Handling disease outbreak data using *OutbreakTools* 0.1-16

Thibaut Jombart   Xavier Didelot   Rolf Ypma   Lulla Opatowski
Anne Cori   Marc A. Suchard

October 5, 2017

Abstract

This vignette introduces the main functionalities of *OutbreakTools*, a package implementing core tools for the analysis of outbreak data. Disease outbreak data can be diverse and complex, and the purpose of *OutbreakTools* 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 *OutbreakTools*, 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 *OutbreakTools*, 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 *OutbreakTools* 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(OutbreakTools)

## Loading required package:  ggplot2
## Loading required package:  network
## network:  Classes for Relational Data
## Version 1.13.0 created on 2015-08-31.
## copyright (c) 2005, Carter T. Butts, University of California-Irvine
##                     Mark S. Handcock, University of California -- Los Angeles
##                     David R. Hunter, Penn State University
##                     Martina Morris, University of Washington
##                     Skye Bender-deMoll, University of Washington
##  For citation information, type citation("network").
##  Type help("network-package") to get started.
##  OutbreakTools 0.1-16 has been loaded
```

```
getClassDef("obkData")

## Class "obkData" [package "OutbreakTools"]
##
## Slots:
##
## Name:         individuals            records                 dna
## Class:   data.frameOrNULL         listOrNULL obkSequencesOrNULL
##
## Name:            contacts            context               trees
## Class:  obkContactsOrNULL         listOrNULL   multiPhyloOrNULL
```

One can also examine a structure using an empty object:

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

```
##
## === obkData object ===
## == Empty slots ==
##  @individuals, @records, @dna, @contacts, @context, @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.
- `@records`: a named list of `data.frame`s storing any time-stamped data gathered at an individual level; 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`. Examples: swab data, fever, onset of symptoms, etc.
- `@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.
- `@context`: a list of `data.frame`s storing any time-stamped data at a non-individual level. Examples: climatic variables, school closures, vaccination campaign, etc.
- `@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] "OutbreakTools"

slotNames(ToyOutbreak)

## [1] "individuals" "records"     "dna"         "contacts"    "context"
## [6] "trees"

head(ToyOutbreak)

##
## === obkData x ===
## == @individuals==
##   infector DateInfected Sex Age      lat        lon
## 1       NA   2000-01-01   M  33 51.52152 -0.1805272
## 2        1   2000-01-02   F  42 51.51502 -0.1770907
## 3        2   2000-01-03   F  44 51.51885 -0.1614321
## 4        2   2000-01-03   M  49 51.51672 -0.1706063
```

```
##
## == @records==
##   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==
## = @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
##
##
## = @meta =
## [ meta information on the sequences ]
##   individualID       date locus sample
```

```
## 1            1 2000-01-01 gene1      1
## 2            2 2000-01-02 gene1      2
## 3            3 2000-01-03 gene1      3
## 4            4 2000-01-03 gene1      4
##
## ...
##    individualID       date locus sample
## 833          415 2000-01-10 gene2    415
## 834          416 2000-01-10 gene2    416
## 835          417 2000-01-10 gene2    417
## 836          418 2000-01-10 gene2    418
##
## == @contacts==
##  Number of individuals = 20
##  Number of contacts = 19
##  Contacts = dynamic
## NetworkDynamic properties:
##   distinct change times: 5
##   maximal time range: 0 until  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
##
##  Edge attribute names:
##     active
##
## Date of origin: [1] "2000-01-01"
##
## == @trees==
## 1 phylogenetic trees
##
## == Empty slots ==
##  @context
```

```
summary(ToyOutbreak)
```

```
## Dataset of 418 individuals with...
## == @individuals ==
## individuals information
##   418 entries
##   recorded fields are:
```

```
##   <infector> class: numeric,  mean: 84.26139,  sd:67.48384, range: [1;245],  1 NAs
##   <DateInfected> class: Date,  mean: 2000-01-08, range: [2000-01-01;2000-01-10],  0 NAs
##   <Sex> class: character,  2 unique values,  frequency range: [192;226],  0 NAs
##   <Age> class: numeric,  mean: 35.09809,  sd:6.10833, range: [19;56],  0 NAs
##   <lat> class: numeric,  mean: 51.51644,  sd:0.00304656, range: [51.50711;51.52625],  0 NAs
##   <lon> class: numeric,  mean: -0.1711455,  sd:0.01051185, range: [-0.2013245;-0.140349],  0 NAs
##
## == @records ==
## records on:  Fever
## $Fever
##   418 entries,  418 individuals, from 2000-01-03 to 2000-01-17
##   recorded fields are:
##   <temperature> class: numeric,  mean: 39.48541,  sd:0.5310073, range: [38;40.9],  0 NAs
##
## == @dna ==
## 836 sequences across 2 loci, 418 individuals, from 2000-01-01 to 2000-01-10
## length of concatenated alignment: 1600 nucleotides
## Attached meta data:
##   836 entries,  418 individuals, from 2000-01-01 to 2000-01-10
##   recorded fields are:
##   <locus> class: character,  2 unique values,  frequency range: [418;418],  0 NAs
##   <sample> class: character,  418 unique values,  frequency range: [2;2],  0 NAs
##
## == @contacts ==
## 19 contacts between 20 individuals
##
## == @trees ==
## 1 phylogenetic trees with 418 tips
```

`ToyOutbreak` is an `obkData` object containing information on individuals (`@individuals`), samples/records made on individuals (`@records`), DNA sequences (`@dna`), a contact network (`@contacts`) and one or more phylogenetic trees (`@trees`). Accessing a given slot is as easy as:

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

```
##   infector DateInfected Sex Age      lat        lon
## 1       NA   2000-01-01   M  33 51.52152 -0.1805272
## 2        1   2000-01-02   F  42 51.51502 -0.1770907
## 3        2   2000-01-03   F  44 51.51885 -0.1614321
## 4        2   2000-01-03   M  49 51.51672 -0.1706063
## 5        2   2000-01-03   M  34 51.51797 -0.1685206
## 6        2   2000-01-03   M  31 51.51401 -0.1662320
```

```
head(ToyOutbreak@records$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
```

```r
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: different individuals may have been sequenced for different genes, at different points in time, etc. The class `obkSequences` stores such information. `obkSequences` objects contain two slots: `@dna` and `@meta`.

The slot `@dna` is 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. Gene names are the names of the list. The row names in each matrix contain unique identifiers for the sequences, typically accession numbers.

The slot `@meta` is a `data.frame` containing some meta-information about the sequences. It contains at least two columns for sampled individuals (`individualID`) and collection dates (`date`). The row names corrspond to sequence labels used in `@dna`, and respect the same ordering.

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

```r
class(ToyOutbreak@dna)

## [1] "obkSequences"
## attr(,"package")
## [1] ".GlobalEnv"

ToyOutbreak@dna

## = @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
##
##
## = @meta =
## [ meta information on the sequences ]
##   individualID       date locus sample
## 1            1 2000-01-01 gene1      1
## 2            2 2000-01-02 gene1      2
## 3            3 2000-01-03 gene1      3
## 4            4 2000-01-03 gene1      4
##
## ...
##     individualID       date locus sample
## 833          415 2000-01-10 gene2    415
## 834          416 2000-01-10 gene2    416
## 835          417 2000-01-10 gene2    417
## 836          418 2000-01-10 gene2    418
```

```
slotNames(ToyOutbreak@dna)
```

```
## [1] "dna"  "meta"
```

```
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
```

```r
class(ToyOutbreak@dna@dna$gene1)
```

```
## [1] "DNAbin"
```

```r
class(ToyOutbreak@dna@meta)
```

```
## [1] "data.frame"
```

```r
head(ToyOutbreak@dna@meta)
```

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

`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`:

```r
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"
```

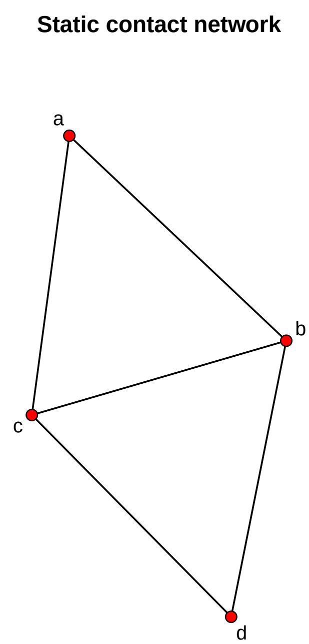
```r
oc.static
```

```
##  Number of individuals = 4
##  Number of contacts = 5
##  Contacts = dynamic
##  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
##
## No edge attributes
##
## Date of origin: NULL
```

