Handling disease outbreak data using *epibase* 0.1-0

Thibaut Jombart, Xavier Didelot, Rolf Ypma, Lulla Opatowski, Anne Cori

May 26, 2013

Abstract

This vignette introduces the main functionalities of *epibase*, a package implementing core tools for the analysis of outbreak data. Disease outbreak data can be diverse and complex, and the purpose of *epibase* is to simplify the handling of this information. The main feature of the package lies in the formal (S4) class `obkData` (for "outbreak data"), which offers a coherent way of handling data on individuals, samples, contact networks, clinical events, as well as phylogenies and genomic sequences. Beyond introducing this data structure, this tutorial illustrates how these objects can be handled and visualized in R.

Contents

1 Storing outbreak data

In this section, we first detail the structure of the classes of objects used in *epibase*, and then explain how to import data into the package.

1.1 Class definitions

Data collected during outbreaks can be hugely diverse and complex. In *epibase*, our purpose is to have a general class of objects which can store virtually any information sampled during an outbreak, without the user worrying about storage issues and consistency amongst different types of data. For most purposes, the core class `obkData` can be taken as a black box, with which the user can interact using specific functions called *accessors*. However, a basic understanding of what type of information is stored in these objects will be useful.

1.1.1 `obkData`: storage of outbreak data

The main class of objects in *epibase* is `obkData`. This formal (S4) class is used to store various types of information gathered during outbreaks. The definition of the class in terms of R objects can be obtained by:

```
library(epibase)
```

```
## Loading required package:  MASS
## Loading required package:  ape
## Loading required package:  adegenet
## Loading required package:  ade4
##
## Attaching package:  'ade4'
## The following object is masked from 'package:base':
##
##    within
## Loading required package:  igraph
##
## Attaching package:  'igraph'
## The following object is masked from 'package:ape':
##
##    as.igraph, edges
##    ============================
##    adegenet 1.3-8 is loaded
##    ============================
##
##  - to start, type '?adegenet'
##  - to browse adegenet website, type 'adegenetWeb()'
##  - to post questions/comments:  adegenet-forum@lists.r-forge.r-project.org
## Loading required package:  network
## network:  Classes for Relational Data
## Version 1.7.2 created on March 15, 2013.
## copyright (c) 2005, Carter T. Butts, University of California-Irvine
##                     Mark S. Handcock, University of Washington
##                     David R. Hunter, Penn State University
##                     Martina Morris, University of Washington
##  For citation information, type citation("network").
##  Type help("network-package") to get started.
```

```
##
## Attaching package:  'network'
## The following objects are masked from 'package:igraph':
##
##    add.edges, add.vertices, %c%, delete.edges, delete.vertices,
##    get.edge.attribute, get.edges, get.vertex.attribute, is.bipartite,
##    is.directed, list.edge.attributes, list.vertex.attributes, %s%,
##    set.edge.attribute, set.vertex.attribute
## Loading required package:  networkDynamic
## Loading required package:  statnet.common
##
## networkDynamic:  version 0.4.1, created on 2013-05-2
## Copyright (c) 2013, Carter T. Butts, University of California -- Irvine
##                     Ayn Leslie-Cook, University of Washington
##                     Pavel N. Krivitsky, Penn State University
##                     Skye Bender-deMoll, University of Washington
##                     with contributions from
##                     Zack Almquist, University of California -- Irvine
##                     David R. Hunter, Penn State University
##                     Li Wang
##                     Kirk Li, University of Washington
##                     Steven M. Goodreau, University of Washington
##                     Martina Morris, University of Washington
## Based on "statnet" project software (statnet.org).
## For license and citation information see statnet.org/attribution
## or type citation("networkDynamic").
## Loading required package:  ggplot2
## Loading required package:  ggmap
## Loading required package:  sna
## sna:  Tools for Social Network Analysis
## Version 2.3-1 created on 2013-02-28.
## copyright (c) 2005, Carter T. Butts, University of California-Irvine
##  For citation information, type citation("sna").
##  Type help(package="sna") to get started.
##
## Attaching package:  'sna'
## The following object is masked from 'package:network':
##
##    %c%
## The following objects are masked from 'package:igraph':
##
##    betweenness, bonpow, %c%, closeness, degree, dyad.census, evcent,
##    hierarchy, is.connected, neighborhood, triad.census
## The following object is masked from 'package:ape':
##
##    consensus
## Loading required package:  scales
## Loading required package:  plyr
##
## Attaching package:  'plyr'
```

```
## The following object is masked from 'package:network':
##
##    is.discrete
## Loading required package:  reshape2
## Loading required package:  mapproj
## Loading required package:  maps
## Loading required package:  RColorBrewer
##  epibase 0.1-0 has been loaded
##
## Attaching package:  'epibase'
## The following object is masked from 'package:adegenet':
##
##    .S3MethodsClasses
```

```
getClassDef("obkData")
```

```
## Class "obkData" [package "epibase"]
##
## Slots:
##
## Name:         individuals             samples           clinical
## Class:    dataframeOrNULL    dataframeOrNULL         listOrNULL
##
## Name:                 dna           contacts              trees
## Class: obkSequencesOrNULL  obkContactsOrNULL   multiPhyloOrNULL
```

One can also examine a structure using an empty object:

```
new("obkData")
```

```
##
## === obkData object ===
## == Empty slots ==
##  @individuals, @samples, @clinical, @dna, @contacts, @trees
```

Each slot of an `obkData` object is optional. By convention, empty slots are always `NULL`. The slots respectively contain:

- `@individuals`: a `data.frame` storing individual data, such as age, sex, or onset of symptoms. If not `NULL`, this `data.frame` will have exactly one row per individual, with row names providing unique identifiers for individuals.
- `@samples`: a `data.frame` storing sample data, typically swab results or accession numbers of DNA sequences. If not `NULL`, this `data.frame` must contain the three following columns: `individualID` (unique identifiers for individuals), `sampleID` (identifiers for samples, possibly repeated if several measurements have been made on the same sample), and `date` (collection dates for the samples).
- `@clinical`: a list of `data.frame`s storing any additional clinical information; there is no constraint on the number of `data.frame`s stored, but each one must contain columns named `individualID` (unique identifiers for individuals) and `date` (date of observations/interventions).
- `@dna`: DNA sequences of one or more genes, stored as an `obkSequences` object. See section below for details on `obkSequences` objects.

- `@contacts`: dynamic contact network between the individuals, stored as an `obkContacts` object. See section below for details on `obkContacts` objects.
- `@trees`: a list of phylogenetic trees with the class `multiPhylo` (from the *ape* package); can be used to store e.g. a posterior distribution of trees from a Bayesian phylogenetic reconstruction using BEAST.

The slots of an object `foo` can be accessed using `foo@[name-of-the-slot]`. Let us use the toy outbreak dataset `ToyOutbreak` and examine its content:

```
data(ToyOutbreak)
class(ToyOutbreak)

## [1] "obkData"
## attr(,"package")
## [1] "epibase"

slotNames(ToyOutbreak)

## [1] "individuals" "samples"     "clinical"    "dna"         "contacts"
## [6] "trees"

head(ToyOutbreak)

##
## === obkData x ===
## == @individuals==
##   infector DateInfected Sex Age   lat     lon
## 1       NA   2000-01-01   M  33 51.52 -0.1805
## 2        1   2000-01-02   F  42 51.52 -0.1771
## 3        2   2000-01-03   F  44 51.52 -0.1614
## 4        2   2000-01-03   M  49 51.52 -0.1706
##
## == @samples==
##   individualID sampleID       date sequenceID locus
## 1            1        1 2000-01-01          1 gene1
## 2            2        2 2000-01-02          2 gene1
## 3            3        3 2000-01-03          3 gene1
## 4            4        4 2000-01-03          4 gene1
##
## == @clinical==
##   individualID       date temperature
## 1            1 2000-01-03        39.1
## 2            2 2000-01-03        40.4
## 3            3 2000-01-07        40.0
## 4            4 2000-01-08        39.8
##
## == @dna==
## [ 836 DNA sequences in 2 loci ]
##
## $gene1
## 418 DNA sequences in binary format stored in a matrix.
##
## All sequences of same length: 600
```

```
##
## Labels: 1 2 3 4 5 6 ...
##
## Base composition:
##     a     c     g     t
## 0.237 0.248 0.252 0.263
##
## $gene2
## 418 DNA sequences in binary format stored in a matrix.
##
## All sequences of same length: 1000
##
## Labels: 419 420 421 422 423 424 ...
##
## Base composition:
##     a     c     g     t
## 0.223 0.243 0.257 0.276
##
##
## == @contacts==
##  Number of individuals = 20
##  Number of contacts = 19
##  Contacts = dynamic
## NetworkDynamic properties:
##   distinct change times: 5
##   maximal time range: 0 to 4
##
##  Network attributes:
##   vertices = 20
##   directed = FALSE
##   hyper = FALSE
##   loops = FALSE
##   multiple = TRUE
##   bipartite = FALSE
##   total edges= 19
##     missing edges= 0
##     non-missing edges= 19
##
##  Vertex attribute names:
##     vertex.names
##
## Date of origin: [1] "2000-01-01"
##
## == @trees==
## 1 phylogenetic trees

summary(ToyOutbreak)

## Dataset of 418 individuals with...
## - 418 samples
##  coming from 418 individuals
##  collected between 2000-01-01 and 2000-01-10
```

```
##   containing information on:
##   sequenceID
##   locus
## - 836 sequences across 2 loci
##  (length of concatenated alignment: 1600 nucleotides)
## - clinical data from 418 individuals
##   containing information on:
##   Fever
## - 19 contacts recorded between 20 individuals
## - 1 phylogenetic tree with 418 tips
```

`ToyOutbreak` is an `obkData` object containing information on individuals (`@individuals`), samples (`@samples`), clinical events (`@clinical`), some DNA sequences (`@dna`), a contact network (`@contacts`) and a phylgenetic tree (`@trees`). Note the presence of the mandatory columns in `@samples`: `individualID`, `sampleID`, and `date`. Accessing a given slot is as easy as:

```
head(ToyOutbreak@individuals)

##   infector DateInfected Sex Age   lat     lon
## 1       NA   2000-01-01   M  33 51.52 -0.1805
## 2        1   2000-01-02   F  42 51.52 -0.1771
## 3        2   2000-01-03   F  44 51.52 -0.1614
## 4        2   2000-01-03   M  49 51.52 -0.1706
## 5        2   2000-01-03   M  34 51.52 -0.1685
## 6        2   2000-01-03   M  31 51.51 -0.1662

head(ToyOutbreak@samples)

##   individualID sampleID       date sequenceID locus
## 1            1        1 2000-01-01          1 gene1
## 2            2        2 2000-01-02          2 gene1
## 3            3        3 2000-01-03          3 gene1
## 4            4        4 2000-01-03          4 gene1
## 5            5        5 2000-01-03          5 gene1
## 6            6        6 2000-01-03          6 gene1

ToyOutbreak@trees

## 1 phylogenetic trees
```

However, we will see how retrieving information from `obkData` objects can be made more powerful using accessors in the following sections.

1.1.2 `obkSequences`: storage of DNA sequences for different genes

Pathogen sequence data can typically be obtained for different genes, making the handling of such information not entirely trivial. The class `obkSequences` stores such information. It consists in a list of matrices of aligned DNA sequences (in rows), stored using *ape*'s class `DNAbin` for efficiency, with each item of the list corresponding to a different gene. If provided, gene names are the names of the list. The row names in each matrix contain unique identifiers for the sequences, typically accession numbers. In `obkData` objects, sequences are matched to samples by the field `sequenceID` in the `@sample` slot, which effectively contains the sequence identifiers. When several loci have been sequenced, the locus

information must also be provided for each sequence identifiers using the column `locus` in the `@sample` slot.

