mapro: Marker Discovery in Prokaryotes

Beatriz Vieira Mourato & Bernhard Haubold

29 Jan 2024,

Contents

Chapter 1

Introduction

The program `fur` finds unique regions in genome sequences [1]. It takes as input a set of target genomes and a set of closely related, but distinct, neighbor genomes. Then `fur` finds the regions common to all targets that are absent form the neighbors. Such regions can be used to construct genetic markers.

The version of `fur` presented in the original publication yielded PCR markers of excellent specificity and sensitivity [1]. However, the program was based on indexing all neighbors in one fell swoop, which didn't scale for large genomes, or for the large collections of bacterial genomes now routinely collected during an outbreak.

In version 4 of `fur` we replaced this monolithic indexing by iterative indexing, which reduced the memory requirement from proportional to the size of the complete neighborhood to the size of the longest neighbor sequence. In addition, iterative indexing opened the way to parallelization, which sped up `fur` dramatically.

In this document we explore the application of the new `fur` to large collections of bacterial genomes. The choice of target genomes is delegated to the NCBI, which keeps a "small curated subset of really good and scientifically important prokaryotic genomes" on its website at

`ftp.ncbi.nlm.nih.gov/genomes/GENOME_REPORTS/prok_reference_genomes.txt`

We downloaded this file on November 16, 2023, although its time stamp is February 11, 2019. Table 1.1 shows an edited version of the 109 bacteria singled out by the NCBI, ordered by the number of sequenced genomes. The number of sequences ranges from 15,316 for *Eschrerichia coli* to one for six taxa, including *Streptomyces coelicolor*, which, according to the NCBI, is "responsible for producing more than half the known natural antibiotics".

Figure 1.1 shows the number of sequenced genomes as a function of the taxon's table rank. Notice the two phases in the graph, the first containing organisms with 2000 genomes or more, and the rest.

We have analyzed these strains to demonstrate the feasibility of marker discovery in the large using Fur. For each strain we download the target and neighbor genomes, extract the markers from them, and pick the best pair of PCR primers from the markers. Finally we test the primers to calculate their sensitivity and specificity. This gives a rough indication of what to expect when applying the primers *in vitro*.

Table 1.1: Entries in `prok_reference_genomes.txt` ordered by number of sequenced genomes.

#	Species	Genomes	#	Species	Genomes
1	*Escherichia coli*	15316	56	*Lactobacillus salivarius*	84
2	*Salmonella enterica*	13751	57	*Burkholderia mallei*	83
3	*Staphylococcus aureus*	10287	58	*Flavobacterium psychrophilum*	73
4	*Streptococcus pneumoniae*	8448	59	*Aeromonas hydrophila*	73
5	*Klebsiella pneumoniae*	6815	60	*Shigella dysenteriae*	72
6	*Mycobacterium tuberculosis*	6456	61	*Bifidobacterium bifidum*	69
7	*Pseudomonas aeruginosa*	4107	62	*Pseudomonas syringae group genomosp. 3*	68
8	*Listeria monocytogenes*	3784	63	*Fusobacterium nucleatum*	68
9	*Neisseria meningitidis*	1938	64	*Coxiella burnetii*	68
10	*Campylobacter jejuni*	1753	65	*Streptococcus sanguinis*	62
11	*Mycobacteroides abscessus*	1614	66	*Prochlorococcus marinus*	48
12	*Burkholderia pseudomallei*	1533	67	*Sinorhizobium medicae*	42
13	*Clostridioides difficile*	1445	68	*Lactobacillus acidophilus*	39
14	*Enterococcus faecium*	1344	69	*Corynebacterium glutamicum*	39
15	*Helicobacter pylori*	1338	70	*Bacteroides thetaiotaomicron*	35
16	*Streptococcus suis*	1280	71	*Aliivibrio fischeri*	34
17	*Streptococcus agalactiae*	1110	72	*Buchnera aphidicola*	30
18	*Bacillus cereus*	1016	73	*Mycoplasma mycoides*	23
19	*Vibrio cholerae*	1002	74	*Rhodobacter sphaeroides*	21
20	*Enterococcus faecalis*	880	75	*Bordetella parapertussis*	21
21	*Vibrio parahaemolyticus*	872	76	*Sinorhizobium fredii*	18
22	*Bordetella pertussis*	765	77	*Treponema denticola*	17
23	*Haemophilus influenzae*	703	78	*Rickettsia prowazekii*	13
24	*Legionella pneumophila*	638	79	*Chlamydia pneumoniae*	13
25	*Neisseria gonorrhoeae*	573	80	*Moorella thermoacetica*	12
26	*Staphylococcus epidermidis*	553	81	*Mesoplasma florum*	12
27	*Shigella flexneri*	540	82	*Salinibacter ruber*	11
28	*Enterobacter cloacae*	512	83	*Caulobacter vibrioides*	11
29	*Bacillus thuringiensis*	499	84	*Thermus thermophilus*	10
30	*Streptococcus pyogenes*	489	85	*Clostridium acetobutylicum*	10
31	*Pseudomonas syringae*	386	86	*Mycolicibacterium smegmatis*	9
32	*Yersinia pestis*	378	87	*Bradyrhizobium diazoefficiens*	9
33	*Lactobacillus plantarum*	345	88	*Amycolatopsis mediterranei*	7
34	*Leptospira interrogans*	297	89	*Thermotoga maritima*	6
35	*Bacillus subtilis*	252	90	*Rhodopirellula baltica*	6
36	*Clostridium botulinum*	242	91	*Mycobacterium leprae*	6
37	*Francisella tularensis*	235	92	*Mesorhizobium ciceri*	6
38	*Bacillus anthracis*	233	93	*Geobacter sulfurreducens*	6
39	*Klebsiella aerogenes*	210	94	*Agrobacterium fabrum*	6
40	*Sinorhizobium meliloti*	204	95	*Ketogulonicigenium vulgare*	5
41	*Acinetobacter pittii*	202	96	*Deinococcus radiodurans*	5
42	*Streptococcus mutans*	194	97	*Rhodospirillum rubrum*	4
43	*Bifidobacterium longum*	189	98	*Desulfovibrio vulgaris*	4
44	*Yersinia enterocolitica*	181	99	*Chloroflexus aurantiacus*	3
45	*Lactococcus lactis*	180	100	*Brachybacterium faecium*	3
46	*Chlamydia trachomatis*	176	101	*Thermosynechococcus elongatus*	2
47	*Bacteroides fragilis*	160	102	*Shewanella oneidensis*	2
48	*Lactobacillus paracasei*	157	103	*Dictyoglomus turgidum*	2
49	*Pseudomonas putida*	118	104	*Thermodesulfovibrio yellowstonii*	1
50	*Borreliella burgdorferi*	111	105	*Thermanaerovibrio acidaminovorans*	1
51	*Gardnerella vaginalis*	104	106	*Streptomyces coelicolor*	1
52	*Xanthomonas campestris*	94	107	*Gloeobacter violaceus*	1
53	*Streptococcus mitis*	91	108	*Chlorobaculum tepidum*	1
54	*Bordetella bronchiseptica*	91	109	*Aquifex aeolicus*	1
55	*Mycoplasma pneumoniae*	89			

