichia coli O157:H7 str. 611
886670   strain    0        Escherichia coli O157:H7 str. ZAP430
410290   strain    0        Escherichia coli O157:H7 str. ...
1046240  strain    0        Escherichia coli O157:H7 str. 121
```

Now we search for the neighbors of O157:H7 using `neighbors`. We restrict the output from `neighbors` to taxa with genome sequences.

99a ⟨*query.sh* 96b⟩+≡ ◁98e 99b▷

```
  printf 83334 | neighbors -g neidb
```

```
# MRCA(targets): 83334, Escherichia coli O157:H7
# MRCA(targets+neighbors): 562, Escherichia coli
# Type  Taxon-ID  Name                         Genomes
t       83334     Escherichia coli O157:H7     GCA_001695515.1|...
tt      155864    Escherichia coli O157:H7...  GCA_000732965.1|...
tt      386585    Escherichia coli O157:H7...  GCA_000008865.2
tt      444447    Escherichia coli O157:H7...  GCA_000181735.1
...
n       3050630   Escherichia coli O78:H51     GCA_030347055.1|...
```

I've slightly edited the output of `neighbors` for clarity. It begins with three hashed lines. The first states the most recent common ancestor of the targets, *E. coli* O157:H7, with taxon-ID 83334. The second line states the most recent common ancestor of the neighbors *and* the targets, *E. coli*, taxon 562. The third line is the header of the subsequent table, which consists of four columns, type, taxon-ID, name, and genomes. There are three possible types, "t" for known target, "tt" for new target, and "n" for neighbor.

We can check that the `neighbors` output also contains the 329 target genomes we previously listed with `dree`.

99b ⟨*query.sh* 96b⟩+≡ ◁99a 99c▷

```
  printf 83334 |
      neighbors -l neidb |
      tail -n +2 |
      grep '^t' |
      wc -l
```

To download the target and neighbor genomes, we need the genome accessions. These are a bit difficult to extract from the "report view" of the `neighbors` results. So we use the "list view" instead. It consists of two columns, sample (target or neighbor) and accession.

99c ⟨*query.sh* 96b⟩+≡ ◁99b 100a▷