Let us examine the DNA information stored in `ToyOutbreak`:

```
class(ToyOutbreak@dna)

## [1] "obkSequences"
## attr(,"package")
## [1] "epibase"

ToyOutbreak@dna

## [ 836 DNA sequences in 2 loci ]
##
## $gene1
## 418 DNA sequences in binary format stored in a matrix.
##
## All sequences of same length: 600
##
## Labels: 1 2 3 4 5 6 ...
##
## Base composition:
##     a     c     g     t
## 0.237 0.248 0.252 0.263
##
## $gene2
## 418 DNA sequences in binary format stored in a matrix.
##
## All sequences of same length: 1000
##
## Labels: 419 420 421 422 423 424 ...
##
## Base composition:
##     a     c     g     t
## 0.223 0.243 0.257 0.276

slotNames(ToyOutbreak@dna)

## [1] "dna"

is.list(ToyOutbreak@dna@dna)

## [1] TRUE

names(ToyOutbreak@dna@dna)

## [1] "gene1" "gene2"

ToyOutbreak@dna@dna$gene1

## 418 DNA sequences in binary format stored in a matrix.
##
## All sequences of same length: 600
##
```

```
## Labels: 1 2 3 4 5 6 ...
##
## Base composition:
##     a     c     g     t
## 0.237 0.248 0.252 0.263

class(ToyOutbreak@dna@dna$gene1)

## [1] "DNAbin"
```

`ToyOutbreak@dna` is an `obkSequences` object containing DNA sequences for two genes. The slot `ToyOutbreak@dna@dna` is a list of `DNAbin` matrices, each containing sequences for a given gene.

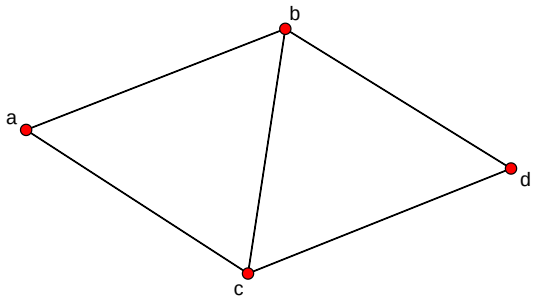
1.1.3 `obkContacts`: storage of dynamics contact networks

obkData objects can also store contact data between individuals, in the slot `@contacts`. These contacts can be fixed or vary in time, in which case data are stored as a dynamic contact network. The slot `@contacts` is an instance of the class `obkContacts`, which currently contains either a `network` object (static graph, from the *network* package), or a `networkDynamic` object, for contacts varying in time (from the *networkDynamic* package). These objects are fully documented in their respective vignettes. Here, we detail a simple toy example from the documentation of `obkContacts`:

```
cf <- c("a", "b", "a", "c", "d")
ct <- c("b", "c", "c", "d", "b")
oc.static <- new("obkContacts", cf, ct, directed=FALSE)
slotNames(oc.static)

## [1] "contacts" "origin"

oc.static

##  Number of individuals = 4
##  Number of contacts = 5
##  Contacts = fixed
##  Network attributes:
##   vertices = 4
##   directed = FALSE
##   hyper = FALSE
##   loops = FALSE
##   multiple = TRUE
##   bipartite = FALSE
##   total edges= 5
##     missing edges= 0
##     non-missing edges= 5
##
##  Vertex attribute names:
##     vertex.names
```

`oc.static` contains a static, non-directed contact network (slot `@contacts`, class `network`). It can be plotted easily using:

```
plot(oc.static, main="Static contact network")
```




```
onset <- c(1, 2, 3, 4, 5)
terminus <- c(1.2, 4, 3.5, 4.1, 6)
oc.dynamic <- new("obkContacts",cf,ct, directed=FALSE,
                  contactStart=onset, contactEnd=terminus)
slotNames(oc.dynamic)

## [1] "contacts" "origin"

oc.dynamic

##  Number of individuals = 4
##  Number of contacts = 5
##  Contacts = dynamic
## NetworkDynamic properties:
##   distinct change times: 9
##   maximal time range: 1 to 6
##
##  Network attributes:
##   vertices = 4
##   directed = FALSE
##   hyper = FALSE
##   loops = FALSE
##   multiple = TRUE
```

```
##   bipartite = FALSE
##   total edges= 5
##     missing edges= 0
##     non-missing edges= 5
##
##  Vertex attribute names:
##     vertex.names
##
## Date of origin: NULL
```

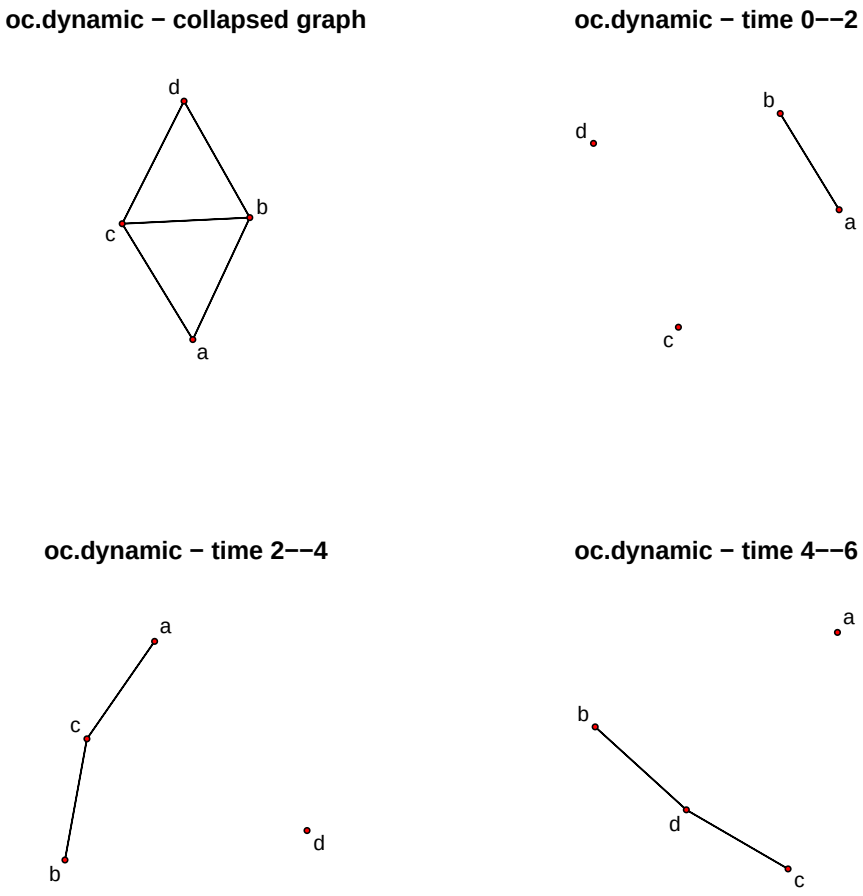
`oc.dynamic` is a dynamic graph, i.e. a graph whose vertices and edges can change over time. By default, plotting the object collapses the graph so that all vertices and edges that exist at some point are displayed; however, sections of the graph for given time intervals can be obtained using `get.contacts` (or alternatively, `network.extract` on the `networkDynamic` object directly). As a reminder, here is the input of the graph `oc.dynamic`:

```
data.frame(onset,terminus,ct,cf)
```

```
##   onset terminus ct cf
## 1     1      1.2  b  a
## 2     2      4.0  c  b
## 3     3      3.5  c  a
## 4     4      4.1  d  c
## 5     5      6.0  b  d
```

And here are various plots, first of the full (collapsed) contact network, then for different time intervals (0–2, 2–4, 4–6):

```
par(mfrow=c(2,2))
plot(oc.dynamic@contacts,main="oc.dynamic - collapsed graph",
     displaylabels=TRUE)
plot(get.contacts(oc.dynamic, from=0, to=2),
     main="oc.dynamic - time 0--2", displaylabels=TRUE)
plot(get.contacts(oc.dynamic, from=2, to=4),
     main="oc.dynamic - time 2--4", displaylabels=TRUE)
plot(get.contacts(oc.dynamic, from=4, to=6),
     main="oc.dynamic - time 4--6", displaylabels=TRUE)
```



networkDynamic allows for extensive manipulation of dynamic networks. For more information, refer to the vignette distributed with the package (`vignette("networkDynamic")`).

1.2 Getting data into *epibase*

Storing data in *epibase* requires the following, fairly simple steps:

1. read data into R
   (a) read `data.frame`s storing individuals, samples, and clinical information in R from a text file, typically using `read.table` or `read.csv` for comma-separated files. Every standard spreadsheet software can export data to these formats.
   (b) read DNA sequences from a single file, typically using `read.dna` from the ape package; this "master" file must contain all DNA sequences of all genes, with unique identifiers for the sequences as labels. While phylogenies can be obtained in R, annotated trees produced by Bayesian software such as BEAST can now be imported using `read.annotated.nexus`.
2. use this information as input to the `obkData` constructor (`new("obkData",...)`) to create an `obkData` object.

In the following, we assume that step 1 is sorted and focus on step 2: using the constructor.

1.2.1 The `obkData` constructor

New objects are created using `new`, with these slots as arguments. If no argument is provided, an empty object is created, as seen before:

```
new("obkData")

##
## === obkData object ===
## == Empty slots ==
##  @individuals, @samples, @clinical, @dna, @contacts, @trees
```

This function accepts the following arguments, which mirror to some extent the structure of the object (see `?obkData` for more information):

- `individuals`: a `data.frame` with a mandatory column named 'individualID', providing unique identifiers for the individuals.
- `samples`: a `data.frame` with 3 mandatory columns named 'individualID', 'sampleID', and 'date', providing identifiers for the individuals, for the samples, and dates. Dates must be provided in a way convertible to `Date` (see `?as.Date`). Default format for dates provided as characters is "%Y-%m-%d" (e.g. 1984-09-23). Alternative format can be specified via the argument `date.format`. If left to `NULL`, the format is determined automatically.
- `clinical`: a list of `data.frame`s, each of which has 2 mandatory fields, individualID' and 'date' (specified as before).
- `dna`: a list of DNA sequences in `DNAbin` or `character` format, as read by `read.dna` or `fasta2DNAbin`.
- `contacts`: a matrix of characters indicating edges using two columns; if contacts are directed, the first column is 'from', the second is 'to'; values should match individual IDs (as returned by `get.individuals(x)`); if numeric values are provided, these are converted to integers and assumed to correspond to individuals returned by `get.individuals(x)`.
- `contacts.start`: a vector of dates indicating the beginning of each contact.
- `contacts.end`: a vector of dates indicating the end of each contact.
- `contacts.duration`: another way to specify `contacts.end`, as duration of contact in days.
- `contacts.directed`: a logical indicating if contacts are directed; defaults to `FALSE`.
- `trees`: a list of phylogenetic trees in the class `multiPhylo` (from the ape package); this is basically a list of `phylo` objects, with the class attribute "`multiPhylo`".
- `date.format`: a character string indicating the date format (see `as.Date`); if `NULL`, date format is detected automatically.