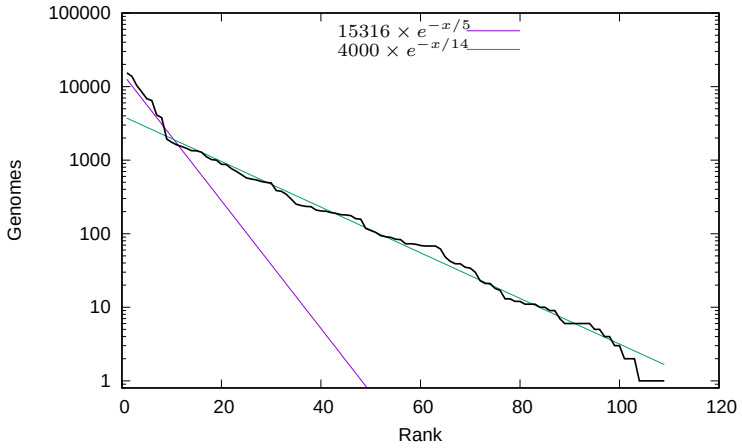

Figure 1.1: The number of sequenced genomes as a function of rank in Table 1.1 (*black*); the colored lines highlight the two phases of the graph.

Chapter 2

Drive the Analysis: `driver.sh`

Introduction

The script `driver.sh` drives the analysis of the reference prokaryotes. It is run as

```
$ bash driver.sh -d <blastDb> -n <neidb> < list.txt
```

where `list.txt` contains data in three columns, the download directory, the accession of the type strain, and its full name. The first five lines look like this:

```
aae     AE000657.1      Aquifex aeolicus VF5
afa     AE007870.2      Agrobacterium fabrum str. C58
afi     CP000020.2      Aliivibrio fischeri ES114
ahy     CP000462.1      Aeromonas hydrophila subsp. hydrophila...
ame     CP002000.1      Amycolatopsis mediterranei U32
...
```

The contents of `list.txt` were extracted from the list of reference prokaryotes supplied by the NCBI.

The script is either run in default, "making" mode, where it makes markers and primers. Alternatively, it is run in "checking" mode, where it checks primers generated in making mode. We separate these two modes as making markers and primers tends to be fast. In contrast, checking markers involves *in silico* PCR using Blast runs agains a large database like `nt` and is often slow and memory intensive.

Implementation

The script `driver.sh` first interacts with the user. Then it iterates over the lines in the file `list.txt`. These consist of download directory, accession, and type strain. For each line, we state what we are currently analyzing, and analyse the target typified by the type strain. Lines that start with a hash (`#`) are ignored.

Prog. 1 (`driver.sh`)

6a ⟨*driver.sh* 6a⟩≡

```
#!/usr/bin/bash
⟨Interact with user, Pr. 1 6b⟩
while read dir acc type; do
    if [[ $dir == "#" ]]; then
          continue
    fi
    echo "Analyzing $dir $type"
    ⟨Analyze target, Pr. 1 8b⟩
done
```

When interacting with the user, we set the usage of `driver.sh`, parse its options, and check them.

