Definition of the Class `"bitsplits"`

Andrei-Alin Popescu & Emmanuel Paradis

22 December 2020

1 Introduction

Bipartitions (also known as *splits*) are important for several methods in phylogenetics. A bipartition is defined on a set of n elements as two exclusive subsets such that the union of these two subsets is identical to the initial set. A labelled, unrooted phylogenetic tree on n tips allows to define several bipartitions:

- The n terminal branches define n *trivial* bipartitions which are always present in the tree whatever its topology.
- The $n - 3$ internal branches define $n - 3$ $(= m)$ *non-trivial* bipartitions which compositions depend on the topology.
- There is one additional trivial bipartition defined by the empty set on one side and the n tips on the other side.

Bipartitions are particularly used in phylogenetic bootstrap where the bootstrap values are the frequencies of the non-trivial bipartitions counted from the sample of bootstrap trees. Similar countings are done in Bayesian methods to compute posterior clade probabilities.

Historically, `ape` used the functions `prop.part` and `prop.clades` to compute bootstrap values, but these are limited when n is large.

2 Rationale

In 2011, Andrei designed the class `"bitsplits"` to store more efficiently bipartitions on a fixed set of tips. The idea is to use the binary (1/0) storage at the bit-level to code the presence/absence of a tip in a given split. For instance, consider the following byte:

```
10000000
```

It could be used to store the information that among eight tips, tip 1 is present in the split while all others are absent. This a trivial split. The formal way to denote this split is '1|2345678'. Thus, this could also be coded as:

```
01111111
```

If tips 1 and 3 are sisters in the tree, the split '13|245678' can be coded in two ways:

```
10100000
01011111
```

This can be generalised to more than eight tips by using more than one byte (reminder: 1 byte = 8 bits), so that for n tips $\lceil n/8 \rceil$ bytes are needed.

The class `"bitsplits"` is used to model a set of weighted splits in R. The splits themselves are modelled using an array of bytes (mode `"raw"` in R) in the following way. Suppose we have n taxa, and that we choose an arbitrary ordering on these taxa. Then we model a split on these taxa by using a bitmask of size n, where if bit i is set, taxon i belongs to one half of the split, and if it is not set it belongs to the other half of the split. Since often we will have more taxa than could be stored in a single byte, we will use a vector of size $\lceil n/8 \rceil$ of bytes, which is basically a breakdown into "byte-sized chunks" of the entire set of bits which needs to be used. In the event that we have trailing bits which are not used, user code should make no assumptions of the values of these bits.

3 Structure

An object of class `"bitsplits"` is a list with the following elements:

1. `matsplit`: a matrix of mode `"raw"`. This is used to model the actual splits in the set. This matrix has $\lceil n/8 \rceil$ rows and m columns. Column i is a vector which models a split as described above;
2. `labels`: a vector of mode character of length n with the taxon labels;
3. `freq`: a numeric vector of length m of split weights (or frequencies).

4 Implementation in ape

The class `"bitsplits"` was introduced in `ape` 2.8 (October 2011). The implementation has evolved over several releases and is now stable. Currently, there are two main functions:

- `bitsplits(x)` returns the bipartitions for either a single tree or a list of trees (`x`).
- `countBipartitions(phy, X)` returns the frequencies of the bipartitions from the reference tree `phy` observed in the list of trees `X`.

There are also a few methods and generic functions such as `as.bitsplits` (see `?bitsplits`) and functions to test split compatibility (see `?is.compatible`).

`bitsplits()` is called by the standard bootstrapping function `boot.phylo` when working with unrooted trees: this is 1000's times faster than calling `prop.part` when $n \geq 100$.

It is important to note that the two functions listed above work on *unrooted* trees and that it is up to the user to do the necessary checks (see the example in `?bitsplits`).