We can now show how to create a new `obkData` from multiple inputs, using the dataset `ToyOutbreakRaw`:

```
data(ToyOutbreakRaw)
class(ToyOutbreakRaw)

## [1] "list"
```

```
names(ToyOutbreakRaw)

## [1] "individuals"    "samples"        "clinical"       "contacts"
## [5] "contacts.start" "contacts.end"   "dna"            "trees"
```

The simplest information we can store is about the samples and individuals:

```
head(ToyOutbreakRaw$samples)

##   individualID sampleID       date sequenceID locus
## 1            1        1 2000-01-01          1 gene1
## 2            2        2 2000-01-02          2 gene1
## 3            3        3 2000-01-03          3 gene1
## 4            4        4 2000-01-03          4 gene1
## 5            5        5 2000-01-03          5 gene1
## 6            6        6 2000-01-03          6 gene1

head(ToyOutbreakRaw$individuals)

##   infector DateInfected Sex Age   lat     lon
## 1       NA   2000-01-01   M  33 51.52 -0.1805
## 2        1   2000-01-02   F  42 51.52 -0.1771
## 3        2   2000-01-03   F  44 51.52 -0.1614
## 4        2   2000-01-03   M  49 51.52 -0.1706
## 5        2   2000-01-03   M  34 51.52 -0.1685
## 6        2   2000-01-03   M  31 51.51 -0.1662

x <- new("obkData", samples=ToyOutbreakRaw$samples,
         individuals=ToyOutbreakRaw$individuals)
head(x)

##
## === obkData x ===
## == @individuals==
##   infector DateInfected Sex Age   lat     lon
## 1       NA   2000-01-01   M  33 51.52 -0.1805
## 2        1   2000-01-02   F  42 51.52 -0.1771
## 3        2   2000-01-03   F  44 51.52 -0.1614
## 4        2   2000-01-03   M  49 51.52 -0.1706
##
## == @samples==
##   individualID sampleID       date sequenceID locus
## 1            1        1 2000-01-01          1 gene1
## 2            2        2 2000-01-02          2 gene1
## 3            3        3 2000-01-03          3 gene1
## 4            4        4 2000-01-03          4 gene1
##
## == Empty slots ==
##  @clinical, @dna, @contacts, @trees
```

This apparently did not do much, as the stored information is nearly identical to the input. However, a number of checks have been made to ensure that dates are actually `Date` objects, that individual identifiers of are indeed unique, etc. More importantly, the two sources of information are

now connected, so that we can later e.g. subset the data per sample or per individual. This is possible for many other types of information we may want to handle. Let us now consider the following, additional inputs, including information about a dynamic contact network:

```
head(ToyOutbreakRaw$contacts)
```

```
##      from to
## [1,]    1  2
## [2,]    2  3
## [3,]    2  4
## [4,]    2  5
## [5,]    2  6
## [6,]    6  7
```

```
head(ToyOutbreakRaw$contacts.start)
```

```
## [1] "2000-01-01" "2000-01-02" "2000-01-02" "2000-01-02" "2000-01-02"
## [6] "2000-01-03"
```

```
head(ToyOutbreakRaw$contacts.end)
```

```
## [1] "2000-01-02" "2000-01-03" "2000-01-03" "2000-01-03" "2000-01-03"
## [6] "2000-01-04"
```

clinical intervention (here, merely temperature measurements):

```
head(ToyOutbreakRaw$clinical$Fever)
```

```
##   individualID       date temperature
## 1            1 2000-01-03        39.1
## 2            2 2000-01-03        40.4
## 3            3 2000-01-07        40.0
## 4            4 2000-01-08        39.8
## 5            5 2000-01-04        39.4
## 6            6 2000-01-06        39.3
```

DNA sequences:

```
ToyOutbreakRaw$dna
```

```
## 836 DNA sequences in binary format stored in a list.
##
## All sequences of same length: 600
##
## Labels: 1 2 3 4 5 6 ...
##
## Base composition:
##     a     c     g     t
## 0.237 0.248 0.252 0.263
```

and even phylogenetic trees:

```
ToyOutbreakRaw$trees

## 1 phylogenetic trees
```

All this information will be compiled into a single object by:

```
x <- new ("obkData", individuals=ToyOutbreakRaw$individuals,
          samples=ToyOutbreakRaw$samples,
          clinical=ToyOutbreakRaw$clinical, contacts=ToyOutbreakRaw$contacts,
          contacts.start=ToyOutbreakRaw$contacts.start,
          contacts.end=ToyOutbreakRaw$contacts.end,
          dna=ToyOutbreakRaw$dna, trees=ToyOutbreakRaw$trees)
head(x)

##
## === obkData x ===
## == @individuals==
##   infector DateInfected Sex Age   lat     lon
## 1       NA   2000-01-01   M  33 51.52 -0.1805
## 2        1   2000-01-02   F  42 51.52 -0.1771
## 3        2   2000-01-03   F  44 51.52 -0.1614
## 4        2   2000-01-03   M  49 51.52 -0.1706
##
## == @samples==
##   individualID sampleID       date sequenceID locus
## 1            1        1 2000-01-01          1 gene1
## 2            2        2 2000-01-02          2 gene1
## 3            3        3 2000-01-03          3 gene1
## 4            4        4 2000-01-03          4 gene1
##
## == @clinical==
##   individualID       date temperature
## 1            1 2000-01-03        39.1
## 2            2 2000-01-03        40.4
## 3            3 2000-01-07        40.0
## 4            4 2000-01-08        39.8
##
## == @dna==
## [ 836 DNA sequences in 2 loci ]
##
## $gene1
## 418 DNA sequences in binary format stored in a matrix.
##
## All sequences of same length: 600
##
## Labels: 1 2 3 4 5 6 ...
##
## Base composition:
##     a     c     g     t
## 0.237 0.248 0.252 0.263
##
## $gene2
```

```
## 418 DNA sequences in binary format stored in a matrix.
##
## All sequences of same length: 600
##
## Labels: 419 420 421 422 423 424 ...
##
## Base composition:
##     a     c     g     t
## 0.237 0.248 0.252 0.263
##
##
## == @contacts==
##  Number of individuals = 20
##  Number of contacts = 19
##  Contacts = dynamic
## NetworkDynamic properties:
##   distinct change times: 5
##   maximal time range: 0 to 4
##
##  Network attributes:
##   vertices = 20
##   directed = FALSE
##   hyper = FALSE
##   loops = FALSE
##   multiple = TRUE
##   bipartite = FALSE
##   total edges= 19
##     missing edges= 0
##     non-missing edges= 19
##
##  Vertex attribute names:
##     vertex.names
##
## Date of origin: [1] "2000-01-01"
##
## == @trees==
## 1 phylogenetic trees

summary(x)

## Dataset of 418 individuals with...
## - 418 samples
##  coming from 418 individuals
##  collected between 2000-01-01 and 2000-01-10
##   containing information on:
##   sequenceID
##   locus
## - 836 sequences across 2 loci
##  (length of concatenated alignment: 1200 nucleotides)
## - clinical data from 418 individuals
##   containing information on:
##   Fever
```

```
## - 19 contacts recorded between 20 individuals
## - 1 phylogenetic tree with 418 tips
```

`x` is a new, coherent representation of the data. This representation ensures, amongst other things, that:

- individual labels are unique and consistent across samples, clinical interventions, contacts, and patient information
- every item is dated using actual dates (`Date` objects), using the same format
- every sample refers to an individual, for which meta-information may be available
- every DNA sequence refers to a sample
- every DNA sequence belongs to a gene
- DNA sequences from the same gene have the same length
- every tip of the trees refers to a DNA sequence
- every contact refers to documented individuals

Moreover, as all these items are now connected, data manipulation on every components will now be drastically simplified (see section below on data handling).

1.2.2 Using other constructors: `obkSequences` and `obkContacts`

The classes `obkSequences` and `obkContacts`, both used in `obkData` objects, also have constructors and can be created independently from `obkData` objects. However, the risk is that one would replace e.g. the DNA sequences stored in an `obkData` object by a new `obkSequences`, which would bypass the consistency checks made by the `obkData` constructor and possibly lead to an invalid object. This practice is therefore discouraged for the moment.

2 Data handling using `obkData` objects

2.1 Accessors

The philosophy underlying formal (S4) classes is that the internal representation of the data can be complex as long as accessing the information is simple. This is made possible by decoupling storage and accession: the user is not meant to access the content of the object directly, but has to use *accessors* to retrieve the information. In this section, we detail the existing accessors for object classes implemented in *epibase*. We use the notation "`[possible-values]`" to list or describe possible values of an argument; the symbols "`[]`" should be omitted from the actual command line. For instance:

```
myFunction(x, y=["foo" or "bar"])
```

means that the argument `y` of function `myFunction` can be either `"foo"` or `"bar"`, and valid calls would be:

```
myFunction(x, y="foo")
```

or:

```
myFunction(x, y="bar")
```

2.1.1 Accessors for `obkData` objects

Available accessors are also documented in `?obkData`. These functions are meant to retrieve information that is not trivially accessible. To simply access slots, use the `@` operator, e.g. `x@samples`, `x@individuals`, etc.

All accessors return `NULL` when information is missing, except for functions returning number of items, which will return `0`. In the following, we illustrate accessors using a random sample (n=5) of the toy dataset `ToyOutbreak`:

```
data(ToyOutbreak)
set.seed(1)
get.nsamples(ToyOutbreak)

## [1] 418

toKeep <- sample(1:nrow(ToyOutbreak@samples), 5)
toKeep

## [1] 222 311 478 757 168

x <- subset(ToyOutbreak, row.samples=toKeep)
summary(x)

## Dataset of 5 individuals with...
## - 5 samples
##  coming from 5 individuals
##  collected between 2000-01-07 and 2000-01-10
##   containing information on:
##   sequenceID
##   locus
## - 5 sequences across 2 loci
##  (length of concatenated alignment: 1600 nucleotides)
## - clinical data from 5 individuals
##   containing information on:
##   Fever
## - 0 contacts recorded between 0 individuals
## - 1 phylogenetic tree with 3 tips
```

- `get.individuals(x, data=["all" or "samples" or "individuals" or "clinical" or "contacts"])`: returns the individual IDs in different components of the object.
- `get.nindividuals(x, data=["all" or "samples" or "individuals" or "clinical" or "contacts"])`: returns the number of individuals in different components of the object.

```
get.nindividuals(x)

## [1] 5
```

```
get.nindividuals(x, "contacts")

## [1] 0

get.individuals(x)

## [1] "222" "311" "60"  "339" "168"
```

There are 5 individuals in the data, except for contact information; this is because contacts were only recorded between the first 20 individuals of `ToyOutbreak`:

```
get.individuals(ToyOutbreak, "contacts")

##  [1] "1"  "2"  "6"  "5"  "4"  "7"  "11" "9"  "3"  "8"  "10" "12" "13" "14" "15"
## [16] "16" "17" "18" "19" "20"
```

- `get.nsamples(x)`: returns the number of sample.
- `get.samples(x)`: returns the unique IDs of the samples in the data.

```
get.nsamples(x)

## [1] 5

get.samples(x)

## [1] "222" "311" "60"  "339" "168"
```

- `get.nlocus(x)`: returns the number of loci.
- `get.locus(x)`: returns the names of the loci in the data.

```
get.nlocus(x)

## [1] 2

get.locus(x)

## [1] "gene1" "gene2"
```

- `get.nsequences(x, what=["total" or "bylocus"])`: returns the number of sequences in `@dna`.
- `get.sequences(x)`: returns the IDs of the sequences in `@dna`.

```
get.nsequences(x)

## [1] 5
```

```
get.nsequences(x, "bylocus")

## gene1 gene2
##     3     2

get.sequences(x)

## gene11 gene12 gene13 gene21 gene22
##  "222"  "311"  "168"  "478"  "757"
```

- `get.trees(x)`: returns the content of `x@trees`.

```
get.trees(x)

## 1 phylogenetic trees
```