6b ⟨*Interact with user, Pr. 1* 6b⟩≡ (6a)

```
⟨Set usage, Pr. 1 7a⟩
⟨Parse options, Pr. 1 7b⟩
⟨Check options, Pr. 1 7c⟩
```

Apart from help, `-h`, `driver.sh` takes two options, the Neighbors database for looking up genomes, and the checking mode. The Neighbors database is mandatory, the checking mode optional. However, in checking mode the Blast database for *in silico* PCR also becomes mandatory.

7a ⟨*Set usage, Pr. 1* 7a⟩≡ (6b)

```
usage="$(basename $0) [-h -c -b <blastDb>] -n <neighborsDb>
Design and check primers for a list of targets.
Example making primers:   bash driver.sh -n neidb < list.txt
Example checking primers: bash driver.sh -n neidb -c"
usage="$usage -b <blastDb> < list.txt"
```

We iterate over the options and exit if there was an error.

7b ⟨*Parse options, Pr. 1* 7b⟩≡ (6b)

```
while getopts "hcb:n:" opt; do
    case $opt in
          h) echo "$usage"
             exit;;
          c) check=1;;
          b) bdb=$OPTARG;;
          n) ndb=$OPTARG;;
          \?) exit 1;;
    esac
done
```

We check the options for Neighbors database. If we are in checking mode, we also check the option for for the Blast database.

7c ⟨*Check options, Pr. 1* 7c⟩≡ (6b)

```
⟨Check Neighbors database option, -n, Pr. 1 7d⟩
if [[ $check ]]; then
    ⟨Check Blast database option, -b, Pr. 1 8a⟩
fi
```

If the user set a Neighbors database, we check that it exists; otherwise, we politely ask for it.

7d ⟨*Check Neighbors database option, -n, Pr. 1* 7d⟩≡ (7c)

```
if [[ $ndb ]]; then
    if [[ ! -f "$ndb" ]]; then
          echo "Can't find $ndb."
          exit 1
    fi
else
    echo "Please provide a Neighbors database."
    exit 1
fi
```

Similarly, if we are in checking mode and the user set the Blast database, we check that it exists; otherwise, we ask for it.

8a ⟨*Check Blast database option, -b, Pr. 1* 8a⟩≡ (7c)
```
if [[ $bdb ]]; then
    if [[ ! -f "$bdb.ndb" ]]; then
          echo "Can't find $bdb."
          exit 1
    fi
else
    echo "Please provide a Blast database."
    exit 1
fi
```

We analyze a given target either in making mode or in checking mode.

8b ⟨*Analyze target, Pr. 1* 8b⟩≡ (6a)
```
if [[ ! $check ]]; then
    ⟨Run in making mode, Pr. 1 8c⟩
else
    ⟨Run in checking mode, Pr. 1 9e⟩
fi
```

In making mode we get the genomes for a given target and check that we got them. Then we extract the markers. At this point we are done with the genomes, so we delete them and check the markers. Then we design the primers.

8c ⟨*Run in making mode, Pr. 1* 8c⟩≡ (8b)
```
⟨Get genomes, Pr. 1 8d⟩
⟨Check genomes, Pr. 1 8e⟩
⟨Extract markers, Pr. 1 9a⟩
⟨Delete the genomes, Pr. 1 9b⟩
⟨Check markers, Pr. 1 9c⟩
⟨Design primers, Pr. 1 9d⟩
```

If the download directory already exists, we remove it. The target and neighbor genomes are then downloaded with the script `genomes.sh`. It takes as arguments the type strain, the download directory, and the Neighbors database.

8d ⟨*Get genomes, Pr. 1* 8d⟩≡ (8c)
```
if [[ -d $dir ]]; then
    rm -r $dir
fi
bash ../scripts/genomes.sh -t "$type" -d $dir -n $ndb
```

If at least one target and one neighbor genome was found, we continue with the analysis. Otherwise, we skip to the next taxon.

8e ⟨*Check genomes, Pr. 1* 8e⟩≡ (8c)
```
nt=$(ls $dir/targets/ | wc -l)
nn=$(ls $dir/neighbors | wc -l)
if [[ $nt -eq 0 || $nn -eq 0 ]]; then
    continue
fi
```

We extract the markers with the script `markers.sh`. It takes as sole argument the download directory.

9a ⟨*Extract markers, Pr. 1* 9a⟩≡ (8c)
```
bash ../scripts/markers.sh -d $dir
```

The genome sequences are not needed any more. However, we do need their header lines to find the threshold distance when checking the primers, so we reduce the genomes to their header lines. In addition, we delete the Fur directory the directories of downloaded sequences, and their zip files.

9b ⟨*Delete the genomes, Pr. 1* 9b⟩≡ (8c)
```
for a in $dir/all/*; do
    head -n 1 $a > $dir/tmp
    mv $dir/tmp $a
done
rm -r $dir/all.db
rm -r $dir/tdata* $dir/ndata*
```

If no markers were found, we skip to the next taxon.

9c ⟨*Check markers, Pr. 1* 9c⟩≡ (8c)
```
if [[ ! -s $dir/markers.fasta ]]; then
    continue
fi
```

We design the primers with `primers.sh`, which again only requires the download directory as argument.