```
  printf 83334 | neighbors -l neidb | head
```

```
# Sample  Accession
t         GCA_001695515.1
t         GCA_017165115.1
t         GCA_017165395.1
...
```

We save the accessions to the file `acc.txt`.

100a ⟨*query.sh* 96b⟩+≡ ◁99c 100b▷
```
printf 83334 | neighbors -l neidb > acc.txt
```

Apart from the 329 target genomes we've already seen, there are a staggering 3127 neighbor genomes.

100b ⟨*query.sh* 96b⟩+≡ ◁100a 100c▷
```
grep -c '^n' acc.txt
```

To download the sequences, we split the accessions into two files, `tacc.txt` for the targets and `nacc.txt` for the neighbors.

100c ⟨*query.sh* 96b⟩+≡ ◁100b 100d▷
```
grep '^t' acc.txt | awk '{print $2}' > tacc.txt
grep '^n' acc.txt | awk '{print $2}' > nacc.txt
```

We use the program `datasets` provided by the NCBI to download genomes. It is available from the `datasets` web site,

`https://www.ncbi.nlm.nih.gov/datasets/`

We restrict our analysis to genomes with assembly-level "complete" and exclude genomes flagged as "atypical". We download the genomes in "dehydrated" form. We begin with the target genomes, of which there are 143. This means out of the 329 target genomes across all assembly levels, only 143 are typical and complete. We save the target genomes in `tdata.zip`.

100d ⟨*query.sh* 96b⟩+≡ ◁100c 100e▷
```
datasets download genome accession \
           --inputfile tacc.txt \
           --assembly-level complete \
           --exclude-atypical \
           --dehydrated \
           --filename tdata.zip
```

We repeat the download for the neighbor genomes, where only 306 out of the 3127 pass muster.

100e ⟨*query.sh* 96b⟩+≡ ◁100d 100f▷
```
datasets download genome accession \
           --inputfile nacc.txt \
           --assembly-level complete \
           --exclude-atypical \
           --dehydrated \
           --filename ndata.zip
```

The genomes arrive as zipped archives. We unzip the targets into the directory `tdata`.

100f ⟨*query.sh* 96b⟩+≡ ◁100e 101a▷
```
unzip tdata.zip -d tdata
```

We unzip the neighbors into the directory `ndata`.

101a ⟨*query.sh* 96b⟩+≡ ◁100f 101b▷
```
unzip ndata.zip -d ndata
```

We rehydrate the 143 target genomes.

101b ⟨*query.sh* 96b⟩+≡ ◁101a 101c▷
```
datasets rehydrate --directory tdata
```

We rehydrate the 306 neighbor genomes.

101c ⟨*query.sh* 96b⟩+≡ ◁101b 101d▷
```
datasets rehydrate --directory ndata
```

For easier handling, we move all genomes into a new directory, `all`. We begin with the targets. To help dividing the data into true, i. e. phylogenetic, targets and neighbors later, we give target genomes the prefix `t`.

101d ⟨*query.sh* 96b⟩+≡ ◁101c 101e▷
```
mkdir all
for a in tdata/ncbi_dataset/data/*/*.fna
do
    b=$(basename $a)
    mv $a all/t$b
done
```

We also move the neighbor genomes into `all` and mark their names with prefix `n`.

101e ⟨*query.sh* 96b⟩+≡ ◁101d 101f▷
```
for a in ndata/ncbi_dataset/data/*/*.fna
do
    b=$(basename $a)
    mv $a all/n$b
done
```

We've now got the taxonomic targets and neighbors united in the directory `all`, but distinguishable by their prefix. Our next task is to extract from this data set the phylogenetic—as opposed to taxonomic—targets and neighbors. We do this by calculating a distance-based phylogeny from the target and neighbor genomes. We calculate the requisite distances using the program `phylonium`[3] and save them in `o157.dist`. This takes about one and a half minutes.

101f ⟨*query.sh* 96b⟩+≡ ◁101e 101g▷
```
phylonium all/* > o157.dist
```

We calculate the neighbor-joining tree from the distances with `nj` and midpoint root it with `midRoot`. Both programs are part of the the biobox[4]. Then we label the internal nodes of the phylogeny with `land`, which is part of Neighbors, and save the final tree in `o157.nwk`.

101g ⟨*query.sh* 96b⟩+≡ ◁101f 102a▷
```
nj o157.dist | midRoot | land > o157.nwk
```

[3]`https://github.com/evolbioinf/phylonium`
[4]`https://github.com/evolbioinf/biobox`

The tree in `o157.nwk` is in the popular Newick format, and you can render it with your favorite tree plotting program. One example of such a program is `plotTree` from the biobox. Its default plot dimensions are too small for our tree of 449 taxa, so we set the dimensions to 1200×1200 pixels.

102a ⟨*query.sh* 96b⟩+≡ ◁101g 102b▷
```
plotTree -d 1200,1200 o157.nwk
```

However, even at this magnification the taxon labels are difficult to read. So we replace the neighbors by a simple `n`, which reveals that the targets are concentrated in the top part of the tree.

102b ⟨*query.sh* 96b⟩+≡ ◁102a 102c▷
```
sed 's/n[^f]*fna/n/g' o157.nwk |
    plotTree -d 1200,1200
```

To further explore the target region of our tree, let's pick the subtree rooted on node 268 as it appears to contain most targets and a bit of neighbor context. We do this with `pickle` in tree mode, `-t`. We now see that node 291 contains a lone target, while its other child, node 292, contains most of the targets and a closely related neighbor clade.

102c ⟨*query.sh* 96b⟩+≡ ◁102b 102d▷
```
sed 's/n[^f]*fna/n/g' o157.nwk |
    pickle -t 268 |
    plotTree -d 1200,1200
```

To make the target clade easier to read, we further zoom into it by picking node 292. This shows us that the targets are in clade 300, while its sister clade 293 contains eight neighbors.

102d ⟨*query.sh* 96b⟩+≡ ◁102c 102e▷
```
sed 's/n[^f]*fna/n/g' o157.nwk |
    pickle -t 292 |
    plotTree -d 1200,1200
```

At this magnification, you might notice that there are also taxonomic neighbors interspersed in our target clade. We count nine such misclassified taxa, which we treat as targets in subsequent analyses.

102e ⟨*query.sh* 96b⟩+≡ ◁102d 102f▷
```
pickle 300 o157.nwk |
    grep -c '^n'
```

Let's remove a chunk of the target clade to make it easier to read both the leaf labels and node labels. Nodes can be removed by using `pickle` in both tree and complement mode. Let's remove node 312.

102f ⟨*query.sh* 96b⟩+≡ ◁102e 103a▷
```
sed 's/n[^f]*fna/n/g' o157.nwk |
    pickle -t 292 |
    pickle -t -c 312 |
    plotTree -d 1200,1200
```

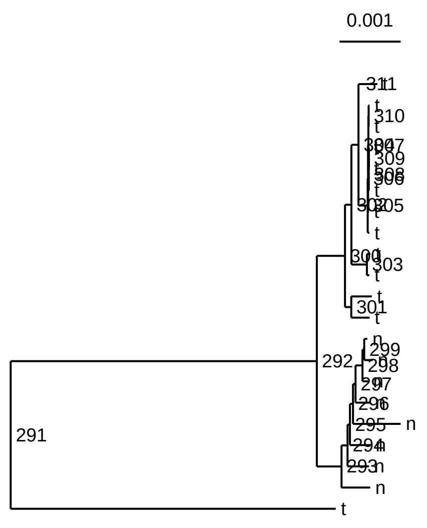

Figure 4.2: Part of the tree containing target and neighbor genomes of *E. coli* O157:H7; leaf labels are reduced to targets (*t*) and neighbors (*n*).

Now we also reduce the target labels to their first character and plot the reduced tree rooted on 291 shown in Figure 4.2.

103a ⟨*query.sh* 96b⟩+≡ ◁ 102f 103b ▷