- `get.dna(x, locus=[`*locus IDs*`], id=[`*sequence IDs*`])`: returns a list of matrices of DNA sequences; the arguments `locus` and `id` are optional; if provided, they should be character strings corresponding to the name of the loci and/or sequences to be retained. Integers or logical will be treated as indicators based on the results of `get.locus` or `get.sequences`.

```
get.dna(x)

## $gene1
## 3 DNA sequences in binary format stored in a matrix.
##
## All sequences of same length: 600
##
## Labels: 222 311 168
##
## Base composition:
##     a     c     g     t
## 0.238 0.248 0.251 0.263
##
## $gene2
## 2 DNA sequences in binary format stored in a matrix.
##
## All sequences of same length: 1000
##
## Labels: 478 757
##
## Base composition:
##     a     c     g     t
## 0.223 0.236 0.260 0.281
```

returns all the DNA sequences, in two matrices corresponding to the different genes. We can request e.g. only the second gene:

```
get.dna(x, locus=2)

## $gene2
## 2 DNA sequences in binary format stored in a matrix.
##
## All sequences of same length: 1000
##
## Labels: 478 757
##
## Base composition:
##     a     c     g     t
## 0.223 0.236 0.260 0.281
```

or even just specific sequences, say ("311" and "222"):

```
get.dna(x, id=c("311","222"))

## $gene1
## 2 DNA sequences in binary format stored in a matrix.
##
## All sequences of same length: 600
##
## Labels: 311 222
##
## Base composition:
##     a     c     g     t
## 0.237 0.248 0.251 0.264
```

Note that we could also refer to sequences by their index in `get.sequences`:

```
get.sequences(x)

## gene11 gene12 gene13 gene21 gene22
##  "222"  "311"  "168"  "478"  "757"

identical(get.dna(x, id=c("311","222")), get.dna(x, id=c(2,1)))

## [1] TRUE
```

- `get.ncontacts(x, from=NULL, to=NULL)`: returns the number of contacts in `x@contacts`; the optional arguments `from` and `to` can be used, in the case of dynamic networks, to specify the range of dates for which contacts should be kept.
- `get.contacts(x, from=NULL, to=NULL)`: returns the contacts in `x@contacts`; the optional arguments `from` and `to` can be used, in the case of dynamic networks, to specify the range of dates for which contacts should be kept. Here, the object `x` contains no contact information, as the individuals of the samples retained were had no documented contacts:

```
get.ncontacts(ToyOutbreak)
```

```
## [1] 19

get.individuals(ToyOutbreak@contacts)

##  [1] "1"  "2"  "6"  "5"  "4"  "7"  "11" "9"  "3"  "8"  "10" "12" "13" "14" "15"
## [16] "16" "17" "18" "19" "20"

get.individuals(x)

## [1] "222" "311" "60"  "339" "168"

get.ncontacts(x)

## [1] 0
```

- `get.data(x, data=[`*`name of data seeked`*`], where=NULL, drop=[`*`TRUE/FALSE`*`], showSource=[`*`TRUE/FALSE`*`])`: multi-purpose accessor seeking a data field with a given name in the entire dataset; `data` can be the name of a slot, or the name of a column in `x@individuals`, `x@samples`, or `x@clinical`. The optional argument `where` allows one to specify in which slot the information should be looked for. The argument `drop` states whether to return a vector (`TRUE`), or a one-column `data.frame` (`FALSE`).

  For instance, we can retrieve temperature measurements using:

```
get.data(x,"temperature")

## [1] 39.2 39.7 38.7 39.6 38.5
```

  or the sex of the different individuals:

```
get.data(x, "Sex")

## [1] "F" "M" "M" "F" "F"
```

  Several fields can be requested, so long as they are stored in the same slot; for instance:

```
get.data(x, c("Sex","Age","infector"))

##     Sex Age infector
## 60    F  37       27
## 168   M  37      106
## 222   M  39       43
## 311   F  32      210
## 339   F  50      159
```

  The source (where matching fields were found) will be indicated if `showSource` is `TRUE`:

```
get.data(x, c("Sex","Age","infector"), showSource=TRUE)
```

```
##     Sex Age infector individualID      source
## 60    F  37       27           60 individuals
## 168   M  37      106          168 individuals
## 222   M  39       43          222 individuals
## 311   F  32      210          311 individuals
## 339   F  50      159          339 individuals
```

This is especially useful when the same field appears in different slots, such as `date`:

```
get.data(x, "date")

##  [1] "2000-01-09" "2000-01-10" "2000-01-07" "2000-01-10" "2000-01-09"
##  [6] "2000-01-10" "2000-01-15" "2000-01-13" "2000-01-11" "2000-01-16"
```

actually corresponds to:

```
get.data(x, "date", showSource=TRUE)

##            date individualID  source
## 222  2000-01-09          222 samples
## 311  2000-01-10          311 samples
## 478  2000-01-07           60 samples
## 757  2000-01-10          339 samples
## 168  2000-01-09          168 samples
## 60   2000-01-10           60   Fever
## 1681 2000-01-15          168   Fever
## 2221 2000-01-13          222   Fever
## 3111 2000-01-11          311   Fever
## 339  2000-01-16          339   Fever
```

as there are dates in both `@samples` and `@clinical`. To retain only the latter, we use the argument `where`:

```
get.data(x, "date", where="clinical", showSource=TRUE)

##           date individualID source
## 60  2000-01-10           60  Fever
## 168 2000-01-15          168  Fever
## 222 2000-01-13          222  Fever
## 311 2000-01-11          311  Fever
## 339 2000-01-16          339  Fever
```

A failed search will return NULL with a warning; for instance, we can try searching for "sugarman":

```
get.data(x, "sugarman")

## Warning:  data ' sugarman 'was not found in the object

## NULL
```

And the same happens when looking for information in an empty slot:

```
x@clinical <- NULL
get.data(x, "date", where="clinical")

## Warning:  x@clinical is NULL

## NULL
```

2.1.2 Accessors for `obkSequences` objects

Accessors of `obkSequences` objects are basically a subset of what is available for `obkData`. They work in the same way, and use the same arguments; they include:

- `get.locus`
- `get.nlocus`
- `get.sequences`
- `get.nsequences`
- `get.dna`

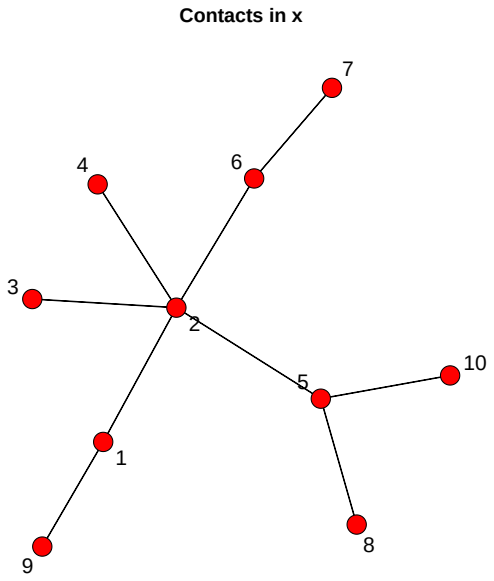
2.1.3 Accessors for `obkContacts` objects

Accessors of `obkContacts` objects are basically a subset of what is available for `obkData`. They work in the same way, and use the same arguments; they include:

- `get.nindividuals`
- `get.individuals`
- `get.ncontacts`
- `get.contacts`

Another useful function is `as.matrix`, which converts the object into an adjacency matrix (by default)), a matrix of incidence, or a matrix listing edges. For instance, using a graph derived from the first 10 individuals in `ToyOutbreak`:

```
x <- subset(ToyOutbreak, individuals=1:10)
get.ncontacts(x)

## [1] 9

plot(x@contacts, main="Contacts in x", label.cex=1.25, vertex.cex=2)
```



(note: see `?plot.network` to customize such graphics).

```
as.matrix(x@contacts)
```

```
##    1 2 6 5 4 7 9 3 8 10
## 1  0 1 0 0 0 0 1 0 0  0
## 2  1 0 1 1 1 0 0 1 0  0
## 6  0 1 0 0 0 1 0 0 0  0
## 5  0 1 0 0 0 0 0 0 1  1
## 4  0 1 0 0 0 0 0 0 0  0
## 7  0 0 1 0 0 0 0 0 0  0
## 9  1 0 0 0 0 0 0 0 0  0
## 3  0 1 0 0 0 0 0 0 0  0
## 8  0 0 0 1 0 0 0 0 0  0
## 10 0 0 0 1 0 0 0 0 0  0
```

```
as.matrix(x@contacts, "edgelist")
```

```
##       [,1] [,2]
##  [1,]    2    1
##  [2,]    8    2
##  [3,]    5    2
##  [4,]    4    2
##  [5,]    3    2
##  [6,]    6    3
##  [7,]    9    4
##  [8,]    7    1
##  [9,]   10    4
## attr(,"n")
## [1] 10
```

```
## attr(,"vnames")
##  [1] "1"  "2"  "6"  "5"  "4"  "7"  "9"  "3"  "8"  "10"
```

2.2 Subsetting the data

A lot of data handling lies in creating subsets of the data based on some given criteria. The method `subset` for `obkData` objects allows for a range of manipulations. The syntax is as follows:

```
subset(x, individuals=NULL, samples=NULL, locus=NULL, sequences=NULL,
       date.from=NULL, date.to=NULL, date.format=NULL,
       row.individuals=NULL, row.samples=NULL,...)
```

See `?subset.obkData` for the details of these arguments. The function works in a fairly intuitive way. The arguments `individuals`, `samples`, `locus` and `sequences` are vectors of characters indicating items to be kept. If integers or logicals are provided, these are assumed to match the output of `get.[...]`. For instance (using a random subset of `ToyOutbreak`):

```
data(ToyOutbreak)
set.seed(1)
get.nsamples(ToyOutbreak)

## [1] 418

toKeep <- sample(1:nrow(ToyOutbreak@samples), 10)
toKeep

##  [1] 222 311 478 757 168 747 785 548 521  52

x <- subset(ToyOutbreak, row.samples=toKeep)
summary(x)

## Dataset of 10 individuals with...
## - 10 samples
##  coming from 10 individuals
##  collected between 2000-01-07 and 2000-01-10
##   containing information on:
##   sequenceID
##   locus
## - 10 sequences across 2 loci
##  (length of concatenated alignment: 1600 nucleotides)
## - clinical data from 10 individuals
##   containing information on:
##   Fever
## - 0 contacts recorded between 0 individuals
## - 1 phylogenetic tree with 4 tips

get.individuals(x)

##  [1] "222" "311" "60"  "339" "168" "329" "367" "130" "103" "52"
```

To retain only individuals 60 and 168, once can do:

```
x1 <- subset(x, indiv=c("60","168"))
```

or

```
x2 <- subset(x, indiv=c(3,5))
identical(x1,x2)

## [1] TRUE
```

Another, non-exclusive way of subsetting the data is using collection dates of the samples. The arguments `date.from` and `date.to` are used for indicating the range of dates of samples to be retained. For instance, the range of data in the influenza H1N1 pandemic dataset `FluH1N1pdm2009` is:

```
data(FluH1N1pdm2009)
x <- new("obkData", individuals = FluH1N1pdm2009$individuals, samples =
         FluH1N1pdm2009$samples, dna = FluH1N1pdm2009$dna, trees =
         FluH1N1pdm2009$trees)
range(get.data(x, "date", where="samples"))

## [1] "2009-03-24" "2009-09-30"
```

We can retain data collected during the first month using:

```
min.date <- min(get.data(x, "date", where="samples"))
min.date

## [1] "2009-03-24"

min.date+31

## [1] "2009-04-24"

x1 <- subset(x, date.to=min.date+31)
summary(x)

## Dataset of 514 individuals with...
## - 514 samples
##  coming from 514 individuals
##  collected between 2009-03-24 and 2009-09-30
##   containing information on:
##   sequenceID
## - 514 sequences across 1 locus
##  (length of concatenated alignment: 1664 nucleotides)
## - clinical data from 0 individuals
## - 1 phylogenetic tree with 514 tips

summary(x1)

## Dataset of 12 individuals with...
## - 12 samples
##  coming from 12 individuals
##  collected between 2009-03-24 and 2009-04-24
##   containing information on:
```

```
##   sequenceID
## - 12 sequences across 1 locus
##  (length of concatenated alignment: 1664 nucleotides)
## - clinical data from 0 individuals
## - 1 phylogenetic tree with 12 tips
```

Note that dates can also be provided as character strings in any sensible format, in which case `subset` detects it automatically.