`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,
```

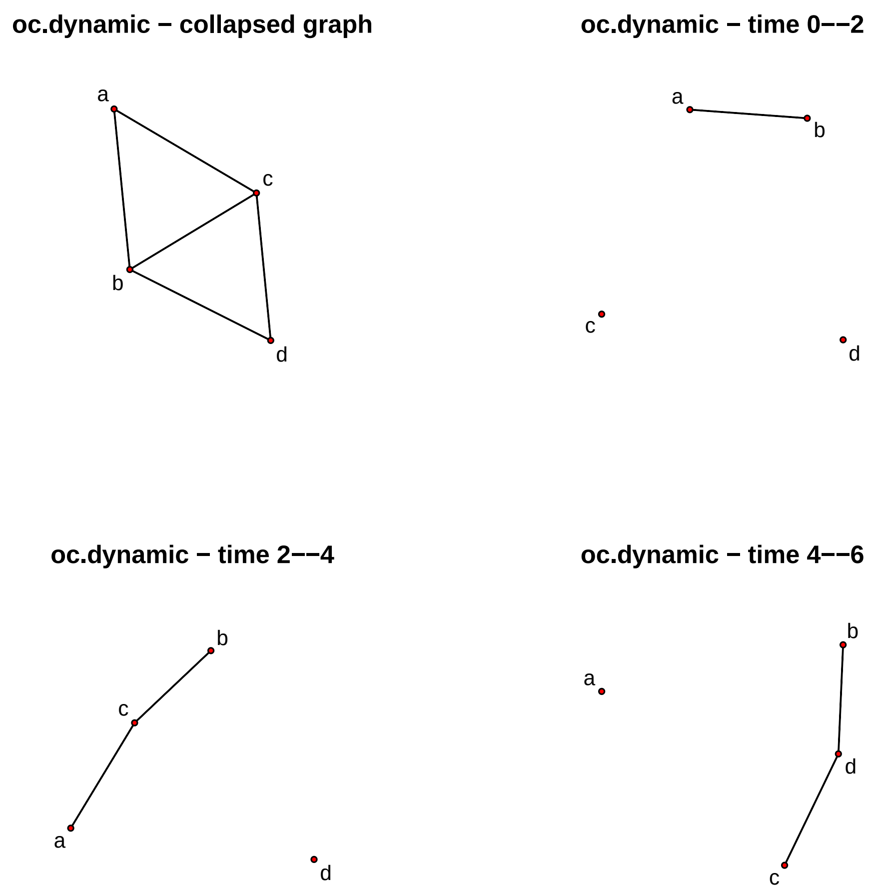
```
                  start=onset, end=terminus)
slotNames(oc.dynamic)

## [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 until  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
##
##  Edge attribute names:
##     active
##
## 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`:

```
as.data.frame(oc.dynamic)

##   onset terminus tail head onset.censored terminus.censored duration edge.id
## 1     1      1.2    a    b          FALSE             FALSE      0.2       1
## 2     2      4.0    b    c          FALSE             FALSE      2.0       2
## 3     3      3.5    a    c          FALSE             FALSE      0.5       3
## 4     4      4.1    c    d          FALSE             FALSE      0.1       4
## 5     5      6.0    d    b          FALSE             FALSE      1.0       5
```

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 *OutbreakTools*

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

1. read data into R

(a) read `data.frames` 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 separate files containing alignments (one file per gene), typically using `read.dna` from the ape package. 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, @records, @dna, @contacts, @context, @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; if missing, row names are used as identifiers.
- `records`: a list of data.frames, each of which has 2 mandatory fields, `individualID` and `date`. Dates can be specified as `Date` or `characters`, in which case they will be converted to dates. Most sensible formats will be detected automatically and processed. Unusual formats should be provided through the argument `date.format`. Each item of the list should be named according to the type of information recorded, e.g. 'swabs', 'temperature', or 'hospitalisation' (admission / discharge events).
- `dna`: a list matrices of DNA sequences in `DNAbin` or `character` format, each component of the list being a different gene. A matrix can be provided if there is a single gene.
- `dna.date`: a vector of collection dates for the DNA sequences; see `obkSequences` manpage for more information.
- `dna.individualID`: a vector of individual from which DNA sequences where obtained; see `obkSequences` manpage for more information.
- `dna.date.format`: a character string indicating the format of the date in `dna.date` if ambiguous; see `obkSequences` manpage for more information.
- `dna.sep`: the character string used to separate fields (e.g. sequenceID/individualID/date) in sequences labels; see `obkSequences` manpage for more information.
- `contacts`: a matrix of characters indicating contacts 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).

- `context`: a list of data.frames, each of which has 1 mandatory field: `date`. Each item of the list should be named according to the type of information recorded, e.g. 'intervention', 'vaccination', 'climat' (temperature, humidity, etc.), or schools (opening/closure).
- `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 with the class `multiPhylo` (from the ape package)
- `date.format`: a character string indicating the date format (see `as.Date`); if `NULL`, date format is detected automatically, which is usually a sensible option.

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"    "contacts"       "contacts.start" "contacts.end"
## [5] "dna"            "trees"          "dna.info"       "records"
```

Here is an overview of the inputs, including data on individuals:

```
head(ToyOutbreakRaw$individuals)

##   infector DateInfected Sex Age      lat        lon
## 1       NA   2000-01-01   M  33 51.52152 -0.1805272
## 2        1   2000-01-02   F  42 51.51502 -0.1770907
## 3        2   2000-01-03   F  44 51.51885 -0.1614321
## 4        2   2000-01-03   M  49 51.51672 -0.1706063
## 5        2   2000-01-03   M  34 51.51797 -0.1685206
## 6        2   2000-01-03   M  31 51.51401 -0.1662320
```

various time-stamped records:

```
lapply(ToyOutbreakRaw$records, head)

## $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
```

contact information:

```r
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"
```

DNA sequences:

```r
ToyOutbreakRaw$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
##
## $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
```

and phylogenetic trees:

```
ToyOutbreakRaw$trees

## 1 phylogenetic trees
```

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

```
attach(ToyOutbreakRaw)

## The following object is masked from package:datasets:
##
##     trees

x <- new ("obkData", individuals=individuals, records=records,
          contacts=contacts, contacts.start=contacts.start,
          contacts.end=contacts.end, dna=dna,
          dna.individualID=dna.info$individualID,
          dna.date=dna.info$date, sample=dna.info$sample, trees=trees)

detach(ToyOutbreakRaw)

head(x)

##
## === obkData x ===
## == @individuals==
##   infector DateInfected Sex Age      lat        lon
## 1       NA   2000-01-01   M  33 51.52152 -0.1805272
## 2        1   2000-01-02   F  42 51.51502 -0.1770907
## 3        2   2000-01-03   F  44 51.51885 -0.1614321
## 4        2   2000-01-03   M  49 51.51672 -0.1706063
##
## == @records==
##   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==
## = @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
##
##
## = @meta =
## [ meta information on the sequences ]
##   individualID       date locus sample
## 1            1 2000-01-01 gene1      1
## 2            2 2000-01-02 gene1      2
## 3            3 2000-01-03 gene1      3
## 4            4 2000-01-03 gene1      4
##
## ...
##     individualID       date locus sample
## 833          415 2000-01-10 gene2    415
## 834          416 2000-01-10 gene2    416
## 835          417 2000-01-10 gene2    417
## 836          418 2000-01-10 gene2    418
##
## == @contacts==
##  Number of individuals = 20
```

```
##  Number of contacts = 19
##  Contacts = dynamic
## NetworkDynamic properties:
##   distinct change times: 5
##   maximal time range: 0 until  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
##
##  Edge attribute names:
##     active
##
## Date of origin: [1] "2000-01-01"
##
## == @trees==
## 1 phylogenetic trees
##
## == Empty slots ==
##  @context
```

```
summary(x)
```

```
## Dataset of 418 individuals with...
## == @individuals ==
## individuals information
##   418 entries
##   recorded fields are:
##   <infector> class: numeric,  mean: 84.26139,  sd:67.48384, range: [1;245],  1 NAs
##   <DateInfected> class: character,  10 unique values,  frequency range: [1;173],  0 NAs
##   <Sex> class: character,  2 unique values,  frequency range: [192;226],  0 NAs
##   <Age> class: numeric,  mean: 35.09809,  sd:6.10833, range: [19;56],  0 NAs
##   <lat> class: numeric,  mean: 51.51644,  sd:0.00304656, range: [51.50711;51.52625],  0 NAs
##   <lon> class: numeric,  mean: -0.1711455,  sd:0.01051185, range: [-0.2013245;-0.140349],  0 NAs
##
## == @records ==
## records on:  Fever
## $Fever
##   418 entries,  418 individuals, from 2000-01-03 to 2000-01-17
##   recorded fields are:
##   <temperature> class: numeric,  mean: 39.48541,  sd:0.5310073, range: [38;40.9],  0 NAs
##
```

```
## == @dna ==
## 836 sequences across 2 loci, 418 individuals, from 2000-01-01 to 2000-01-10
## length of concatenated alignment: 1600 nucleotides
## Attached meta data:
##   836 entries,  418 individuals, from 2000-01-01 to 2000-01-10
##   recorded fields are:
##   <locus> class: character,  2 unique values,  frequency range: [418;418],  0 NAs
##   <sample> class: character,  418 unique values,  frequency range: [2;2],  0 NAs
##
## == @contacts ==
## 19 contacts between 20 individuals
##
## == @trees ==
## 1 phylogenetic trees 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 records, contacts, DNA sequences and patient information
- every item is dated using actual dates (`Date` objects)
- every sample/record refers to an individual and a date
- every DNA sequence refers to an individual and a date
- 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

Having all these items connected allows to simplify data manipulation fairly drastically (see section below on data handling). For instance, it will be easy to isolate a subset of individuals from the data, which will impact not only patient information but also the phylogenies, DNA sequences, records and contacts. It will also be straightforward to add e.g. patient information onto phylogenies.