```
  sed -E 's/([nt])[^f]*fna/\1/g' o157.nwk |
      pickle -t 291 |
      pickle -t -c 312 |
      plotTree -d 1200,1200
```

In Figure 4.2 clade 300 is our target clade, which took us a while to discover. Our discovery process is encapsulated in the program `fintac`, which finds the target clade by looking for the clade that maximizes the sum of the targets inside its subtree and the neighbors outside.

103b ⟨*query.sh* 96b⟩+≡ ◁ 103a 103c ▷

```
  fintac o157.nwk
```

```
#Clade  Parent  Split (%)
300     292     97.6
```

We can also use the `-a` option of `fintac` to list all splits.

103c ⟨*query.sh* 96b⟩+≡ ◁ 103b 104a ▷

```
  fintac -a o157.nwk | head
```

```
#Clade  Parent  Split (%)
300     292     97.6
302     300     97.1
304     302     96.7
291     290     96.0
```

```
290     271     95.8
292     291     95.8
311     304     95.1
312     311     94.9
271     270     91.5
```

The closely related sister clade of 300, 293, is made up of neighbors. As we already noted, the root of this partial tree, 291, is connected to a singleton branch leading to a target. This is one of two taxonomic targets outside of clade 300.

104a ⟨*query.sh* 96b⟩+≡ ◁ 103c 104b ▷

```
pickle -c 300 o157.nwk | grep '^t'
```

```
tGCA_022558925.1_ASM2255892v1_genomic.fna
tGCA_003722195.1_ASM372219v1_genomic.fna
```

Since the distance between this lone putative target and the many targets in clade 300 is substantial, we reassign the loner as a neighbor.

We begin our search for markers by splitting our tree into targets, i. e. clade 300, and neighbors, the precise difinition of which comes later.

To separate the targets, we make a directory, `targets`. Then we use `pickle` in default mode to list the taxa in the target clade. For each taxon we create a symbolic link to the original data.

104b ⟨*query.sh* 96b⟩+≡ ◁ 104a 104c ▷

```
mkdir targets
pickle 300 o157.nwk |
    grep -v '^#' |
    while read a; do
          ln -s $(pwd)/all/$a $(pwd)/targets/$a
    done
```

The program `fur` works on the idea that the *closest* neighbors are the most informative for marker discovery. So we construct our neighbor set from the genomes in clade 293.

104c ⟨*query.sh* 96b⟩+≡ ◁ 104b 104d ▷