A third way of specifying subsets of data is using indexing of the rows of `@individuals` or `@samples`, using the arguments `row.individuals` and `row.samples`, respectively. This is particularly useful for instance for select specific test outcomes (e.g. positive swabs), patients within a given age class or of a given sex, or data from a given location. For instance, we can retain data from Mexico using:

```
toKeep <- get.data(x, "location")=="Mexico"
sum(toKeep)

## [1] 43

x.mex <- subset(x, row.individuals=toKeep)
summary(x.mex)

## Dataset of 43 individuals with...
## - 43 samples
##  coming from 43 individuals
##  collected between 2009-03-24 and 2009-09-25
##   containing information on:
##   sequenceID
## - 43 sequences across 1 locus
##  (length of concatenated alignment: 1664 nucleotides)
## - clinical data from 0 individuals
## - 1 phylogenetic tree with 43 tips

head(x.mex)

##
## === obkData x ===
## == @individuals==
##     location
## 246   Mexico
## 247   Mexico
## 248   Mexico
## 249   Mexico
##
## == @samples==
##     individualID sampleID       date                                sequenceID
## 246          246      246 2009-07-11 A/Merida/2189_CIR/2009_Mexico_2009-07-11
## 247          247      247 2009-04-15     A/Mexico/4269/2009_Mexico_2009-04-15
## 248          248      248 2009-04-14     A/Mexico/4482/2009_Mexico_2009-04-14
## 249          249      249 2009-04-19     A/Mexico/4603/2009_Mexico_2009-04-19
##
## == @dna==
## [ 43 DNA sequences in 1 locus ]
```

```
##
## [[1]]
## 43 DNA sequences in binary format stored in a matrix.
##
## All sequences of same length: 1664
##
## Labels: A/Merida/2189_CIR/2009_Mexico_2009-07-11 A/Mexico/4269/2009_Mexico_2009-04-15 A/Mexico/448
##
## Base composition:
##     a     c     g     t
## 0.354 0.186 0.223 0.237
##
##
## == @trees==
## 1 phylogenetic trees
##
## == Empty slots ==
##  @clinical, @contacts
```

Finally, note that several filters can be specified at the same time. For instance, in the following we extract European data collected between the 1st June and the 31st August:

```
x.summerEur <- subset(x, date.from="01/06/2009", date.to="31/08/2009",
                      row.indiv=get.data(x, "location")=="Europe")
summary(x.summerEur)

## Dataset of 30 individuals with...
## - 30 samples
##  coming from 30 individuals
##  collected between 2009-06-01 and 2009-08-26
##   containing information on:
##   sequenceID
## - 30 sequences across 1 locus
##  (length of concatenated alignment: 1664 nucleotides)
## - clinical data from 0 individuals
## - 1 phylogenetic tree with 30 tips

head(x.summerEur)

##
## === obkData x ===
## == @individuals==
##    location
## 73   Europe
## 74   Europe
## 75   Europe
## 76   Europe
##
## == @samples==
##    individualID sampleID       date                                sequenceID
## 73           73       73 2009-08-05 A/Catalonia/S1206/2009_Europe_2009-08-05
## 74           74       74 2009-08-19 A/Catalonia/S1254/2009_Europe_2009-08-19
```

```
## 75           75       75 2009-08-21 A/Catalonia/S1271/2009_Europe_2009-08-21
## 76           76       76 2009-08-25 A/Catalonia/S1272/2009_Europe_2009-08-25
##
## == @dna==
## [ 30 DNA sequences in 1 locus ]
##
## [[1]]
## 30 DNA sequences in binary format stored in a matrix.
##
## All sequences of same length: 1664
##
## Labels: A/Catalonia/S1206/2009_Europe_2009-08-05 A/Catalonia/S1254/2009_Europe_2009-08-19 A/Catalo
##
## Base composition:
##     a     c     g     t
## 0.354 0.187 0.224 0.235
##
##
## == @trees==
## 1 phylogenetic trees
##
## == Empty slots ==
##  @clinical, @contacts
```

2.3 Obtaining phylogenies from genetic sequences

The package *ape* implements a wide range of genetic distances (see `?dist.dna`) and most usual algorithms for distance-based phylogenetic reconstruction. In *epibase*, the function `make.phylo` is a wrapper for these methods, allowing to derive trees for a selection or all the genes present in an `obkData` object. Trees can be stored in the `obkData` (`result='obkData'`) or returned as a `multiPhylo` object (`result='multiPhylo'`). We illustrate this procedure using `x.summerEur`, the data of pandemic H1N1 influenza collected in Europe during the summer 2009 (see previous section):

```
x.summerEur@trees <- NULL
get.nsequences(x.summerEur)

## [1] 30
```

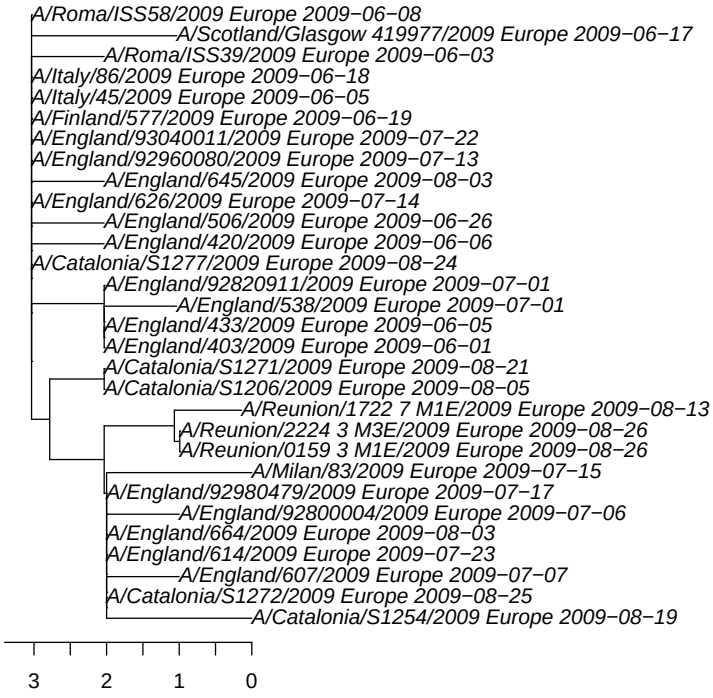
`make.phylo` admits a range of arguments allowing to select which genes (`locus`), model of evolution (`model`), and tree reconstruction method (`method`) should be used. By default, a Neighbour-Joining tree based on Hamming distances (number of differing nucleotides) is derived for every gene, and the resulting trees are plotted:

```
x2 <- make.phylo(x.summerEur)
summary(x2)

## Dataset of 30 individuals with...
## - 30 samples
##  coming from 30 individuals
##  collected between 2009-06-01 and 2009-08-26
##   containing information on:
```

```
##   sequenceID
## - 30 sequences across 1 locus
##  (length of concatenated alignment: 1664 nucleotides)
## - clinical data from 0 individuals
## - 1 phylogenetic tree with 30 tips
```

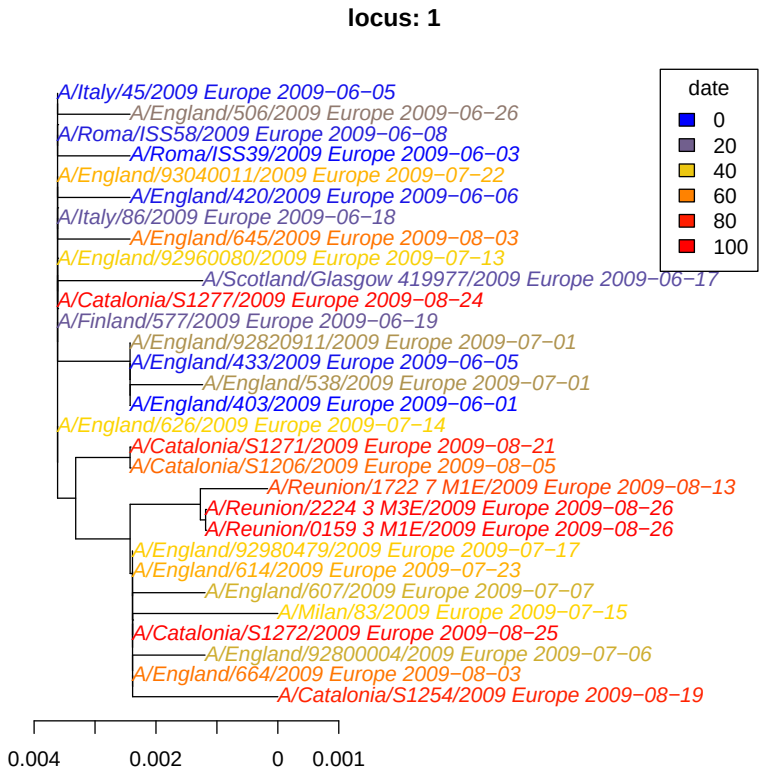
`x2` now contains a phylogenetic tree derived from the sequences in `x.summerEur`. This one can be plotted simply, using:

```
plot(get.trees(x2)[[1]])
axisPhylo()
```



Note that we could ask for a different model of evolution, for instance Kimura's 2 parameters distance, and we may want to display the tree and indicate collection dates using tip colors; this can be done by:

```
tree1 <- make.phylo(x.summerEur, locus=1, ask=FALSE, model="K80",
                    plot=TRUE, color.by="dat", palette=seasun)
axisPhylo()
```



Finally, note that *epibase* also integrates functions to read annotated trees with Newick (`read.annotated.tree`) or NEXUS (`read.annotated.nexus`) formats. This will be particularly useful to process the outputs of Bayesian phylogenetic reconstruction software such as BEAST. See `?read.annotated.nexus` for more information.