9d ⟨*Design primers, Pr. 1* 9d⟩≡ (8c)
```
bash ../scripts/primers.sh -d $dir
```

We have now finished the making mode and turn to the checking mode. In checking mode we check we got any primers, then we test them.

9e ⟨*Run in checking mode, Pr. 1* 9e⟩≡ (8b)
⟨*Check primers, Pr. 1* 9f⟩
⟨*Test primers, Pr. 1* 9g⟩

If no primers were found, we skip ahead to the next taxon.

9f ⟨*Check primers, Pr. 1* 9f⟩≡ (9e)
```
if [[ ! -s $dir/primers.fasta ]]; then
    continue
fi
```

To test the primers, we run `primers.sh` in checking mode, `-c`. Apart from the download directory, this requires the Blast database, the type strain, the type accession, and the Neighbors database.

9g ⟨*Test primers, Pr. 1* 9g⟩≡ (9e)
```
bash ../scripts/primers.sh -c -d $dir -b $bdb -t "$type" \
     -a $acc -n "$ndb"
```

This completes our script `driver.sh`.

Chapter 3

Get Genomes: `genomes.sh`

Introduction

The script `genomes.sh` fetches target and neighbor genomes for marker discovery with Fur. The script starts from a type strain and takes the taxonomic parent of that strain as the target taxon. The result of a run of `genomes.sh` is one directory containing the directories `targets` and `neighbors` ready for analysis with Fur.

Implementation

The script `genome.sh` interacts with the user, then gets the genomes.

Prog. 2 (`genomes.sh`)

11a ⟨*genomes.sh* 11a⟩≡

```
#!/usr/bin/bash
⟨Interact with user, Pr. 2 11b⟩
⟨Get genomes, Pr. 2 13b⟩
```

By way of user interaction we set the usage, parse the options, and check them.

11b ⟨*Interact with user, Pr. 2* 11b⟩≡ (11a)

```
⟨Set usage, Pr. 2 11c⟩
⟨Parse options, Pr. 2 12a⟩
⟨Check options, Pr. 2 12b⟩
```

The script `genomes.sh` takes four options,

1. `-t` the type strain
2. `-d` the directory in which the target and neighbor genomes for the type stain are deposited
3. `-n` the Neighbors database
4. `-h` help

The first three options are mandatory.

11c ⟨*Set usage, Pr. 2* 11c⟩≡ (11b)

```
ts="Enterobacter cloacae subsp. cloacae ATCC 13047"
usage="$(basename $0) [-h] -t <typeStrain> -d <downloadDir>"
usage="$usage -n <neighborsDb>
Download target and neighbor genomes for a type strain.
Example: bash genomes.sh -t \"$ts\""
usage="$usage -d ecl -n neidb"
```

We parse the options. If the user asked for help, we print the usage and exit. If the user entered an illegal option or omitted an argument, the system prints a message and we exit with error code 1.

12a ⟨*Parse options, Pr. 2* 12a⟩≡ (11b)
```
while getopts "ht:d:n:" arg; do
    case $arg in
          h) echo "$usage"
             exit;;
          t) strain=$OPTARG;;
          d) dir=$OPTARG;;
          n) db=$OPTARG;;
          \?) exit 1;;
    esac
done
```

We check the three mandatory options, `-t`, `-d`, and `-n`.

12b ⟨*Check options, Pr. 2* 12b⟩≡ (11b)
```
⟨Check -t, Pr. 2 12c⟩
⟨Check -d, Pr. 2 12d⟩
⟨Check -n, Pr. 2 13a⟩
```

If the user didn't set a type strain with `-t`, we politely ask for one and exit.

12c ⟨*Check -t, Pr. 2* 12c⟩≡ (12b)
```
if [[ ! $strain ]]; then
    echo "Please provide a type strain."
    echo "$usage"
    exit 1
fi
```

Similarly, if the user didn't provide a download directory with `-d`, we ask for it and exit. If, however, (s)he did provide one and it already exists, we'd rather not overwrite that and also bail with message.

12d ⟨*Check -d, Pr. 2* 12d⟩≡ (12b)
```
if [[ ! $dir ]]; then
    echo "Please provide a download directory."
    echo "$usage"
    exit 1
else
    if [[ -d $dir ]]; then
          echo "Download directory already exists."
          exit 1
    fi
fi
```

If the user didn't provided a Neighbors database, we ask for one, otherwise we check its existence.

13a ⟨*Check -n, Pr. 2* 13a⟩≡ (12b)
```
if [[ ! $db ]]; then
    echo "Please provide a Neighbors database."
    echo "$usage"
    exit 1
else
    if [[ ! -f $db ]]; then
          echo "Database $db doesn't exist."
          exit 1
    fi
fi
```

To get the genomes, we make the download directory, change into it, and prefix the database path by `../`. Then we get the genome accessions and divide them into taxonomic targets and neighbors. We download the corresponding genomes and split them into the phylogenetic targets and neighbors.