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 *OutbreakTools*. 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 of 5 individuals of the toy dataset `ToyOutbreak`:

```
data(ToyOutbreak)
set.seed(1)
toKeep <- sample(get.nindividuals(ToyOutbreak),5)
toKeep

## [1] 111 156 239 377  84

x <- subset(ToyOutbreak, individuals=toKeep)
summary(x)

## Dataset of 5 individuals with...
## == @individuals ==
## individuals information
##   5 entries
##   recorded fields are:
##   <infector> class: numeric,  mean: 86.4,  sd:39.22754, range: [33;133],  0 NAs
##   <DateInfected> class: Date,  mean: 2000-01-08, range: [2000-01-08;2000-01-10],  0 NAs
##   <Sex> class: character,  2 unique values,  frequency range: [2;3],  0 NAs
##   <Age> class: numeric,  mean: 33.4,  sd:5.813777, range: [24;38],  0 NAs
##   <lat> class: numeric,  mean: 51.51609,  sd:0.002712762, range: [51.51188;51.51939],  0 NAs
##   <lon> class: numeric,  mean: -0.1699346,  sd:0.01078318, range: [-0.1851876;-0.1575907],  0 NAs
##
## == @records ==
## records on:  Fever
## $Fever
##   5 entries,  5 individuals, from 2000-01-09 to 2000-01-15
```

```
##   recorded fields are:
##   <temperature> class: numeric,  mean: 39.16,  sd:0.4037326, range: [38.5;39.6],  0 NAs
##
## == @dna ==
## 10 sequences across 2 loci, 5 individuals, from 2000-01-08 to 2000-01-10
## length of concatenated alignment: 1600 nucleotides
## Attached meta data:
##   10 entries,  5 individuals, from 2000-01-08 to 2000-01-10
##   recorded fields are:
##   <locus> class: character,  2 unique values,  frequency range: [5;5],  0 NAs
##   <sample> class: character,  5 unique values,  frequency range: [2;2],  0 NAs
```

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

```
get.nindividuals(x)

## [1] 5

get.nindividuals(x, "records")

## [1] 5

get.nindividuals(x, "dna")

## [1] 5

get.nindividuals(x, "contacts")

## [1] 0
```

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

```
get.nlocus(x)

## [1] 2
```

```r
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`.

```r
get.nsequences(x)

## [1] 10

get.nsequences(x, "bylocus")

## gene1 gene2
##     5     5

get.sequences(x)

## gene11 gene12 gene13 gene14 gene15 gene21 gene22 gene23 gene24 gene25
##   "84"  "111"  "156"  "239"  "377"  "502"  "529"  "574"  "657"  "795"
```

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

```r
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`.

```r
get.dna(x)

## $gene1
## 5 DNA sequences in binary format stored in a matrix.
##
## All sequences of same length: 600
##
## Labels:
## 84
## 111
## 156
## 239
## 377
##
## Base composition:
```

```
##     a     c     g     t
## 0.237 0.248 0.251 0.264
##
## $gene2
## 5 DNA sequences in binary format stored in a matrix.
##
## All sequences of same length: 1000
##
## Labels:
## 502
## 529
## 574
## 657
## 795
##
## Base composition:
##     a     c     g     t
## 0.224 0.244 0.257 0.276
```

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
## 5 DNA sequences in binary format stored in a matrix.
##
## All sequences of same length: 1000
##
## Labels:
## 502
## 529
## 574
## 657
## 795
##
## Base composition:
##     a     c     g     t
## 0.224 0.244 0.257 0.276
```

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

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

## named list()
```

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

```
get.sequences(x)

## gene11 gene12 gene13 gene14 gene15 gene21 gene22 gene23 gene24 gene25
##   "84"  "111"  "156"  "239"  "377"  "502"  "529"  "574"  "657"  "795"
```

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

## [1] FALSE
```

- `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:

```r
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] "111" "156" "239" "377" "84"

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`, in the data.frames in `x@records`, or `x@context`, or in the `@dna@meta`. 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`), while `showSource` allows to put information in context (i.e., adding `individualID`, `date` and `source` where applicable.

  For instance, we can retrieve temperature measurements using:

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

## [1] 38.5 39.3 39.2 39.6 39.2

get.data(x,"temperature", showSource=TRUE)

##   temperature individualID       date source
## 1        38.5           84 2000-01-12  Fever
## 2        39.3          111 2000-01-09  Fever
```

```
## 3        39.2           156 2000-01-10  Fever
## 4        39.6           239 2000-01-12  Fever
## 5        39.2           377 2000-01-15  Fever
```

or the sex of the different individuals:

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

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

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

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

##   Sex Age infector
## 1   F  38       33
## 2   F  24      133
## 3   M  38       97
## 4   M  35      107
## 5   M  32       62
```

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

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

##   Sex Age infector individualID      source
## 1   F  38       33          111 individuals
## 2   F  24      133          156 individuals
## 3   M  38       97          239 individuals
## 4   M  35      107          377 individuals
## 5   M  32       62           84 individuals
```

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

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

##  [1] "2000-01-12" "2000-01-09" "2000-01-10" "2000-01-12" "2000-01-15"
##  [6] "2000-01-08" "2000-01-08" "2000-01-09" "2000-01-09" "2000-01-10"
## [11] "2000-01-08" "2000-01-08" "2000-01-09" "2000-01-09" "2000-01-10"
```

actually corresponds to:

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

##          date individualID       date source
## 1  2000-01-12           84 2000-01-12  Fever
## 2  2000-01-09          111 2000-01-09  Fever
## 3  2000-01-10          156 2000-01-10  Fever
## 4  2000-01-12          239 2000-01-12  Fever
## 5  2000-01-15          377 2000-01-15  Fever
```

```
## 6   2000-01-08           84 2000-01-08    dna
## 7   2000-01-08          111 2000-01-08    dna
## 8   2000-01-09          156 2000-01-09    dna
## 9   2000-01-09          239 2000-01-09    dna
## 10  2000-01-10          377 2000-01-10    dna
## 11  2000-01-08           84 2000-01-08    dna
## 12  2000-01-08          111 2000-01-08    dna
## 13  2000-01-09          156 2000-01-09    dna
## 14  2000-01-09          239 2000-01-09    dna
## 15  2000-01-10          377 2000-01-10    dna
```

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

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

##           date individualID       date source
## 84  2000-01-12           84 2000-01-12  Fever
## 111 2000-01-09          111 2000-01-09  Fever
## 156 2000-01-10          156 2000-01-10  Fever
## 239 2000-01-12          239 2000-01-12  Fever
## 377 2000-01-15          377 2000-01-15  Fever
```

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

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

## Warning in .local(x, ...):  data 'sugarman' was not found in the object

## 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`
- `get.individuals`
- `get.nindividuals`
- `get.dates`
- `get.ndates`

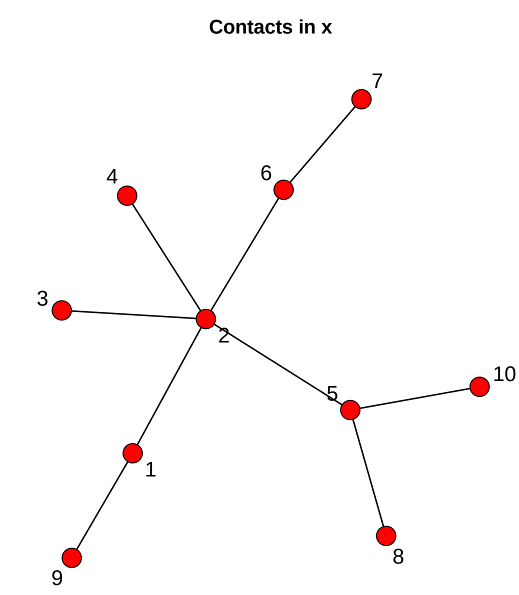
2.1.3 Accessors for `obkContacts` objects

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`
- `get.dates`
- `get.ndates`

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

Lastly, for dynamic graphs, the function `as.data.frame` returns all the relevant information:

```
as.data.frame(x@contacts)
```

```
##        onset   terminus tail head onset.censored terminus.censored duration
## 1 2000-01-01 2000-01-02    2    1          FALSE             FALSE        1
## 2 2000-01-02 2000-01-03    3    2          FALSE             FALSE        1
## 3 2000-01-02 2000-01-03    4    2          FALSE             FALSE        1
## 4 2000-01-02 2000-01-03    5    2          FALSE             FALSE        1
## 5 2000-01-02 2000-01-03    6    2          FALSE             FALSE        1
## 6 2000-01-03 2000-01-04    7    6          FALSE             FALSE        1
## 7 2000-01-03 2000-01-04    8    5          FALSE             FALSE        1
## 8 2000-01-03 2000-01-04    9    1          FALSE             FALSE        1
## 9 2000-01-03 2000-01-04   10    5          FALSE             FALSE        1
##   edge.id
## 1       1
## 2       2
## 3       3
## 4       4
## 5       5
## 6       6
## 7       7
## 8       8
## 9       9
```

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, locus=NULL, sequences=NULL,
       date.from=NULL, date.to=NULL, date.format=NULL, ...)
```

See `?subset.obkData` for the details of these arguments. The function works in a fairly intuitive way. The arguments `individuals`, `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, these two formulations are equivalent:

```
data(ToyOutbreak)
x1 <- subset(ToyOutbreak, individuals=1:10)
x2 <- subset(ToyOutbreak, get.individuals(ToyOutbreak)[1:10])
identical(x1,x2)