3 Simulating outbreak data

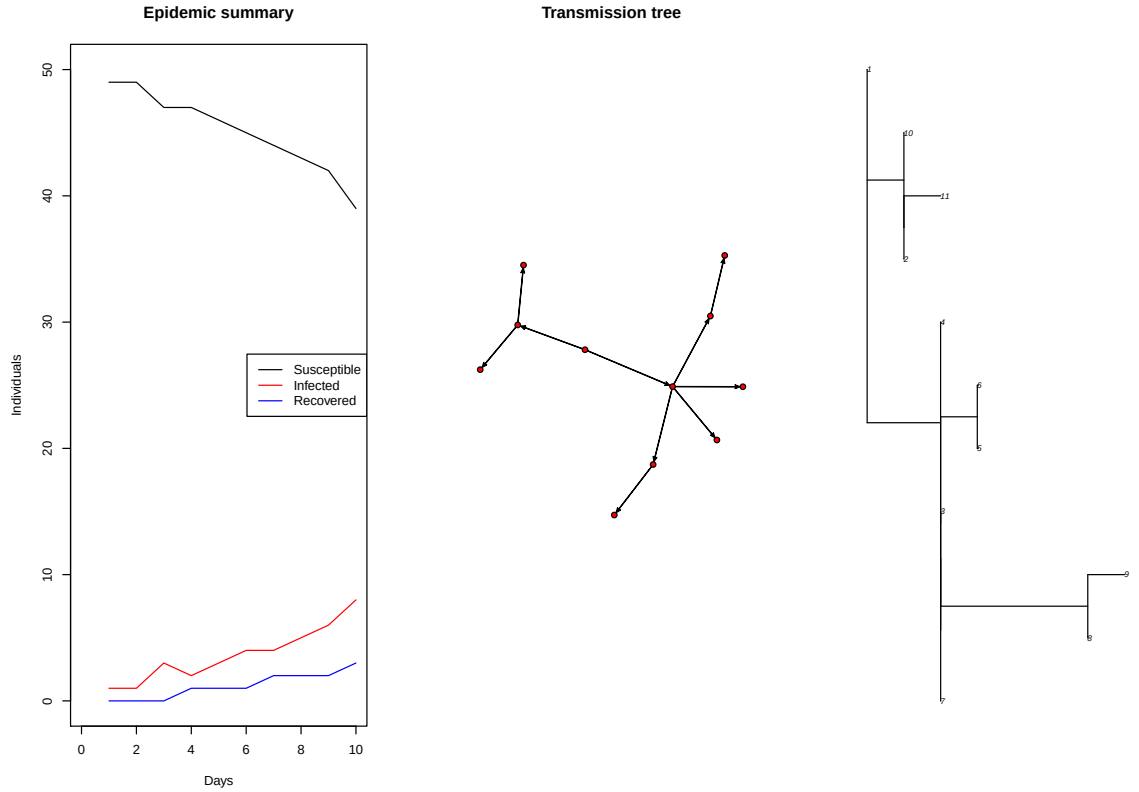
epibase provides some basic functionality for the simulation of outbreak data through the `simuEpi` function. A basic SIR (susceptible-infectious-removed) model is assumed, and the result is returned as an `obkData` object. The arguments are as follows:

- `N`: the size of the population, which remains constant throughout. The simulation will start with one infectious individual, `N-1` susceptibles and zero removed. Default is `N=1000`.
- `D`: duration of the simulation, in days. Default is `D=10`.
- `beta`: probability that a susceptible individual becomes infected by a given infectious individual on a given day. Default is `beta=0.001`.
- `nu`: rate of recovery, ie the probability that an infectious individual becomes removed on a given day. Default is `nu=10`.
- `L`: length of genetic sequences to be generated. Default is `L=1000`.
- `mu`: rate of mutation per site per transmission event. Default is `mu=0.001`.

- `showPlots`: logical indicating whether to plot the SIR trajectory over time, the transmission tree, and the phylogenetic tree if created. Default is `showPlots=FALSE`.
- `makePhyloTree`: logical indicating whether to create a neighbor-joining tree from the simulated sequences. Default is `makePhyloTree=FALSE`.

Let us look at an example in a very small population of size `N=50` and with the infectious rate `beta` raised accordingly to generate a few transmission events:

```
set.seed(1)
x <- simuEpi(N=50,beta=0.01,showPlots=TRUE,makePhylo=TRUE)
```




```
summary(x)

## Dataset of 11 individuals with...
## - 11 samples
##  coming from 11 individuals
##  collected between 2000-01-01 and 2000-01-10
##   containing information on:
##   sequenceID
## - 11 sequences across 1 locus
##  (length of concatenated alignment: 1000 nucleotides)
## - clinical data from 0 individuals
## - 1 phylogenetic tree with 11 tips
```

We can see that 11 individuals got infected over the default period of `D=10` days during which the outbreak was simulated. The panel on the left shows the trajectories for the number of susceptible,

infectious and removed individuals over time. The panel in the middle shows the transmission tree. The panel on the right shows a Neighbor-Joining tree based on the simulated sequence data.

4 Graphics for `obkData` objects

Several plotting options are available for `obkData`, corresponding to different sub-functions (see `?plot.obkData`). The syntax to use is `plot(x, y=["timeline" or "geo" or "mst" or "phylo" or "contacts"], ...)` where `x` is an *obkData* object, and `y` indicates the type of graphic to generate. Further arguments can be passed via `...`. The different types of graphics are:

- '`timeline`': plots the timeline of the outbreak; the timeline of every case is plotted in a single window; uses `plotIndividualTimeline`.
- '`geo`' plots the cases on a map. Needs geographical information. Uses `plotGeo`.
- '`mst`': plots a minimal spanning tree of the genetic data. Uses `plotggMST`.
- '`phylo`': plots a phylogenetic tree of the genetic data. Uses `plotggphy`.
- '`contacts`': plots a phylogenetic tree of the genetic data. Uses the plot method for *obkContacts*.

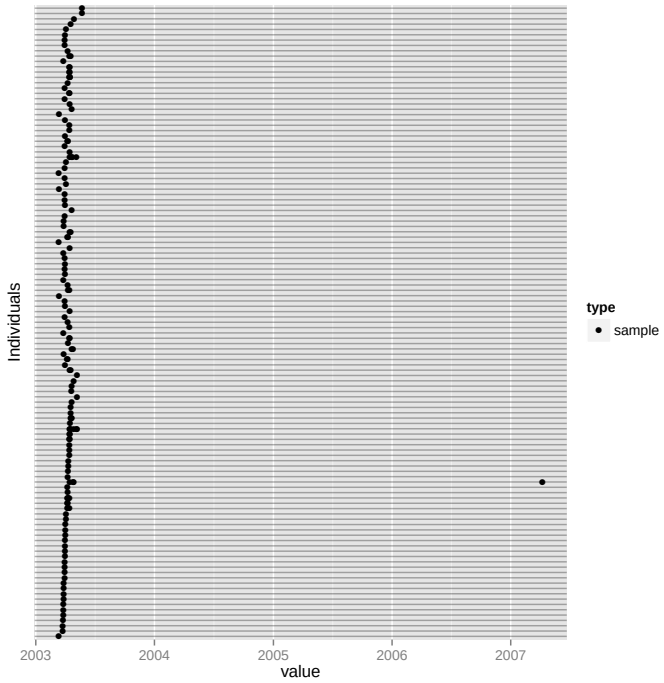
4.1 Plotting a timeline of samples

This plotting option relies on the function `plotIndividualTimeline`; see `?plotIndividualTimeline` for more information. Let's plot the outbreak of equine influenza provided in `HorseFlu`:

```
data(HorseFlu)
summary(HorseFlu)

## Dataset of 119 individuals with...
## - 154 samples
##  coming from 119 individuals
##  collected between 2003-03-13 and 2007-04-09
##   containing information on:
##   shedding (mean: 6405.37, sd: 27377.6)
##   sequenceID
## - 2291 sequences across 1 locus
##  (length of concatenated alignment: 903 nucleotides)
## - clinical data from 85 individuals
##   containing information on:
##   FirstVac
##   LastVac
```

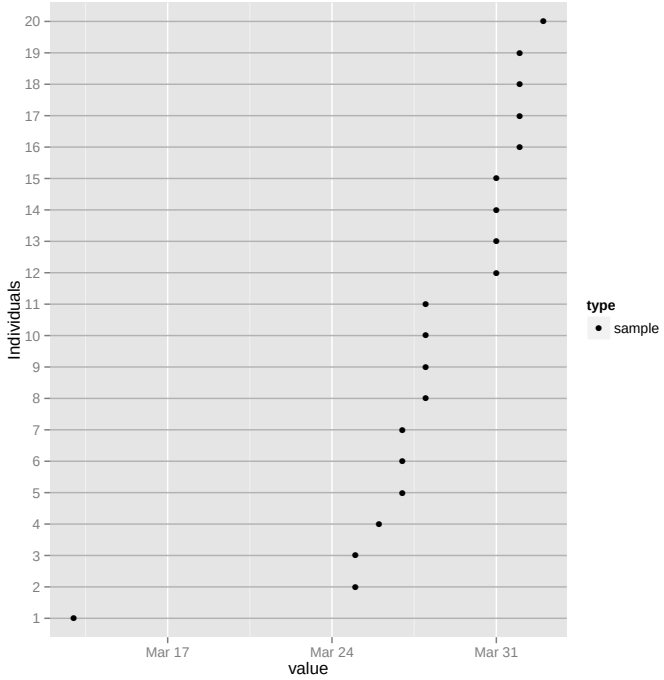
then plot

```
plot(HorseFlu,'timeline')
```



These are a lot of horses in one plot, and we may want to restrict the plot to a selection of individuals. We can do this by a vector specifying the indices of the individuals to plot. Lets plot the first twenty:

```
plot(HorseFlu,selection=1:20)
```

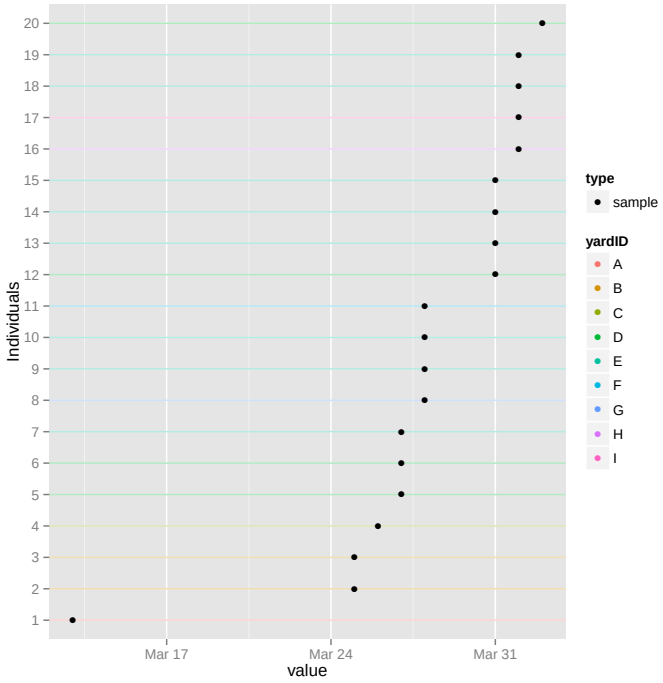


Notice that the names of the individuals are now plotted. The default behaviour is to plot these when less than fifty individuals are plotted, but we can manually override this by setting `plotNames`.

The plotting of sampling times is toggled by `plotSamples`. This defaults to TRUE, as an error will be generated when no 'date' fields can be found to plot, as would be the case for the equine dataset.