13b ⟨*Get genomes, Pr. 2* 13b⟩≡ (11a)
```
mkdir $dir
cd $dir
db="../$db"
⟨Get genome accessions, Pr. 2 13c⟩
⟨Divide accessions into taxonomic targets and neighbors, Pr. 2 13d⟩
⟨Download genomes, Pr. 2 14a⟩
⟨Divide genomes into phylogenetic targets and neighbors, Pr. 2 14d⟩
```

To obtain the accessions of the taxonomic target and neighbor genomes, we start by looking up the taxon ID of the type strain's parent in the Neighbors database. This gives us the ID of the targets we pass to `neighbors`, which lists all target and neighbor accessions. We save these accessions in the file `acc.txt`.

13c ⟨*Get genome accessions, Pr. 2* 13c⟩≡ (13b)
```
tid=$(taxi "$strain" $db |
            tail -n +2 |
            awk '{print $2}')
echo $tid |
    neighbors -l $db > acc.txt
```

The file `all.txt` contains two columns, type and genome accession. The type is either `t` for *target*, or `n` for *neighbor*. So we can use `grep` to divide the accessions into those of the targets, which we save in `tacc.txt`, and those of the neighbors, which we save in `nacc.txt`.

13d ⟨*Divide accessions into taxonomic targets and neighbors, Pr. 2* 13d⟩≡ (13b)
```
for a in t n; do
    grep ^${a} acc.txt |
          awk '{print $2}' > ${a}acc.txt
done
```

We download the genomes in compressed form and then unpack them.

14a ⟨*Download genomes, Pr. 2* 14a⟩≡ (13b)
⟨*Download compressed genomes, Pr. 2* 14b⟩
⟨*Unpack genomes, Pr. 2* 14c⟩

We use the tool `datasets`[1] for downloading the genomes listed in `[tn]acc.txt`. However, we restrict ourselves to genomes with complete assemblies and exclude those flagged as "atypical". The download is a zipped file in "dehydrated" format, which we save to separate zip files for targets and neighbors, `tdata.zip` and `ndata.zip`, respectively.

14b ⟨*Download compressed genomes, Pr. 2* 14b⟩≡ (14a)
```
for a in t n; do
    datasets download genome accession \
              --inputfile ${a}acc.txt \
              --assembly-level complete \
              --exclude-atypical \
              --dehydrated \
              --filename ${a}data.zip
done
```

If genomes were downloaded, we unpack them genomes by first unzipping them into the directories `[tn]data` and then rehydrating the sequences in these directories. If either no targets or no neighbors were downloaded, there is no point in continuing. In that case we delete any target sequences that might have been downloaded, and exit.

14c ⟨*Unpack genomes, Pr. 2* 14c⟩≡ (14a)
```
for a in t n; do
    if [[ -s ${a}data.zip ]]; then
          unzip ${a}data.zip -d ${a}data
          datasets rehydrate --directory ${a}data
    else
          if [[ -s tdata.zip ]]; then
              rm -r tdata*
          fi
          exit
    fi
done
```

At this point we have a set of taxonomic targets and neighbors. We now divide these into phylogenetic targets and neighbors. For this we calculate their genome phylogeny, look up the target clade in that phylogeny, and split the genome sequences accordingly.

14d ⟨*Divide genomes into phylogenetic targets and neighbors, Pr. 2* 14d⟩≡ (13b)
⟨*Calculate genome phylogeny, Pr. 2* 14e⟩
⟨*Look up target clade in phylogeny, Pr. 2* 15e⟩
⟨*Split genomes by target clade, Pr. 2* 16a⟩

The genome phylogeny needs labeled leaves, so we rename the genome files before calculating the desired distance phylogeny from them.

14e ⟨*Calculate genome phylogeny, Pr. 2* 14e⟩≡ (14d)
⟨*Rename data files, Pr. 2* 15a⟩
⟨*Calculate distance phylogeny, Pr. 2* 15b⟩

[1] `www.ncbi.nlm.nih.gov/datasets/`

In the genome phylogeny, the labels of taxonomic targets are expected to have prefix `t`, those of the neighbors prefix `n`. So we create a directory, `all`, and move the taxonomic target and neighbor genomes into it with properly prefixed names.

15a ⟨*Rename data files, Pr. 2* 15a⟩≡ (14e)
```
mkdir all
for p in t n; do
    for a in ${p}data/ncbi_dataset/data/*/*.fna; do
          b=$(basename $a)
          mv $a all/${p}$b
    done
done
```

We count the number of genomes available. If there is one or none, we exit with message. If there are two, we calculate a UPGMA phylogeny. If there are at least three, we calculate a neighbor-joining phylogeny.

15b ⟨*Calculate distance phylogeny, Pr. 2* 15b⟩≡ (14e)
```
n=$(ls all/ | wc -l)
if [[ $n -le 1 ]]; then
    echo "Need at least two genomes to calculate phylogeny."
    exit 1
elif [[ $n -eq 2 ]]; then
    ⟨Calculate UPGMA phylogeny, Pr. 2 15c⟩
else
    ⟨Calculate neighbor-joining phylogeny, Pr. 2 15d⟩
fi
```

We calculate the pairwise distances between all genomes using `phylonium`. Then we convert these distances into a UPGMA tree, label its nodes, and save it to `all.nwk`.