## [1] TRUE
```

Another, non-exclusive way of subsetting the data is using dates. 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)
attach(FluH1N1pdm2009)

## The following object is masked from package:datasets:
##
##     trees

x <- new("obkData", individuals = individuals, dna = FluH1N1pdm2009$dna,
      dna.individualID = samples$individualID, dna.date = samples$date,
      trees = FluH1N1pdm2009$trees)

detach(FluH1N1pdm2009)

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

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

We can retain data collected during the first month using:

```
min.date <- min(get.dates(x))
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...
## == @individuals ==
## individuals information
##   514 entries
##
## == @dna ==
## 514 sequences across 1 loci, 514 individuals, from 2009-03-24 to 2009-09-30
## length of concatenated alignment: 1664 nucleotides
## Attached meta data:
##   514 entries,  514 individuals, from 2009-03-24 to 2009-09-30
##   recorded fields are:
##   <locus> class: character,  1 unique values,  frequency range: [514;514],  0 NAs
```

```
summary(x1)
```

```
## Dataset of 514 individuals with...
## == @individuals ==
## individuals information
##   514 entries
##
## == @dna ==
## 12 sequences across 1 loci, 12 individuals, from 2009-03-24 to 2009-04-24
## length of concatenated alignment: 1664 nucleotides
## Attached meta data:
##   12 entries,  12 individuals, from 2009-03-24 to 2009-04-24
##   recorded fields are:
##   <locus> class: character,  1 unique values,  frequency range: [12;12],  0 NAs
```

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

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:

```
temp <- get.data(x, "location", showSource=TRUE)
head(temp)
```

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

```
toKeep <- temp$individualID[temp$location=="Europe"]
x.summerEur <- subset(x, date.from="01/06/2009", date.to="31/08/2009",
                      indiv=toKeep)
summary(x.summerEur)
```

```
## Dataset of 60 individuals with...
## == @individuals ==
```

```
## individuals information
##   60 entries
##
## == @dna ==
## 30 sequences across 1 loci, 30 individuals, from 2009-06-01 to 2009-08-26
## length of concatenated alignment: 1664 nucleotides
## Attached meta data:
##   30 entries,  30 individuals, from 2009-06-01 to 2009-08-26
##   recorded fields are:
##   <locus> class: character,  1 unique values,  frequency range: [30;30],  0 NAs
```

```
head(x.summerEur)
```

```
##
## === obkData x ===
## == @individuals==
##    location
## 4    Europe
## 68   Europe
## 69   Europe
## 70   Europe
##
## == @dna==
## = @dna =
## [ 30 DNA sequences in 1 locus ]
## $locus.1
## 30 DNA sequences in binary format stored in a matrix.
##
## All sequences of same length: 1664
##
## Labels:
## A/Finland/577/2009_Europe_2009-06-19
## A/Managua/4702.04/2009_CentralAmerica_2009-08-19
## A/Managua/5401.01/2009_CentralAmerica_2009-09-16
## A/Shenzhen/25_SZCDC/2009_China_2009-08-31
## A/Texas/45121606/2009_USACanada_2009-09-12
## A/Utah/07/2009_USACanada_2009-06-14
## ...
##
## Base composition:
##     a     c     g     t
## 0.354 0.186 0.224 0.236
##
##
## = @meta =
## [ meta information on the sequences ]
##                                                    individualID       date
## A/Finland/577/2009_Europe_2009-06-19                         73 2009-08-05
## A/Managua/4702.04/2009_CentralAmerica_2009-08-19             74 2009-08-19
## A/Managua/5401.01/2009_CentralAmerica_2009-09-16             75 2009-08-21
## A/Shenzhen/25_SZCDC/2009_China_2009-08-31                    76 2009-08-25
##                                                      locus
```

```
## A/Finland/577/2009_Europe_2009-06-19             locus.1
## A/Managua/4702.04/2009_CentralAmerica_2009-08-19 locus.1
## A/Managua/5401.01/2009_CentralAmerica_2009-09-16 locus.1
## A/Shenzhen/25_SZCDC/2009_China_2009-08-31        locus.1
##
## ...
##                                         individualID       date   locus
## A/Auckland/3/2009_Oceania_2009-04-25                 371 2009-08-26 locus.1
## A/Guangdong/1/2009_China_2009-05-17                  374 2009-06-03 locus.1
## A/Beijing/502/2009_China_2009-05-20                  375 2009-06-08 locus.1
## A/Reunion/0215_4_M1E/2009_Europe_2009-09-04          409 2009-06-17 locus.1
##
## == @trees==
## 1 phylogenetic trees
##
## == Empty slots ==
##  @records, @contacts, @context
```

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 *OutbreakTools*, 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):

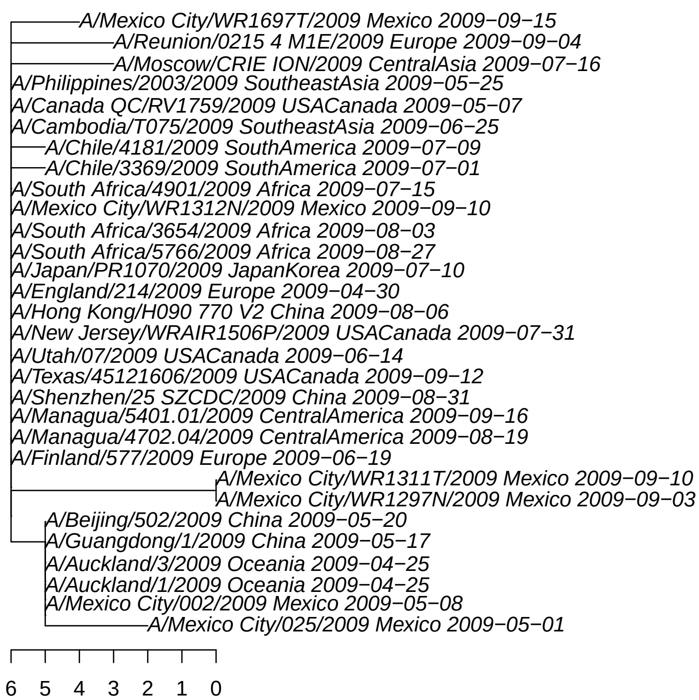
```
x.summerEur@trees <- NULL
get.nsequences(x.summerEur)

## [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 60 individuals with...
## == @individuals ==
## individuals information
##   60 entries
##
## == @dna ==
## 30 sequences across 1 loci, 30 individuals, from 2009-06-01 to 2009-08-26
## length of concatenated alignment: 1664 nucleotides
## Attached meta data:
##   30 entries,  30 individuals, from 2009-06-01 to 2009-08-26
##   recorded fields are:
##   <locus> class: character,  1 unique values,  frequency range: [30;30],  0 NAs
```

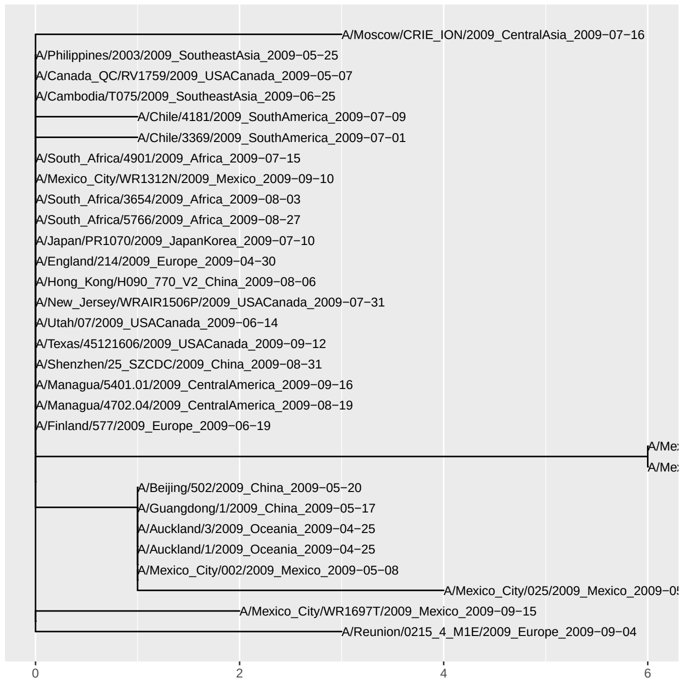
x2 now contains a phylogenetic tree derived from the sequences in `x.summerEur`. This one can be plotted simply, using:

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



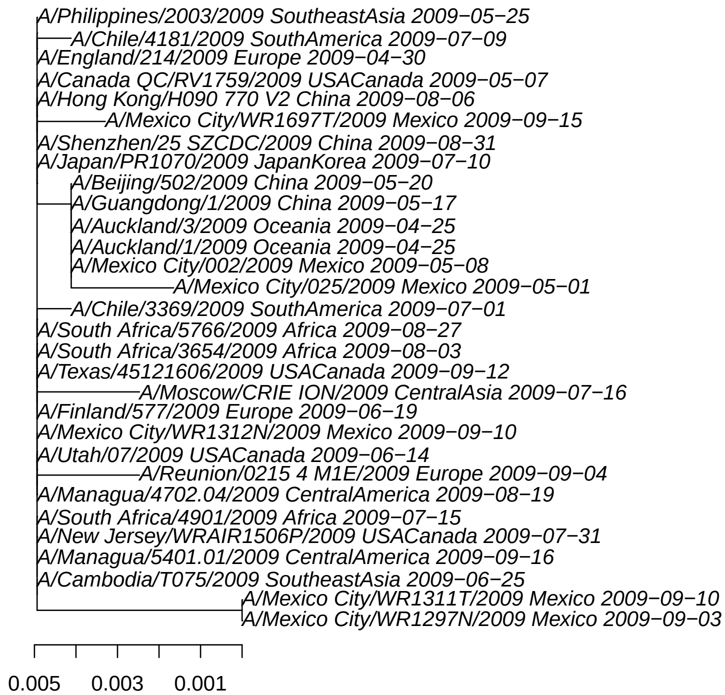
or alternatively:

```
plot(x2, "phylo")
```



Note that we could ask for a different model of evolution, for instance Kimura's 2 parameters distance:

```
x3 <- make.phylo(x.summerEur, locus=1, ask=FALSE, model="K80")
plot(get.trees(x3)[[1]])
axisPhylo()
```



Finally, note that *OutbreakTools* 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

OutbreakTools 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 a list containing the SIR dynamics (`x$dynamics`), an `obkData` object (`x$x`) and an optional `ggplot` graphic of the SIR dynamics (`x$plot`).

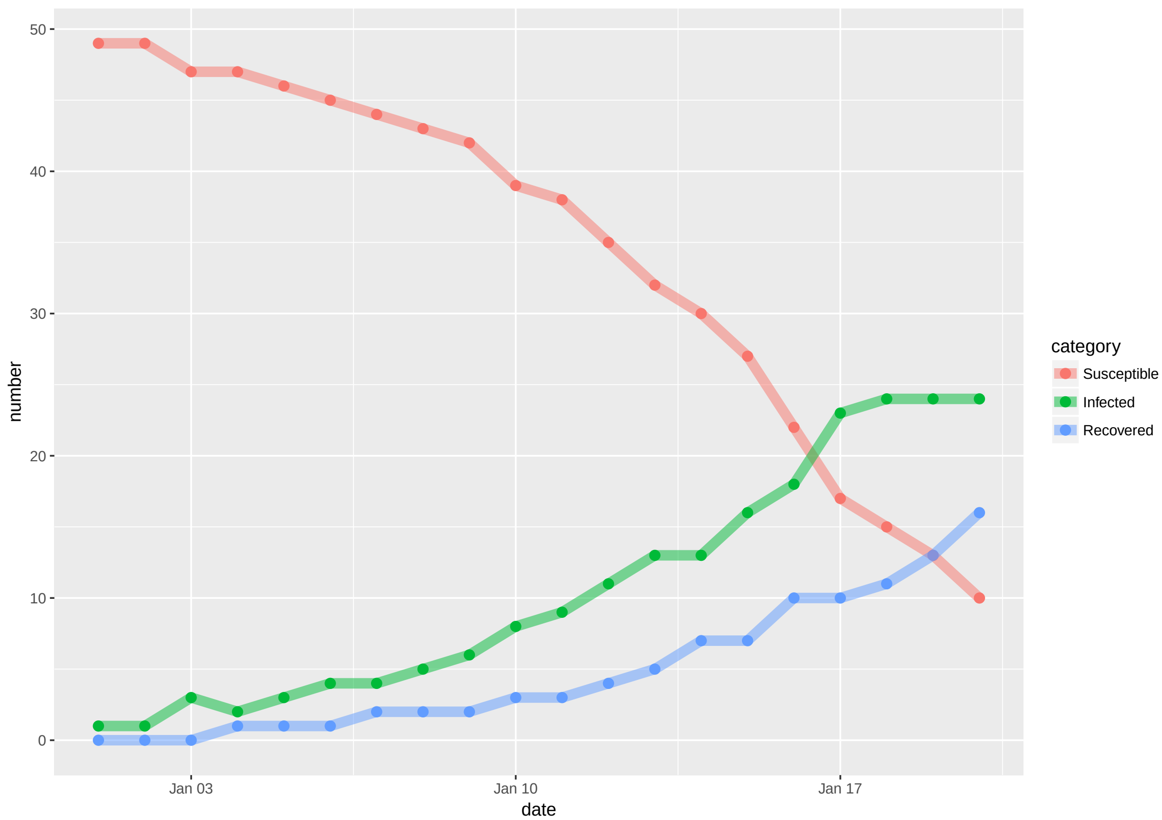
The arguments are as follows:

- `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`.
- `plot`: logical indicating whether to create a plot of the SIR trajectory over time. Default is `plot=TRUE`. Plot will be a `ggplot` object stored as the `$plot` slot of the returned list.
- `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, D=20, beta=0.01,plot=TRUE,makePhylo=TRUE)
```




```
summary(x)

##          Length Class      Mode
## x        1      obkData    S4
## dynamics 4      data.frame list
## plot     9      gg         list

x$dynamics

##    Susceptible Infected Recovered       date
## 1           49        1         0 2000-01-01
```

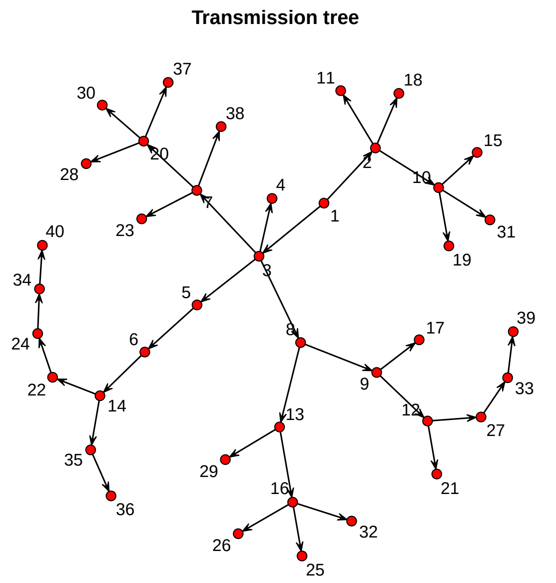
```
## 2           49       1        0 2000-01-02
## 3           47       3        0 2000-01-03
## 4           47       2        1 2000-01-04
## 5           46       3        1 2000-01-05
## 6           45       4        1 2000-01-06
## 7           44       4        2 2000-01-07
## 8           43       5        2 2000-01-08
## 9           42       6        2 2000-01-09
## 10          39       8        3 2000-01-10
## 11          38       9        3 2000-01-11
## 12          35      11        4 2000-01-12
## 13          32      13        5 2000-01-13
## 14          30      13        7 2000-01-14
## 15          27      16        7 2000-01-15
## 16          22      18       10 2000-01-16
## 17          17      23       10 2000-01-17
## 18          15      24       11 2000-01-18
## 19          13      24       13 2000-01-19
## 20          10      24       16 2000-01-20
```

```
summary(x$x)
```

```
## Dataset of 40 individuals with...
## == @individuals ==
## individuals information
##   40 entries
##
## == @dna ==
## 40 sequences across 1 loci, 40 individuals, from 2000-01-01 to 2000-01-20
## length of concatenated alignment: 1000 nucleotides
## Attached meta data:
##   40 entries,  40 individuals, from 2000-01-01 to 2000-01-20
##   recorded fields are:
##   <locus> class: character,  1 unique values,  frequency range: [40;40],  0 NAs
##
## == @contacts ==
## 39 contacts between 40 individuals
##
## == @trees ==
## 1 phylogenetic trees with 40 tips
```

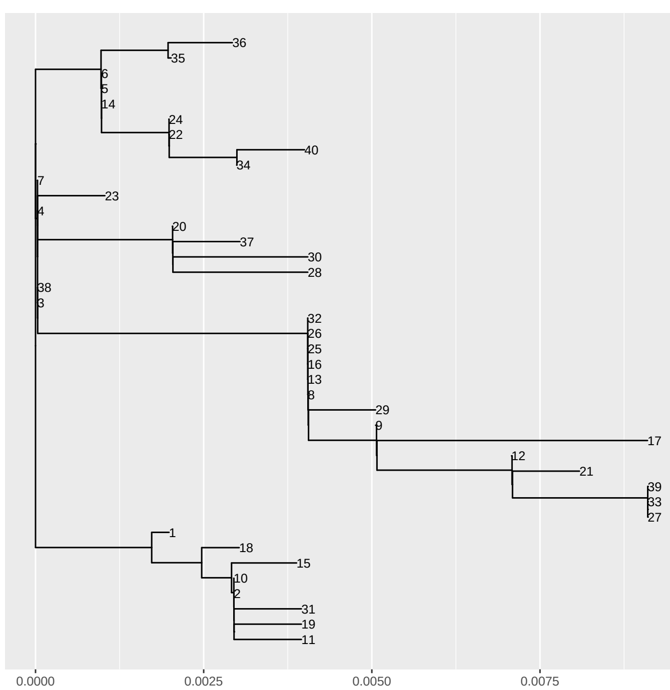
We can see that 40 individuals got infected over the time period of `D=20` days during which the outbreak was simulated. The actual transmission tree is stored as contact information:

```
plot(x$x, "contacts", main="Transmission tree")
```



The object also possesses a Neighbor-Joining tree based on the simulated sequence data:

```
plot(x$x, "phylo")
```