```
mkdir neighbors
pickle 293 o157.nwk |
    grep -v '^#' |
    while read a; do
          ln -s $(pwd)/all/$a $(pwd)/neighbors/$a
    done
```

In preparation for running `fur`, we make its database with `makeFurDb`. Since the database is used to compare the targets in node 300 with the neighbors in node 293, we call it `300_293.db`. Its construction takes approximately 25 s.

104d ⟨*query.sh* 96b⟩+≡ ◁ 104c 104e ▷

```
makeFurDb -t targets/ -n neighbors/ -d 300_293.db
```

Given the database, we can apply `fur` to it. This takes nine seconds and returns 8.6 kb marker candidates, of which 2.2 kb are `N`s. We save these sequences in the file `300_293.fasta`.

104e ⟨*query.sh* 96b⟩+≡ ◁ 104d 105a ▷

```
fur -d 300_293.db > 300_293.fasta
```

```
Step           Sequences  Length    Ns
-------------  ---------  ------    --
Subtraction_1        247  226718     0
Intersection          44   16383   504
Subtraction_2         12    8668  2201
```

To check whether these markers crosshybridize with markers in the wider neighborhood, we repeat the analysis with a neighborhood consisting of everything but node 300 making up the neighborhood.

105a ⟨*query.sh* 96b⟩+≡ ◁104e 105b▷

```
rm neighbors/*
pickle -c 300 o157.nwk |
    grep -v '^#' |
    while read a; do
          ln -s $(pwd)/all/$a $(pwd)/neighbors/$a
    done
```

The new database compares clade 300 to everything else in the tree, so we call it `300.db`. Its construction takes 13.5 minutes.

105b ⟨*query.sh* 96b⟩+≡ ◁105a 105c▷

```
makeFurDb -t targets -n neighbors -d 300.db
```

Now the intersection step of `fur` comes up empty.

105c ⟨*query.sh* 96b⟩+≡ ◁105b 106a▷

```
fur -d 300.db
```

```
Step           Sequences  Length  Ns
-------------  ---------  ------  --
Subtraction_1         30   24281   0
Intersection           0       0   0
```

An empty intersection after a non-empty subtraction implies the target representative contains material that is absent from all neighbors, but not present in all other targets. This seems odd. Now, the requirement that markers be present in all targets means that a single atypical target genome can wipe out markers found everywhere else. So our next task is to look for atypical genomes among the targets.

When we downloaded the data, we already excluded genomes deemed "atypical" by the program `datasets` based on about a dozen criteria. We concentrate on genome length. According to `datasets`, a genome is atypical if it is 50% larger or smaller than the average genome length[5]. If we think of atypical genomes as outliers, this definition is somewhat unorthodox. A more common definition of outliers is based on the distance between the first and third quartile of a distribution. This definition is implemented in the program `outliers`, which is also part of Neighbors. For each target genome, we extract its name and its length and save the data in `tlen.dat`.

106a ⟨*query.sh* 96b⟩+≡ ◁ 105c 106b ▷
```
for a in targets/*.fna; do
    echo -n $a ' ';cres $a |
          grep To |
          awk '{print $2}'
done > tlen.dat
```

When we search the genome lengths for outliers, we find three mildly large and one extremely short genome.

106b ⟨*query.sh* 96b⟩+≡ ◁ 106a 106c ▷
```
awk '{print $2}' tlen.dat | outliers
```

```
#Lower_outer_fence  Lower_inner_fence  Lower_quartile...
5.24282e+06         5.399746625e+06    5.55667325e+06...
Mild_outliers: 5.825793e+06 5.831209e+06 5.8662e+06
Extreme_outlier: 4.869019e+06
```

Let's find the name of the extremely short genome, remove it from the targets, and rerun the analysis. This time the intersection is not empty and contains 2.8 kb, but we lose it all in the second subtraction step.

106c ⟨*query.sh* 96b⟩+≡ ◁ 106b 107a ▷
```
awk '$2==4.869019e+06' tlen.dat
rm targets/tGCA_030908645.1_ASM3090864v1_genomic.fna
makeFurDb -t targets -n neighbors -o -d 300.db
fur -d 300.db
```