15c ⟨*Calculate UPGMA phylogeny, Pr. 2* 15c⟩≡ (15b)
```
phylonium all/* |
    upgma |
    land > all.nwk
```

To calculate a neighbor-joining phylogeny, we also start from all pairwise distances and convert them to a neighbor-joining tree. This is midpoint-rooted, labeled, and also stored to `all.nwk`

15d ⟨*Calculate neighbor-joining phylogeny, Pr. 2* 15d⟩≡ (15b)
```
phylonium all/* |
    nj |
    midRoot |
    land > all.nwk
```

We look up the target clade in `all.nwk` with a call to the program `fintac`.

15e ⟨*Look up target clade in phylogeny, Pr. 2* 15e⟩≡ (14d)
```
tc=$(fintac all.nwk |
           tail -n +2 |
           awk '{print $1}')
```

We split the genomes into targets and neighbors by first getting the phylogenetic targets, then the phylogenetic neighbors.

16a ⟨*Split genomes by target clade, Pr. 2* 16a⟩≡ (14d)
⟨*Get phylogenetic targets, Pr. 2* 16b⟩
⟨*Get phylogenetic neighbors, Pr. 2* 16c⟩

We make the directory `targets` for the phylogenetic targets and list the taxa in the target clade using the program `pickle`. For each target taxon, we create a symbolic link from the file in `all` into the directory `targets`.

16b ⟨*Get phylogenetic targets, Pr. 2* 16b⟩≡ (16a)
```
mkdir targets
pickle $tc all.nwk |
    grep -v '^#' |
    while read a; do
          ln -s $(pwd)/all/$a $(pwd)/targets/$a
    done
```

The phylogenetic neighbors are the complement of the targets. We link them into the directory `neighbors`.

16c ⟨*Get phylogenetic neighbors, Pr. 2* 16c⟩≡ (16a)
```
mkdir neighbors
pickle -c $tc all.nwk |
    grep -v '^#' |
    while read a; do
          ln -s $(pwd)/all/$a $(pwd)/neighbors/$a
    done
```

We now have two sets of genomes in directories `targets` and `neighbors` ready for marker discovery with Fur; our work on `genomes.sh` is done.

Chapter 4

Get Markers: `markers.sh`

Introduction

The script `markers.sh` applies Fur to the contents of a download directory, `d`, generated with `genomes.sh`. The result is a set of markers in `d/markers.fasta`.

Implementation

The script `markers.sh` interacts with the user, then generates the markers.

Prog. 3 (`markers.sh`)

18a ⟨*markers.sh* 18a⟩≡
```
#!/usr/bin/bash
⟨Interact with user, Pr. 3 18b⟩
⟨Generate markers, Pr. 3 19b⟩
```

In the user interaction we set the usage, parse the options, and check them.

18b ⟨*Interact with user, Pr. 3* 18b⟩≡ (18a)
```
⟨Set usage, Pr. 3 18c⟩
⟨Parse options, Pr. 3 18d⟩
⟨Check options, Pr. 3 19a⟩
```

We declare two options, the download directory (`-d`) and help (`-h`). Setting the download directory is mandatory.

18c ⟨*Set usage, Pr. 3* 18c⟩≡ (18b)
```
usage="$(basename $0) [-h] -d <downloadDir>
Use Fur to extract markers from targets and neighbors.
Example: bash markers.sh ecl"
```

We parse the options. If the user asked for help, we print the usage and exit. If the user entered an unknown option or omitted the argument of `-d`, the system prints an error message and we exit.

18d ⟨*Parse options, Pr. 3* 18d⟩≡ (18b)
```
while getopts "hd:" arg; do
    case $arg in
          h) echo "$usage"
             exit;;
          d) dir=$OPTARG;;
          \?) exit 1;;
    esac
done
```

If the user set the download directory with `-d`, we make sure it exists; otherwise, we ask for it.

19a ⟨*Check options, Pr. 3* 19a⟩≡ (18b)

```
if [[ ! $dir ]]; then
    echo "Please provide a download directory."
    echo "$usage"
    exit 1
else
    if [[ ! -d $dir ]]; then
          echo "Download directory $dir doesn't exist."
    fi
fi
```

To generate the markers, we change into the download directory and run `makeFurDb` followed by `fur`. We filter the output of `fur` through `cleanSeq` and save the result to `markers.fasta`.

19b ⟨*Generate markers, Pr. 3* 19b⟩≡ (18a)

```
cd $dir
makeFurDb -t targets -n neighbors -d all.db
fur -d all.db |
    cleanSeq > markers.fasta
```

We've extracted the markers, which completes `markers.sh`.

Chapter 5

Generate and test primers: `primers.sh`

Introduction

The script `primers.sh` takes as input a download directory, `d`, generated with `markers.sh`. It then either picks primers from the markers contained in `d` and stores them in `d/primers.fasta`, or checks primers, which are assumed to be located in `d/primers.fasta`. The primers generated with `primers.sh` come annotated with their pair penalty. The results of checking primers are their sensitivity and specificity stored in `d/cops.out`.

Implementation

The script `primers.sh` interacts with the user, changes into the download directory, and then either picks primers or checks them.