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

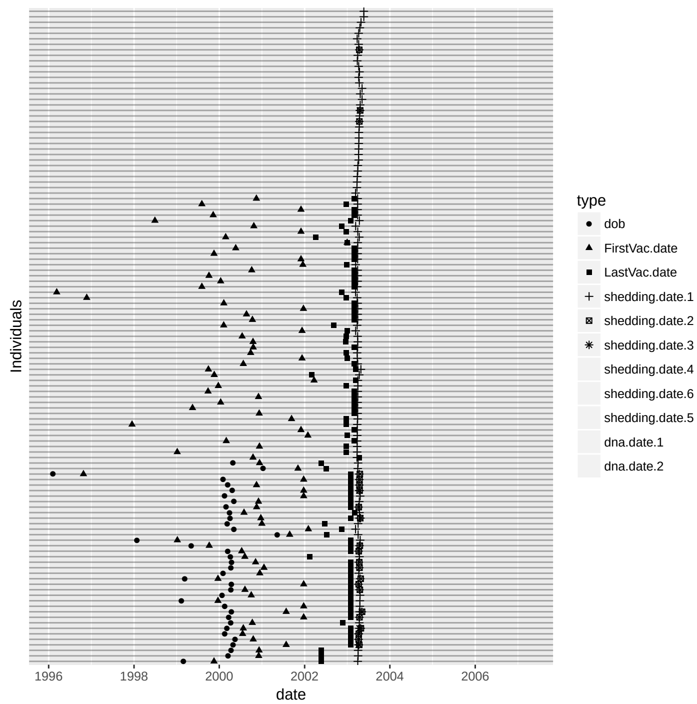
This plotting option relies on the function `plotIndividualTimeline`; see `?plotIndividualTimeline` for more information. Let's plot the outbreak of equine influenza provided in `HorseFlu`:

```
data(HorseFlu)
summary(HorseFlu)

## Dataset of 121 individuals with...
## == @individuals ==
## individuals information
##   120 entries
##   recorded fields are:
##   <yardID> class: factor,  25 unique values,  frequency range: [1;33],  1 NAs
##   <dob> class: Date,  mean: 2000-01-01, range: [1996-02-07;2001-05-12],  83 NAs
##   <sex> class: factor,  3 unique values,  frequency range: [25;49],  46 NAs
##   <lat> class: numeric,  mean: 52.19754,  sd:0.2180945, range: [51.16454;52.26274],  2 NAs
##   <lon> class: numeric,  mean: 0.3204825,  sd:0.3313301, range: [-1.429922;0.436602],  2 NAs
##
## == @records ==
## records on:  FirstVac, LastVac, shedding
## $FirstVac
##   85 entries,  85 individuals, from 1996-03-10 to 2002-12-31
## $LastVac
##   85 entries,  85 individuals, from 2002-02-14 to 2003-04-12
## $shedding
##   153 entries,  119 individuals, from 2003-03-13 to 2003-05-23
##   recorded fields are:
##   <sampleID> class: character,  153 unique values,  frequency range: [1;1],  0 NAs
##   <shedding> class: numeric,  mean: 6405.373,  sd:27377.58, range: [1;295000],  0 NAs
##
## == @dna ==
## 2361 sequences across 1 loci, 51 individuals, from 2003-03-13 to 2007-04-09
## length of concatenated alignment: 903 nucleotides
## Attached meta data:
##   2361 entries,  51 individuals, from 2003-03-13 to 2007-04-09
```

```
##   recorded fields are:
##   <locus> class: character,  1 unique values,  frequency range: [2361;2361],  0 NAs
##   <sampleID> class: integer,  mean: 305113.7,  sd:30439.25, range: [904;311920],  35 NAs
```

The default plot is a timeline showing all time-stamped data

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

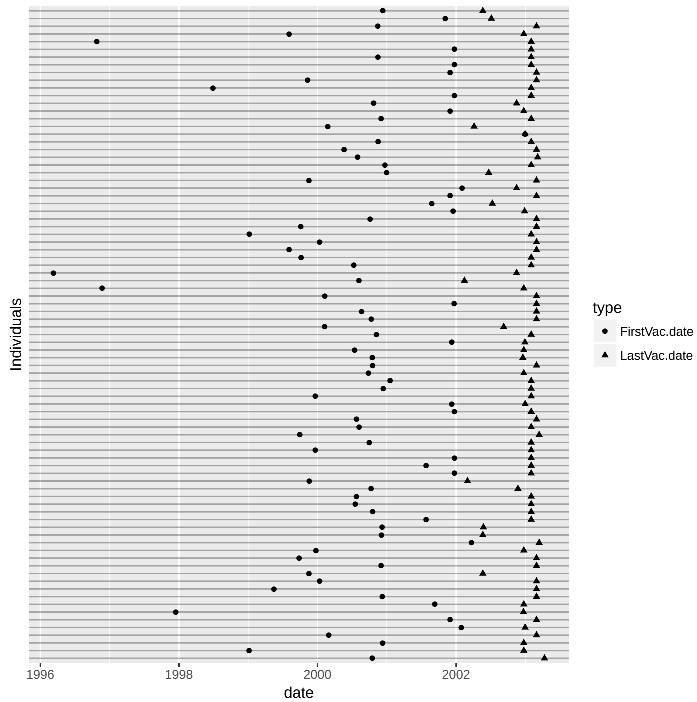


A problem appears here: in this dataset, there is simply too much information to display on a single graphic. This can be improved by passing further arguments to `plotIndividualTimeline`:

```
args(plotIndividualTimeline)

## function (x, what = "", selection = NULL, ordering = NULL, orderBy = NULL,
##     colorBy = NULL, periods = NULL, plotNames = length(selection) <
##         50, ...)
## NULL
```

For instance, we can choose to visualize only vaccination dates:

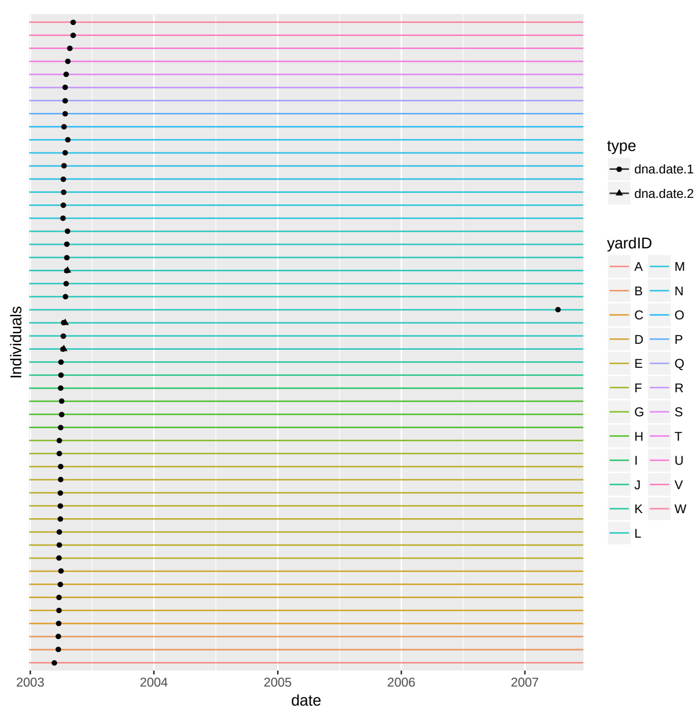
```
plot(HorseFlu,'timeline', what="Vac")
```



note that the argument `what` actually uses regular expressions to find matching fields in the data, so that here `'Vac'` allows us to keep `FirstVac` and `LastVac`.

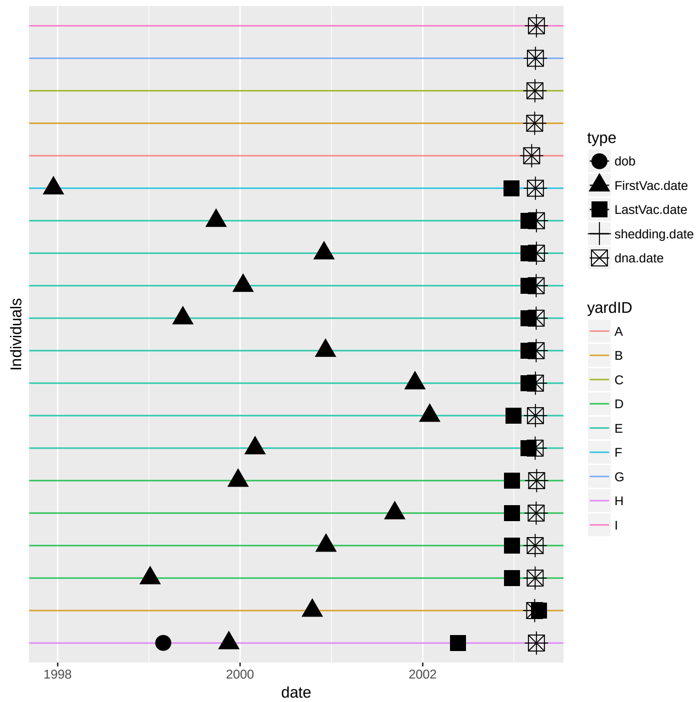
Individuals can also be ordered and colored according to individual meta information. For instance, to visualize collection dates of DNA and sort individuals per yards:

```
plotIndividualTimeline(HorseFlu, what="dna", colorBy="yardID", orderBy="yardID",plotNames=TRUE)
```



Note that only individuals for which requested information is present (here, DNA sequences) are plotted. It is also possible to specify a subset of individuals using `selection`:

```
plot(HorseFlu,selection=1:20, colorBy="yardID", orderBy="yardID", size=5)
```