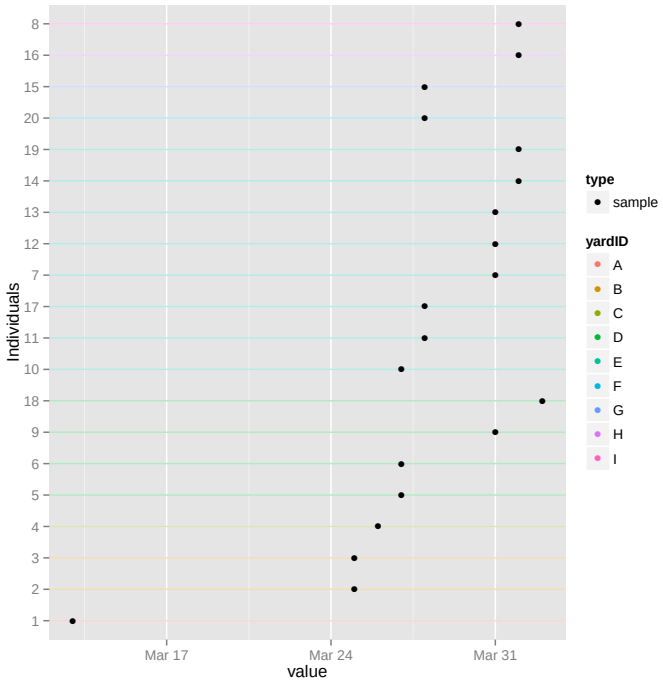
We can also colour individuals by a characteristic provided in the `obkData` object. In this case, let us colour the horses by the yard they were in, a column called 'yardID'

```
plot(HorseFlu,selection=1:20,colorBy='yardID')
```



It might be useful to also order the individuals, which can be done according to some provided information (here, the yard) using `orderBy`:

```
plot(HorseFlu,selection=1:20,colorBy='yardID',orderBy='yardID')
```



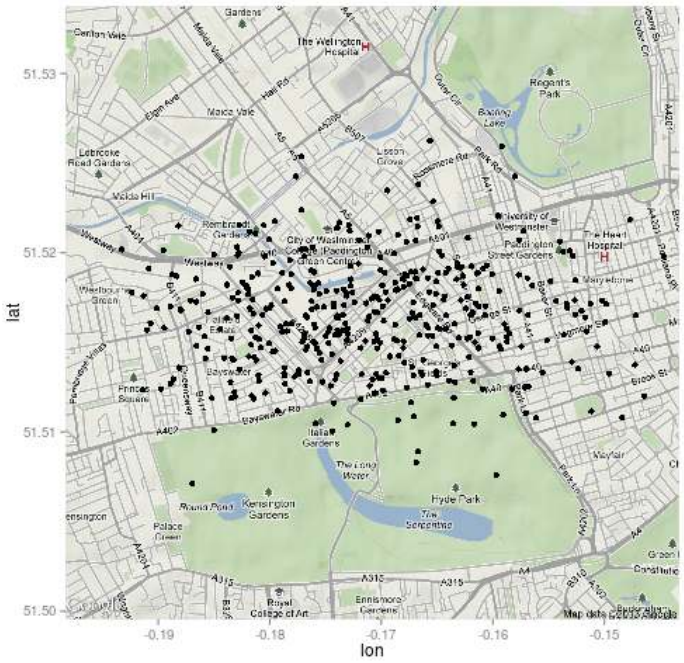
4.2 Visualizing samples on a map

If geographical information is available, the function `plotGeo` can be used to visualize the cases on a map (which is by default downloaded from googlemaps). `plotGeo` is the function used by the generic `plot` of `obkData` when the second argument is `'geo'`. Geographical information can be provided as longitude/latitudes, or as strings specifying locations (which are converted to lon/lat using googlemaps). Let us plot the toy outbreak already used before, and which already contains longitudes and latitudes.

```
data(ToyOutbreak)
head(ToyOutbreak@individuals)

##   infector DateInfected Sex Age   lat     lon
## 1       NA   2000-01-01   M  33 51.52 -0.1805
## 2        1   2000-01-02   F  42 51.52 -0.1771
## 3        2   2000-01-03   F  44 51.52 -0.1614
## 4        2   2000-01-03   M  49 51.52 -0.1706
## 5        2   2000-01-03   M  34 51.52 -0.1685
## 6        2   2000-01-03   M  31 51.51 -0.1662
```

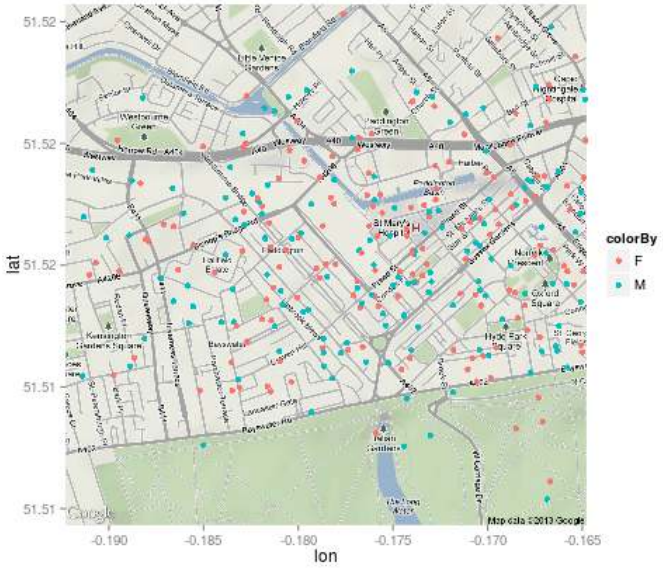
We specify the columns holding these data with `location`, and we have to tell the function that these are valid lon/lat with 'isLonLat' (which defaults to FALSE):

```
plot(ToyOutbreak,'geo', location=c('lon','lat'), isLonLat=TRUE, zoom=14)
```



We can also colour individuals by a certain characteristic using `colorBy` (here, by sex), and even centre the map on a given individual using `center`:

```
plot(ToyOutbreak,'geo', location=c('lon','lat'), isLonLat=TRUE, zoom=15,
     colorBy='Sex', center='11')
```

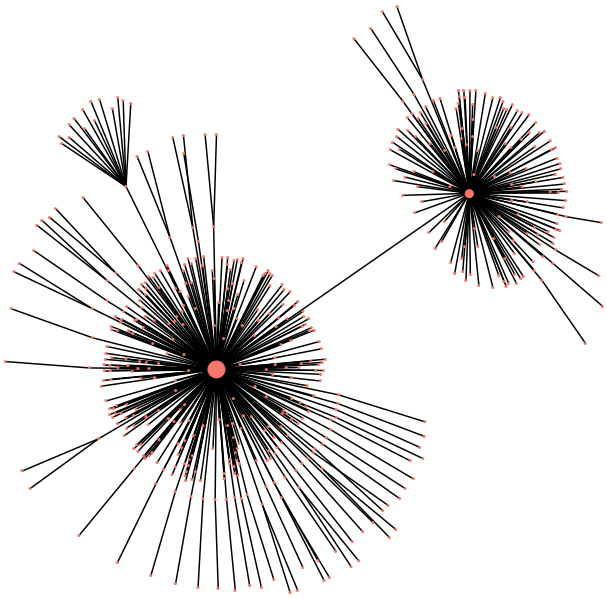


4.3 Building minimum spanning trees from genetic sequences

This plotting option relies on the function `plotggMST`; see `?plotggMST` for more information.

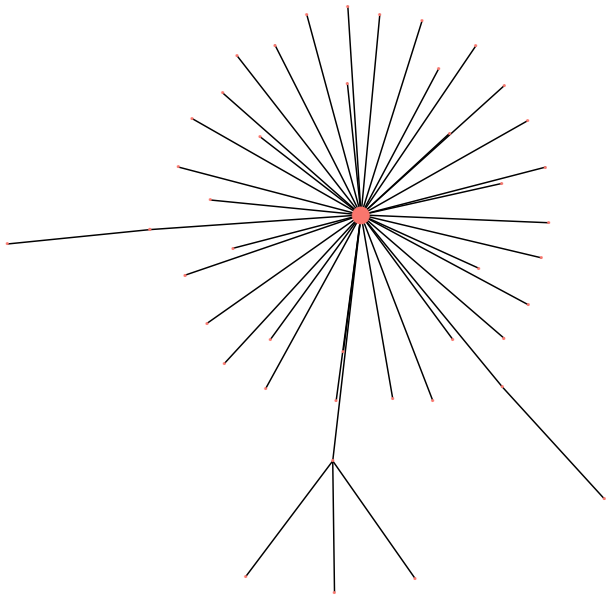
It can be useful to plot a minimal spanning tree of the sequences, to quickly visualize the genetic diversity and the relation between sequences. This can be achieved using `plotggMST`, or simply `plot` using `mst` for the second argument:

```
data(HorseFlu)
plot(HorseFlu,'mst')

## [1] 1
```



this is a large tree, we can also look at the diversity within one individual, e.g. individual 42:

```
plot(HorseFlu,'mst',individualID=42)

## [1] 1
```



4.4 Plotting phylogenetic trees

Phylogenies stored in `obkData` (slot `@trees`) can be plotted using `plotggphy`. This function can be particularly useful as it allows for taking the collection dates into account and for plotting a time tree (where branch length represent time, rather than quantity of evolution). We illustrate this function using data on pandemic influenza stored in `FluH1N1pdm2009`. We first create an `obkData`:

```
data(FluH1N1pdm2009)
x <- new("obkData", individuals = FluH1N1pdm2009$individuals,
         samples = FluH1N1pdm2009$samples, dna = FluH1N1pdm2009$dna,
         trees = FluH1N1pdm2009$trees)
head(x)

##
## === obkData x ===
## == @individuals==
##      location
## 1 CentralAsia
## 2 CentralAsia
## 3   USACanada
## 4      Europe
##
## == @samples==
##   individualID sampleID       date
## 1            1        1 2009-09-12
## 2            2        2 2009-09-12
## 3            3        3 2009-07-04
## 4            4        4 2009-04-29
##                                       sequenceID
```

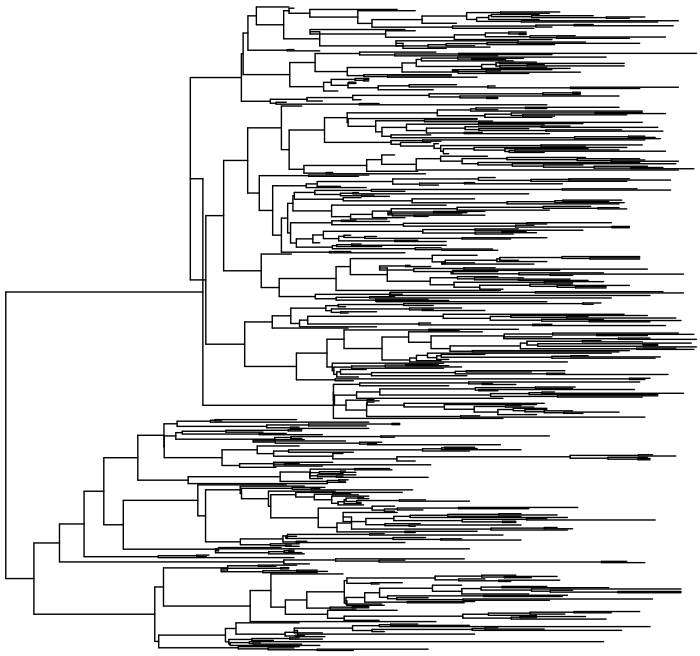
```
## 1 A/Afghanistan/N10782/2009_CentralAsia_2009-09-12
## 2 A/Afghanistan/N10790/2009_CentralAsia_2009-09-12
## 3        A/Alaska/AF2096/2009_USACanada_2009-07-04
## 4         A/Andalucia/GP230/2009_Europe_2009-04-29
##
## == @dna==
## [ 514 DNA sequences in 1 locus ]
##
## [[1]]
## 514 DNA sequences in binary format stored in a matrix.
##
## All sequences of same length: 1664
##
## Labels: A/Mexico_City/WR1704T/2009_Mexico_2009-09-25 A/Pernambuco/609/2009_SouthAmerica_2009-09-22
##
## Base composition:
##     a     c     g     t
## 0.354 0.187 0.223 0.236
##
##
## == @trees==
## 1 phylogenetic trees
##
## == Empty slots ==
##  @clinical, @contacts
```

The phylogenie(s) contained in `x` can be extracted by:

```
get.trees(x)

## 1 phylogenetic trees

tre <- get.trees(x)[[1]]
tre

##
## Phylogenetic tree with 514 tips and 513 internal nodes.
##
## Tip labels:
##  A/Afghanistan/N10782/2009_CentralAsia_2009-09-12, A/Afghanistan/N10790/2009_CentralAsia_2009-09-1
##
## Rooted; includes branch lengths.
```