Prog. 4 (`primers.sh`)

21a ⟨*primers.sh* 21a⟩≡

```
#!/usr/bin/bash
⟨Interact with user, Pr. 4 21b⟩
⟨Change into download directory, Pr. 4 24b⟩
if [[ ! $check ]]; then
    ⟨Pick primers, Pr. 4 24c⟩
else
    ⟨Check primers, Pr. 4 24d⟩
fi
```

To interact with the user we set the usage of `primers.sh`, parse its options, and check them.

21b ⟨*Interact with user, Pr. 4* 21b⟩≡ (21a)

```
⟨Set usage, Pr. 4 21c⟩
⟨Parse options, Pr. 4 22a⟩
⟨Check options, Pr. 4 22c⟩
```

Apart from help (`-h`), `primers.sh` takes as options the download directory (`-d`) and whether or not to check primers. In checking mode, we need a Blast database with taxonomy information for *in silico* PCR, the name of the type strain, its accession, and the name of the Neighbors database. The download directory is always mandatory. In checking mode, the Blast database, the type strain, the Neighbors database, and the reference list also become mandatory.

21c ⟨*Set usage, Pr. 4* 21c⟩≡ (21b)

```
ts="Enterobacter cloacae subsp. cloacae ATCC 13047"
a="CP001918"
usage="$(basename $0) [-h -c -b <blastDb> -t <typeStrain>"
usage="$usage -a <typeAcc> -n <neighborsDb>]"
usage="$usage -d <downloadDir>
Design and check primers.
Example without checking: bash primers.sh -d ecl
Example with checking: bash primers.sh -d ecl -c"
usage="$usage -b /ssd01/Nt/nt -t \"$ts\" -a $a -n neidb"
```

We iterate over the options and classify each one.

22a ⟨*Parse options, Pr. 4* 22a⟩≡ (21b)
```
while getopts "hcb:t:n:d:a:" opt; do
    case $opt in
        ⟨Classify option, Pr. 4 22b⟩
    esac
done
```

We classify the current option and exit if an error occurred.

22b ⟨*Classify option, Pr. 4* 22b⟩≡ (22a)
```
h) echo "$usage"
   exit;;
c) check=1;;
b) bdb=$OPTARG;;
t) strain=$OPTARG;;
n) ndb=$OPTARG;;
d) dir=$OPTARG;;
a) typeAcc=$OPTARG;;
\?) exit 1;;
```

We check the download directory, `-d`. If primer checking is requested, `-c`, we make sure there really are primers. Then we also check the Blast database, `-b`, the type strain, `-t`, and the Neighbors database, `-n`.

22c ⟨*Check options, Pr. 4* 22c⟩≡ (21b)
```
⟨Check -d, Pr. 4 23a⟩
if [[ $check ]]; then
    ⟨Ensure primers, Pr. 4 22d⟩
    ⟨Check -b, Pr. 4 23b⟩
    ⟨Check -t, Pr. 4 23c⟩
    ⟨Check -n, Pr. 4 24a⟩
fi
```

If there are no primers, we quit with a message.

22d ⟨*Ensure primers, Pr. 4* 22d⟩≡ (22c)
```
if [[ ! -s "$dir/primers.fasta" ]]; then
    echo "Can't find primers in $dir/primers/fasta"
    exit 1
fi
```

If the user set a download directory with `-d`, we check that it exists. Otherwise we ask for it.

23a ⟨*Check -d, Pr. 4* 23a⟩≡ (22c)
```
if [[ $dir ]]; then
    if [[ ! -d $dir ]]; then
          echo "Download directory $dir doesn't exist."
          exit 1
    fi
else
    echo "Please enter a dowload directory."
    echo "$usage"
    exit 1
fi
```

If the user set a Blast database with `-b`, we check it exists; otherwise we ask for it.

23b ⟨*Check -b, Pr. 4* 23b⟩≡ (22c)
```
if [[ $bdb ]]; then
    if [[ ! -f "$bdb.ndb" ]]; then
          echo "Can't find Blast db $bdb."
          exit 1
    fi
else
    echo "Please provide a Blast db."
    echo "$usage"
    exit 1
fi
```

If the user didn't set the type strain with `-t`, we ask for it.

23c ⟨*Check -t, Pr. 4* 23c⟩≡ (22c)
```
if [[ ! $strain ]]; then
    echo "Please set the type strain."
    echo "$options"
    exit 1
fi
```

If the user didn't set the accession of the type strain, we ask for it.

23d ⟨*Check -a, Pr. 4* 23d⟩≡
```
if [[ ! $typeAcc ]]; then
    m="Please the accession of the type strain"
    exit 1
fi
```

If the user set a Neighbors database with `-n`, we check it exists. Otherwise, we ask for one.

24a ⟨*Check -n, Pr. 4* 24a⟩≡ (22c)
```
if [[ $ndb ]]; then
    if [[ ! -f $ndb ]]; then
          echo "Can't find Neighbors database $ndb."
          exit 1
    fi
else
    echo "Please provide a Neighbors database."
    echo "$options"
    exit 1
fi
```

After we change into the download directory, we adjust the paths of the Neighbors database, the Blast database, and the reference list, unless they are given as absolute paths.