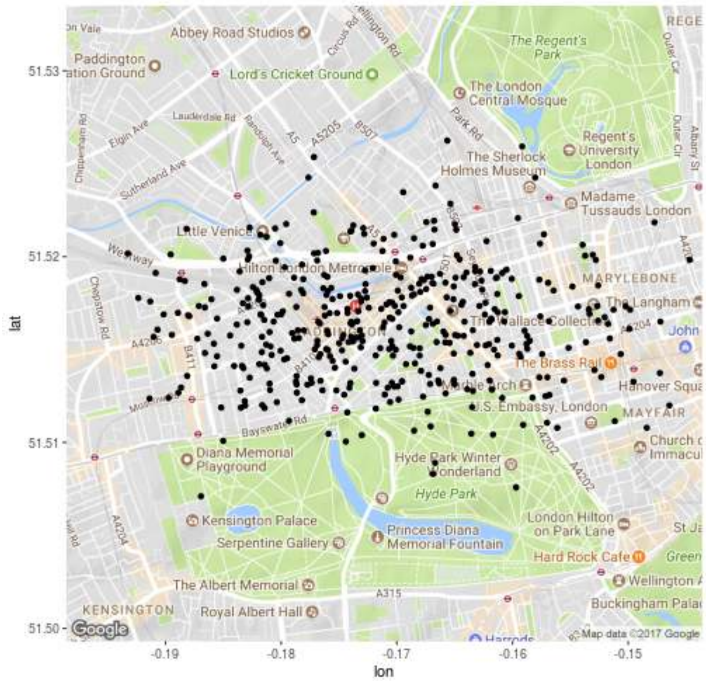
4.2 Visualizing samples on a map

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 51.52152 -0.1805272
## 2        1   2000-01-02   F  42 51.51502 -0.1770907
## 3        2   2000-01-03   F  44 51.51885 -0.1614321
## 4        2   2000-01-03   M  49 51.51672 -0.1706063
## 5        2   2000-01-03   M  34 51.51797 -0.1685206
## 6        2   2000-01-03   M  31 51.51401 -0.1662320
```

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):

```r
plot(ToyOutbreak,'geo', location=c('lon','lat'), 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`:

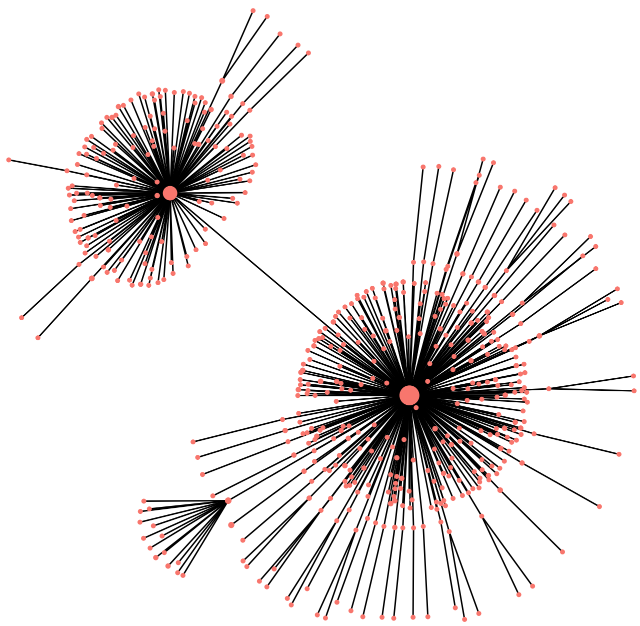
```r
plot(ToyOutbreak,'geo', location=c('lon','lat'), zoom=15,
     colorBy='Sex', center='11')
```

4.3 Building minimum spanning trees from genetic sequences

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

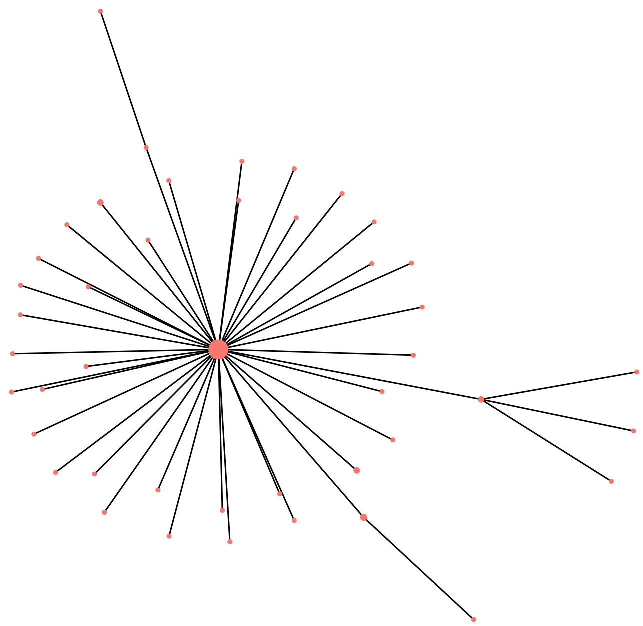
It can be useful to plot a minimal spanning tree of the sequences, to quickly visualize the genetic diversity and the relation between sequences. This can be achieved using `plotggMST`, or simply `plot` using `mst` for the second argument:

```
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)
attach(FluH1N1pdm2009)

## The following object is masked from package:datasets:
##
##     trees

x <- new("obkData", individuals = individuals, dna = FluH1N1pdm2009$dna,
      dna.individualID = samples$individualID, dna.date = samples$date,
      trees = FluH1N1pdm2009$trees)

detach(FluH1N1pdm2009)

summary(x)

## Dataset of 514 individuals with...
## == @individuals ==
## individuals information
##   514 entries
##
## == @dna ==
## 514 sequences across 1 loci, 514 individuals, from 2009-03-24 to 2009-09-30
```

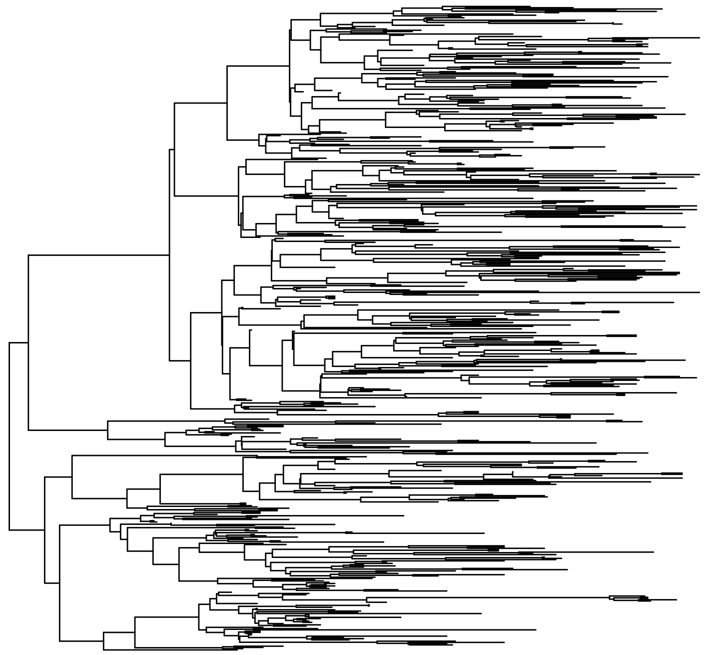
```
## length of concatenated alignment: 1664 nucleotides
## Attached meta data:
##   514 entries,  514 individuals, from 2009-03-24 to 2009-09-30
##   recorded fields are:
##   <locus> class: character,  1 unique values,  frequency range: [514;514],  0 NAs
```