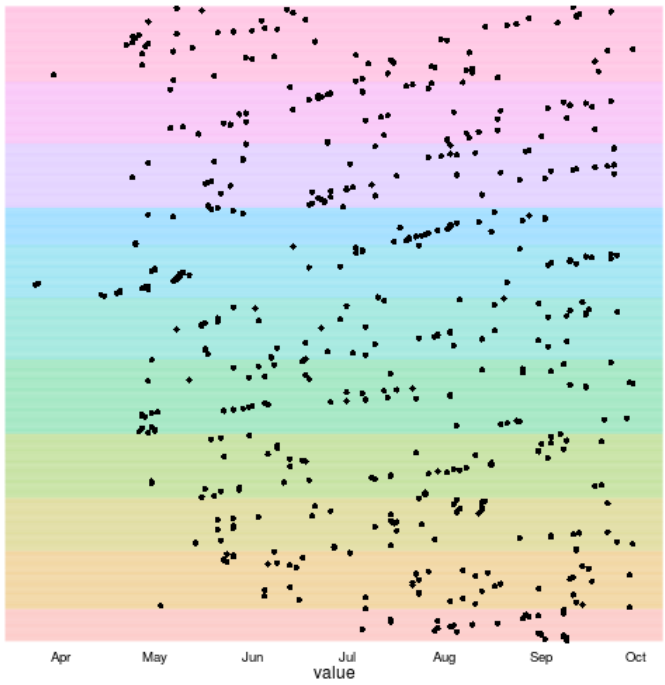
and plotted using *ape*'s standard `plot` function:

```
plot(get.trees(x)[[1]], show.tip=FALSE)
```



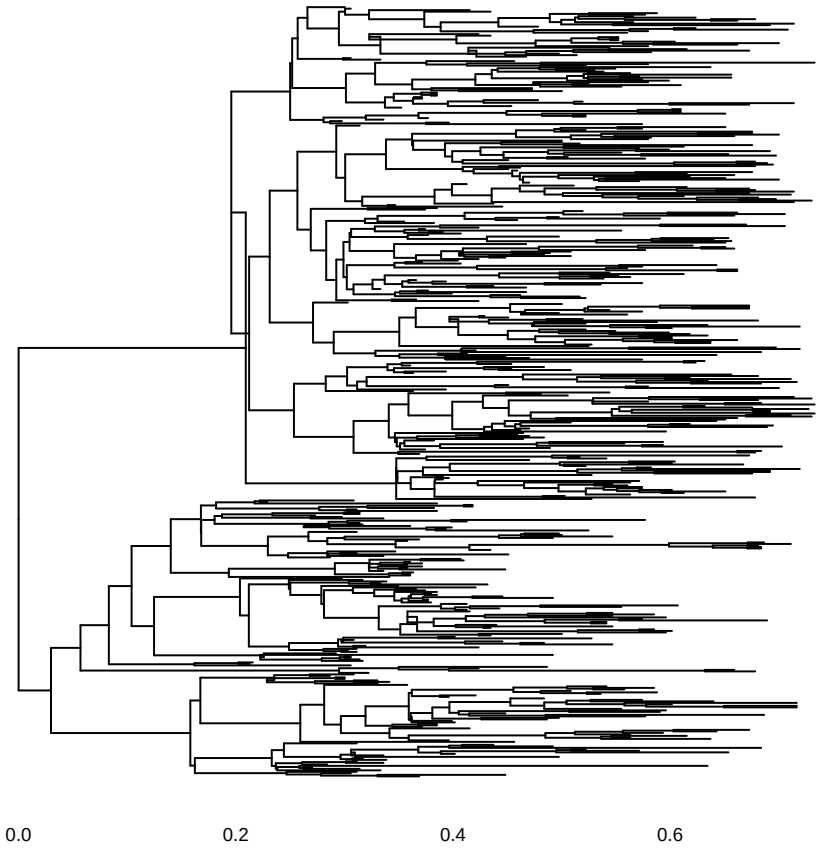
However, we are loosing the temporal information about the samples:

```
plot(x, colorBy="location", orderBy="location")
```



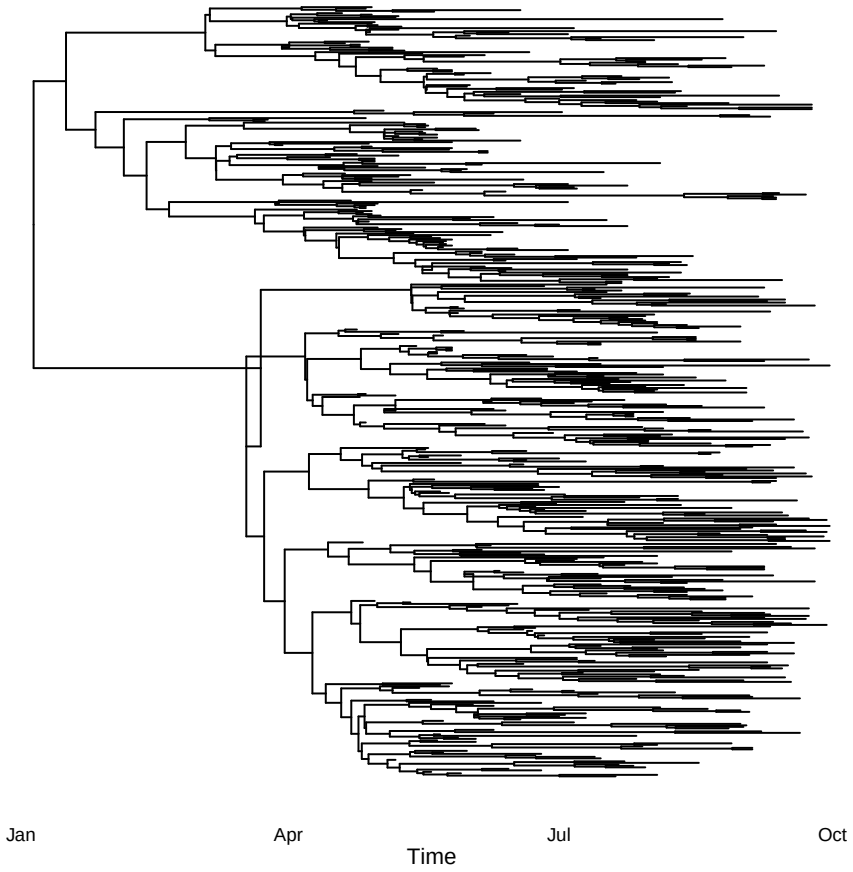
The basic plot of `plotggphy` gives a tree quite similar to *ape*'s:

```
plotggphy(x)
```



However, `plotggphy` is also more flexible and powerful. In particular, the argument `build.tip.attribute` allows to derive attributes for the tips based on information on samples and individuals. Here, for instance, we can use it to retrieve dates for each tip:

```
p <- plotggphy(x, ladderize = TRUE, build.tip.attribute = TRUE,
               branch.unit = "year", tip.dates = "date")
```



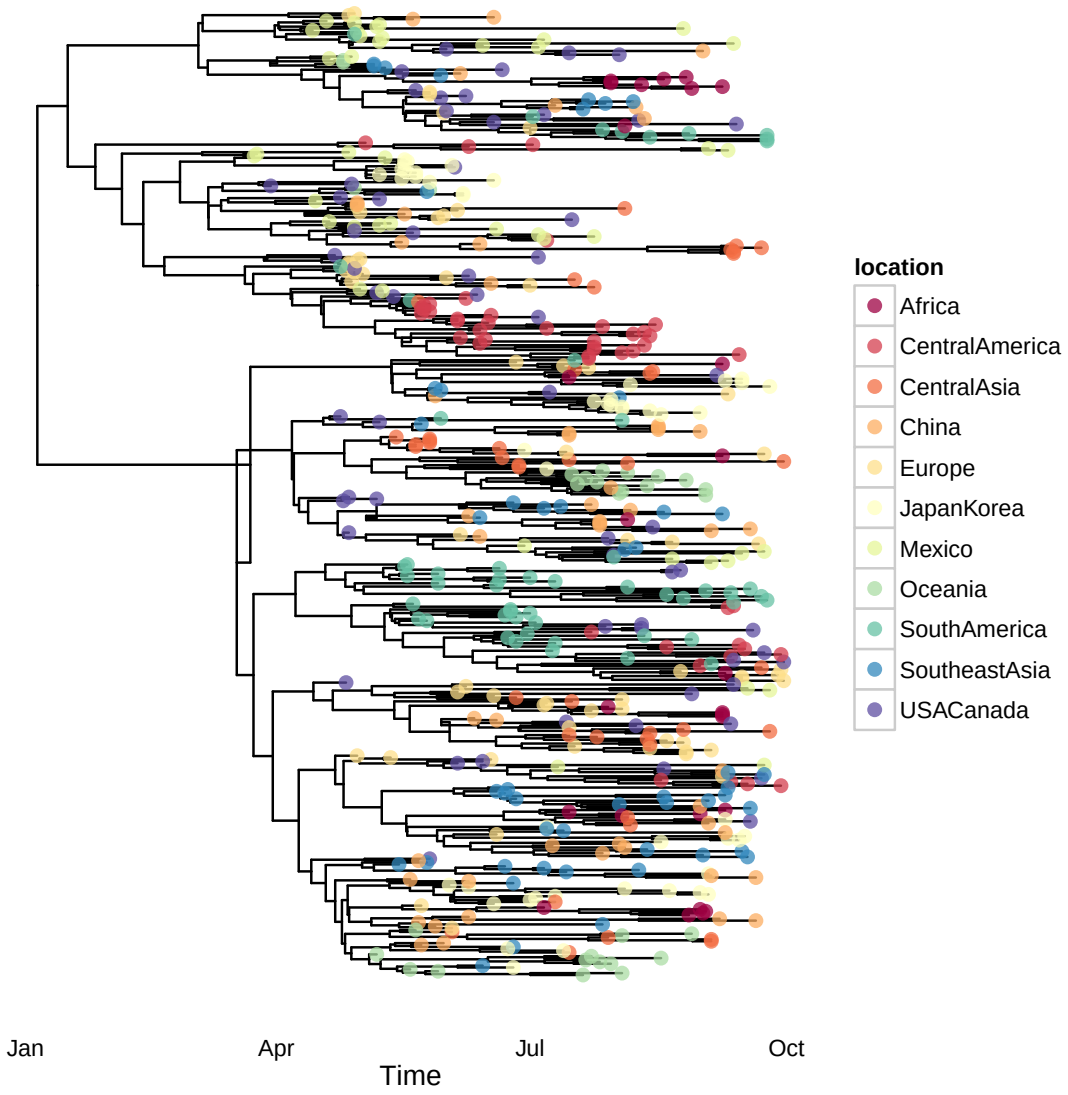
Note that `p` is a graphical object, which can be re-used later to generate and modify the plot. Importantly, other attributes can also be used and represented by colors on the tips. For instance, `x` contains information about the location of different individuals:

```
head(x@individuals)
```

```
##       location
## 1  CentralAsia
## 2  CentralAsia
## 3    USACanada
## 4       Europe
## 5 SouthAmerica
## 6 SouthAmerica
```

Which can be exploited by:

```
p <- plotggphy(x, ladderize = TRUE, build.tip.attribute = TRUE,
               branch.unit = "year", tip.dates = "date", tip.colour = "location",
               tip.size = 3, tip.alpha = 0.75)
```


location
Africa
CentralAmerica
CentralAsia
China
Europe
JapanKorea
Mexico
Oceania
SouthAmerica
SoutheastAsia
USACanada
Jan
Apr
Jul
Oct
Time