```
Step           Sequences  Length  Ns
-------------  ---------  ------  --
Subtraction_1         39   28967   0
```

[5] https://www.ncbi.nlm.nih.gov/datasets/docs/v2/troubleshooting/faq/#what-are-atypical-genomes

```
Intersection           7    2750  13
Subtraction_2          0       0   0
```

For the second subtraction step `fur` calls Blast. By default this runs in the sensitive blastn mode. However, we can switch to the less sensitive megablast mode, which leaves two fragment with a total of 210 bp. Not much, but much more than nothing.

107a ⟨*query.sh* 96b⟩+≡ ◁ 106c 107b ▷

```
  fur -d -m 300.db
```

```
fur -m -d 300.db/ > 300.fasta
Step           Sequences  Length  Ns
-------------  ---------  ------  --
Subtraction_1         39   28967   0
Intersection           7    2750  13
Subtraction_2          2     210   0
```

In an attempt to improve our marker yield, we look for nested markers. Notice that in Figure 4.2 the branch separating node 291 from node 292 is much longer than the branch separating clades 300 and 293. So we begin our search for nested markers by again looking for regions that distinguish targets 300 from neighbors 293, only this time we don't include the extremely short genome. In a second step we'll compare targets 292 to everything else. But first we compare 300 vs. 293, which gives 13.3 kb markers with almost 5 kb `Ns`, up from 8.8 kb in our previous analysis.

107b ⟨*query.sh* 96b⟩+≡ ◁ 107a 107c ▷

```
  rm neighbors/*
  pickle 293 o157.nwk |
          grep -v '^#' |
          while read a; do
              ln -s $(pwd)/all/$a $(pwd)/neighbors/$a
          done
  makeFurDb -t targets -n neighbors -d 300_293.db -o
  fur -d 300_293.db > 300_293.fasta
```

```
Step           Sequences  Length    Ns
-------------  ---------  ------    --
Subtraction_1        331  255255     0
Intersection         108   33484   691
Subtraction_2         15   13279  4863
```

Now we compare 292 to the rest. We begin by preparing the targets and remove the extra short genome.

107c ⟨*query.sh* 96b⟩+≡ ◁ 107b 108a ▷

```
  rm targets/*
  pickle 292 o157.nwk |
      grep -v '^#' |
      while read a; do
            ln -s $(pwd)/all/$a $(pwd)/targets/$a
      done
  rm targets/tGCA_030908645.1_ASM3090864v1_genomic.fna
```

And now the neighbors.

108a ⟨*query.sh* 96b⟩+≡ ◁ 107c 108b ▷
```
rm neighbors/*
pickle -c 292 o157.nwk |
    grep -v '^#' |
    while read a; do
          ln -s $(pwd)/all/$a $(pwd)/neighbors/$a
    done
```

We run `makeFurDb` and `fur` to find a very poor yield in default mode, but with megablast mode there are 3.2 kb markers, of which half a kb is `N`s. That's a start.

108b ⟨*query.sh* 96b⟩+≡ ◁ 108a
```
makeFurDb -t targets -n neighbors -d 292.db
fur -m -d 292.db > 292.fasta
```

```
Step           Sequences  Length   Ns
-------------  ---------  ------   --
Subtraction_1        218   78859    0
Intersection         107   18743  852
Subtraction_2         12    3188  555
```

The next step in the design of diagnostic markers would be to construct PCR primers based on `292.fasta` and `300_293.fasta`. We would then test *in silico* the specificity and sensitivity of the pair of primers in detecting *E. coli* O157:H7. Any primers from the *in silico* work would ultimately need to be tested in the lab.

Bibliography

[1] A. A. A. Donovan and B. W. Kernighan. *The Go Programming Language*. Addison-Wesley, New York, 2016.

[2] B. Haubold, F. Klötzl, L. Hellberg, D. Thompson, and M. Cavalar. `Fur`: Find Unique genomic Regions for diagnostic PCR. *Bioinformatics*, 37:2081–2087, 2021.

[3] F. Klötzl and B. Haubold. `Phylonium`: fast estimation of evolutionary distances from large samples of similar genomes. *Bioinformatics*, 36:2040–46, 2020.

[4] E. Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Enno Ohlebusch, Ulm, 2013.