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.
```

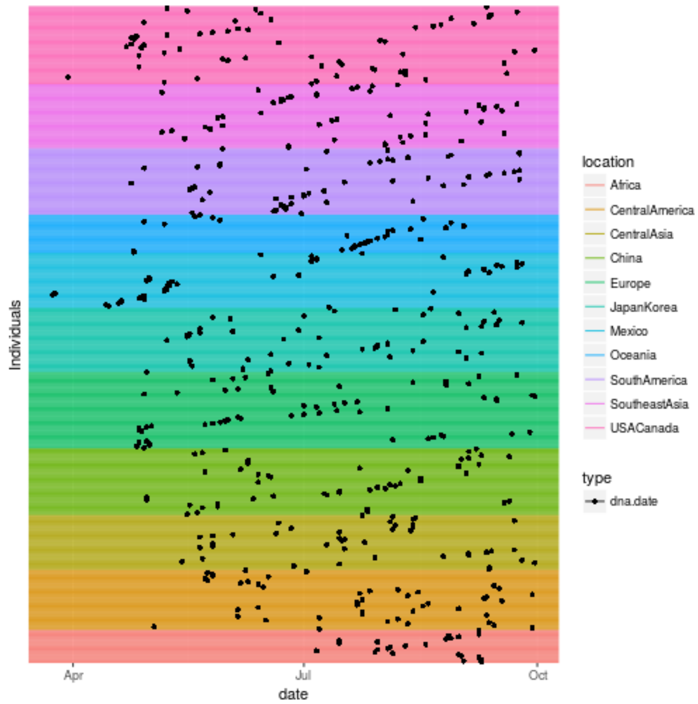
and plotted using *ape*'s standard `plot` function:

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



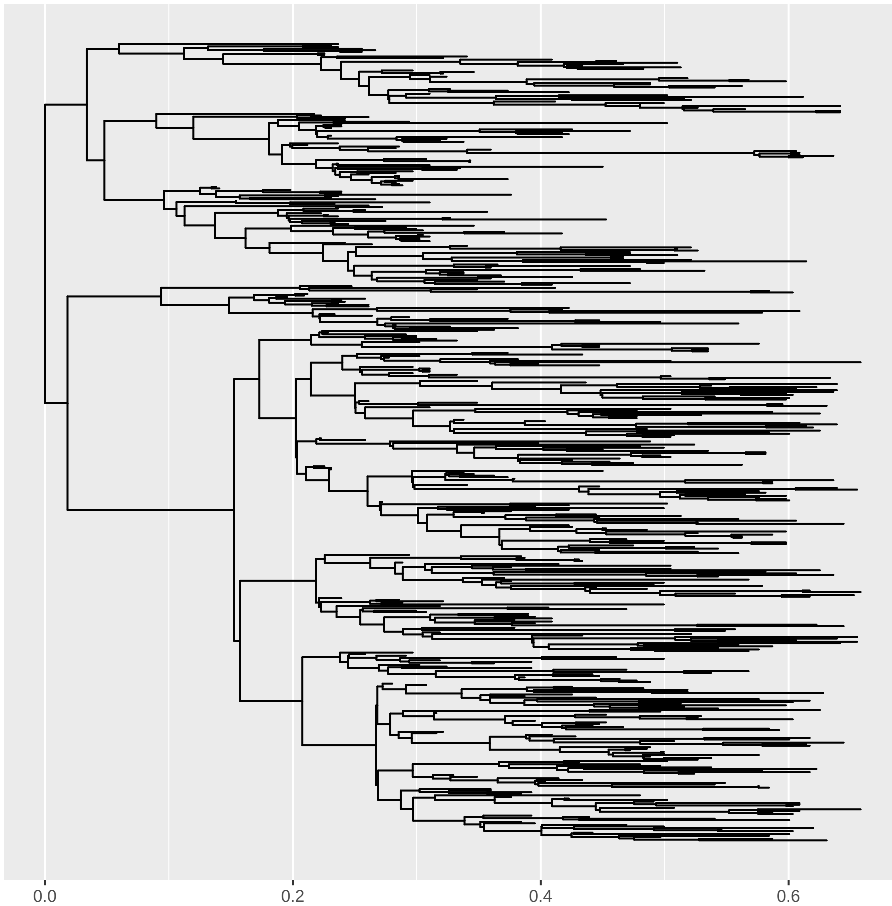
However, we are loosing the temporal information about the samples:

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



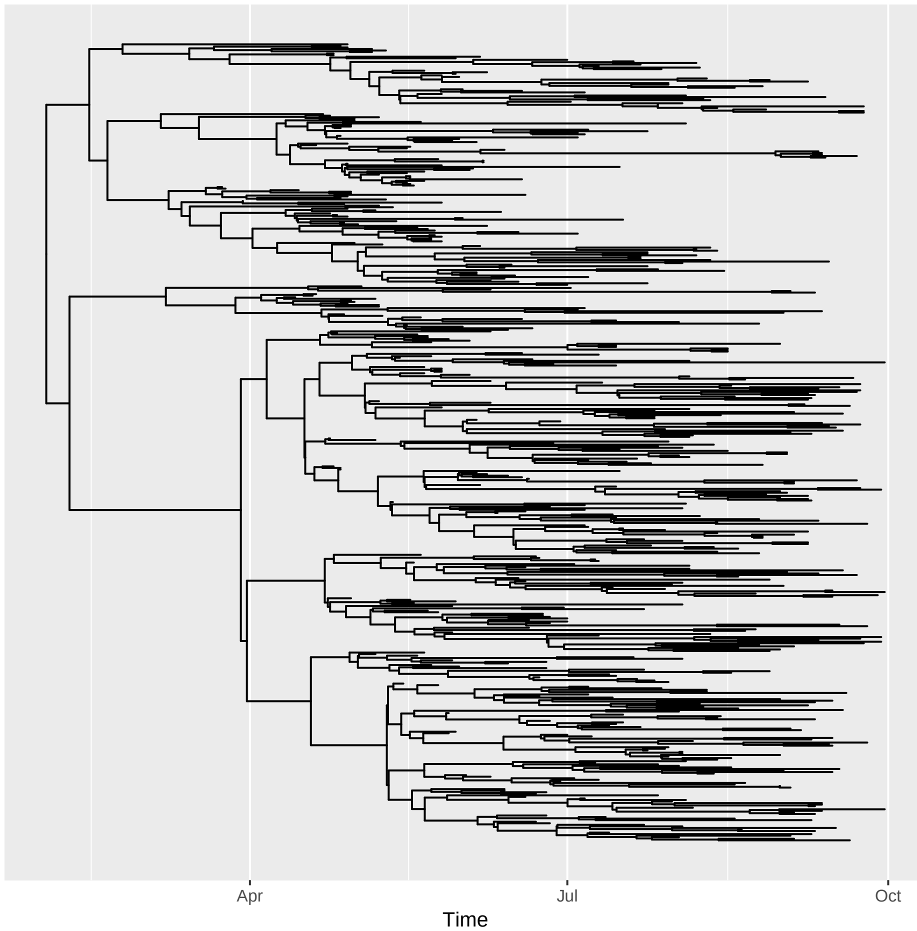
The basic plot of `plotggphy` gives a tree quite similar to *ape*'s:

```
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,  branch.unit = "year")
```



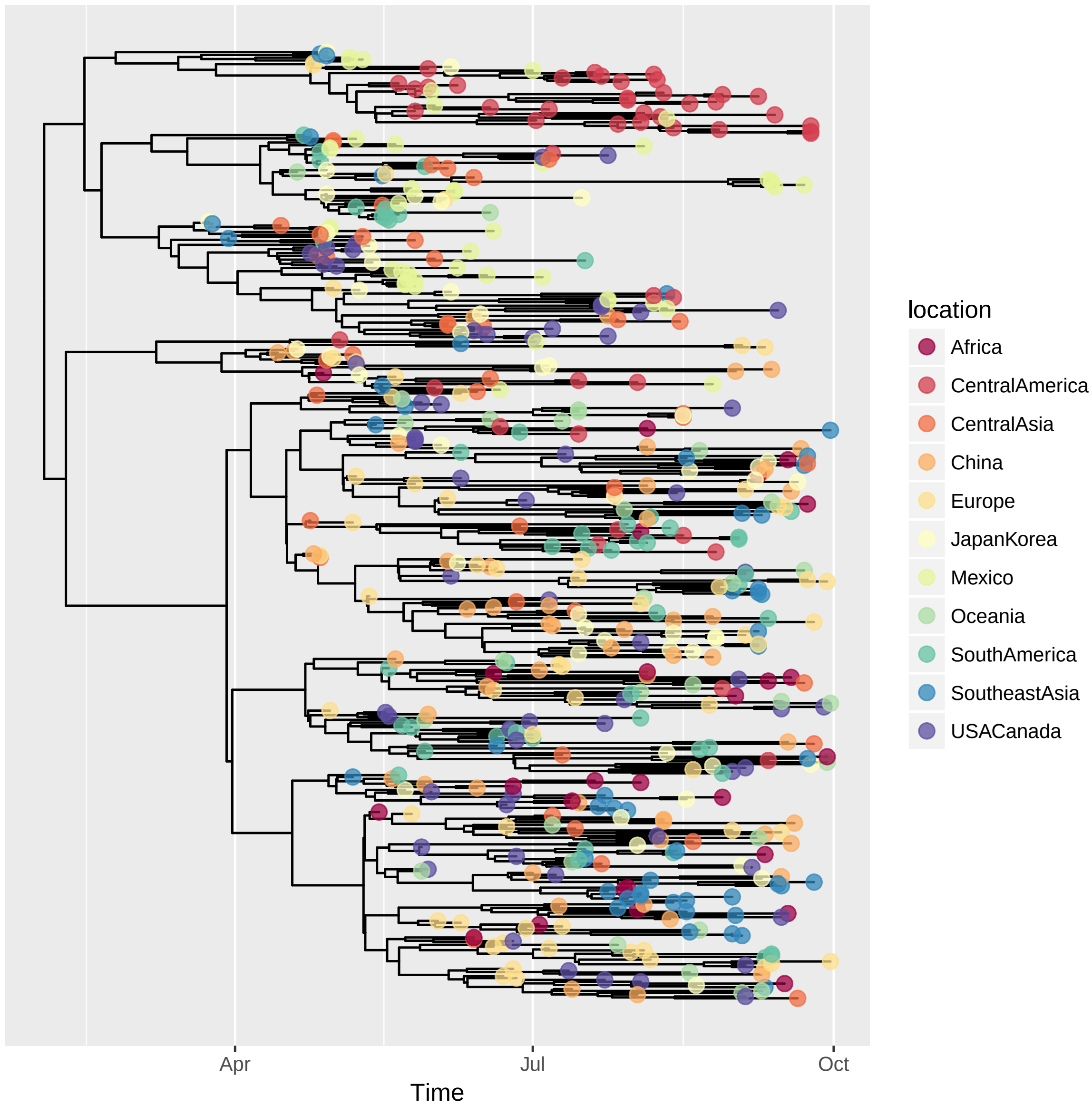
Note that `p` is a graphical (`ggplot`) object, which can be re-used later to generate and modify the plot. Importantly, other attributes can also be used and represented by colors on the tips. For instance, `x` contains information about the location of different individuals:

```
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, branch.unit = "year",
               tip.color = "location", tip.size = 3, tip.alpha = 0.75)
```


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