24b ⟨*Change into download directory, Pr. 4* 24b⟩≡ (21a)
```
cd $dir
if [[ "$ndb" != "/"* ]]; then
    ndb="../$ndb"
fi
if [[ "$bdb" != "/"* ]]; then
    bdb="../$bdb"
fi
if [[ "$ref" != "/"* ]]; then
    ref="../$ref"
fi
```

To pick primers, we convert the markers in `markers.fasta` to input for `primer3`. The output from `primer3` is sorted by penalty and we store the best primer pair in `primers.fasta`. In the header of the forward primer we also store the pair penalty.

24c ⟨*Pick primers, Pr. 4* 24c⟩≡ (21a)
```
fa2prim markers.fasta |
    primer3_core |
    prim2tab |
    tail -n +2 |
    sort -n |
    head -n 1 |
    while read p f r i; do
          printf ">f penalty: %s\n%s\n>r\n%s\n" \
                 $p $f $r > primers.fasta
          break
    done
```

Primer checking is done in two steps, first we score the primers, then we correct the primer scores.

24d ⟨*Check primers, Pr. 4* 24d⟩≡ (21a)
```
⟨Score primers, Pr. 4 25a⟩
⟨Check primer scores, Pr. 4 25e⟩
```

Primers are scored by running the program `scop`. It takes as input the list of target taxa. To look up the target taxa, we need the target's taxon ID.

25a ⟨*Score primers, Pr. 4* 25a⟩≡ (24d)
⟨*Look up taxon ID of target, Pr. 4* 25b⟩
⟨*Look up target taxa, Pr. 4* 25c⟩
⟨*Run* `scop`*, Pr. 4* 25d⟩

We look up the taxon ID of the target from the Neighbors database using the program `taxi`.

25b ⟨*Look up taxon ID of target, Pr. 4* 25b⟩≡ (25a)
```
tid=$(taxi "$strain" $ndb |
            tail -n +2 |
            awk '{print $2}')
```

We look up the target taxa from the Neighbors database using `neighbors` and store them in the file `taxa.txt`.

25c ⟨*Look up target taxa, Pr. 4* 25c⟩≡ (25a)
```
echo $tid |
    neighbors $ndb |
    grep '^t' |
    awk '{print $2}' > taxa.txt
```

We score the primers in `primers.fasta` by running `scop` on the Blast database and the taxa we are targeting in that database.

25d ⟨*Run* `scop`*, Pr. 4* 25d⟩≡ (25a)
```
scop -d $bdb -t taxa.txt primers.fasta > scop.out
```

The primer scores we just calculated with `scop` are corrected with `cops`. This takes as input the threshold distance for inclusion among the targets. So calculate that threshold distance before we run `cops`.

25e ⟨*Check primer scores, Pr. 4* 25e⟩≡ (24d)
⟨*Calculate threshold distance, Pr. 4* 25f⟩
⟨*Run* `cops`*, Pr. 4* 26c⟩

The threshold distance is calculated as twice the cumulative distance from the type strain's leaf in the tree to the parent of the target clade. So we first look up the leaf label of the type strain. Then we look up the target clade, and finally calculate the distance to the target clade's parent.

25f ⟨*Calculate threshold distance, Pr. 4* 25f⟩≡ (25e)
⟨*Look up leaf label of type strain, Pr. 4* 25g⟩
⟨*Look up target clade, Pr. 4* 26a⟩
⟨*Calculate distance from type strain to target clade's parent, Pr. 4* 26b⟩

We search among the header lines of the targets for the name of the file that contains the type strain. This is the desired leaf label of the type strain.

25g ⟨*Look up leaf label of type strain, Pr. 4* 25g⟩≡ (25f)
```
leafLabel=$(head -n 1 targets/* |
                  grep -B 1 $typeAcc |
                  head -n 1 |
                  tr -d ' =><' |
                  sed 's/targets\///')
```

We use `fintac` to look up the target clade.

26a ⟨*Look up target clade, Pr. 4* 26a⟩≡ (25f)
```
targetClade=$(fintac all.nwk |
                    tail -n +2 |
                    head -n 1 |
                    awk '{print $1}')
```

We calculate the distance from the type strain to the target clade's parent using the program `climt`.This prints four columns,

1. the steps up from the starting node
2. the label of the current node, v
3. the length of the branch from the current node to its parent, l
4. the cumulative branch length on the path to the current node, c

So we climb until v is the target clade, then calculate the threshold, t as

$$t = 2(l + c).$$

26b ⟨*Calculate distance from type strain to target clade's parent, Pr. 4* 26b⟩≡ (25f)
```
t=$(climt $leafLabel all.nwk |
          awk -v v=$targetClade '$2==v{print 2*($3+$4)}')
threshold=$t
```

We run cops on the output of `scop` and save the result in `cops.out`.

26c ⟨*Run cops, Pr. 4* 26c⟩≡ (25e)
```
cops -D -d $bdb -r $typeAcc -t $threshold scop.out > cops.out
```

This concludes our implementation of `primers.sh`.

Bibliography

[1] B. Haubold, F. Klötzl, L. Hellberg, D. Thompson, and M. Cavalar. `Fur`: Find unique genomic regions for diagnostic PCR. *Bioinformatics*, 37:2081–2087, 2021.