OncoSimulR: forward genetic simulation in asexual populations with arbitrary epistatic interactions and a focus on modeling tumor progression.

Ramon Diaz-Uriarte
Dept. Biochemistry, Universidad Autónoma de Madrid
Instituto de Investigaciones Biomédicas "Alberto Sols" (UAM-CSIC)
Madrid, Spain*
http://ligarto.org/rdiaz

2016-04-15 (Rev: 0d47718)

Contents

1 Introduction

OncoSimulR was originally developed to simulate tumor progression using several models of tumor progression with emphasis on allowing users to set restrictions in the accumulation of mutations as specified, for example, by Oncogenetic Trees (OT; [**?**, **?**]) or Conjunctive Bayesian Networks (CBN; [**?**, **?**, **?**]), with the possibility of adding passenger mutations to the simulations and several types of sampling.

Since then, OncoSimulR has been vastly extended to allow you to specify other types of restrictions in the accumulation of genes, as in the "semimonotone" model of Farahani and Lagergren [**?**] and the XOR models of Korsunsky and collaborators [**?**]. Moreover, different fitness effects related to the order in which mutations appear can also be incorporated, involving arbitrary numbers of genes. This is different from "restrictions in the accumulation of mutations". With order effects, shown empirically in a recent cancer paper by Ortmann and collaborators [**?**], the effect of having both mutations "A" and "B" differs depending on whether "A" appeared before or after "B". More generally, now OncoSimulR also allows you to specify arbitrary epistatic interactions between arbitrary collections of genes and to model, for example, synthetic mortality or synthetic viability (again, involving an arbitrary number of genes, some of which might also depend on other genes, or show order effects with other genes). Moreover, it is possible to specify the above interactions in terms of modules, not genes. This idea is discussed in, for example, [**?**, **?**]: the restrictions encoded in, say, CBNs or OT can be considered to apply not to genes, but to modules, where each module is a set of genes (and the intersection between modules is the empty set) that performs a specific biological function. Modules, then, play the role of a "union operation" over the set of genes in a module. In addition, arbitrary numbers of genes without interactions (and with fitness effects coming from any distribution you might want) are also possible.

The models so far implemented are all continuous time models, which are simulated using the BNB algorithm of Mather et al. [**?**]. The core of the code is implemented in C++, providing for fast execution. Finally, to help with simulation studies, code to simulate random graphs of the kind often seen in CBN, OTs, etc, is also available.

*ramon.diaz@iib.uam.es, rdiaz02@gmail.com

1.1 Key features of OncoSimulR

As mentioned above, OncoSimulR is now a very general package for forward genetic simulation, with applicability well beyond tumor progression. This is a summary of some of the key features:

- You can specify arbitrary interactions between genes, with arbitrary fitness effects, with explicit support for:
  - Restrictions in the accumulations of mutations, as specified by Oncogenetic Trees (OTs), Conjunctive Bayesian Networks (CBNs), semimonotone progression networks, and XOR relationships.
  - Epistatic interactions, including, but not limited to, synthetic viability and synthetic lethality.
  - Order effects.
- You can add passenger mutations.
- More generally, you can add arbitrary numbers of non-interacting genes with arbitrary fitness effects.
- You can allow for deviations from the OT, CBN, semimonotone, and XOR models, specifying a penalty for such deviations (the s_h parameter).
- You can conduct multiple simulations, and sample from them with different temporal schemes and using both whole tumor or single cell sampling.
- Right now, three different models are available, two that lead to exponential growth, one of them loosely based on Bozic et al. [**?**], and another that leads to logistic-like growth, based on McFarland et al. [**?**].
- Code in C++ is available (though not yet callable from R) for using several other models, including the one from Beerenwinkel and collaborators [**?**].
- You can use very large numbers of genes (e.g., see an example of 50000 in section **??**).
- Simulations are generally very fast as I use C++ to implement the BNB algorithm.
- You can obtain the true sequence of events and the phylogenetic relationships between clones.

Further details about the motivation for wanting to simulate data this way in the context of tumor progression can be found in [**?**], where additional comments about model parameters and caveats are discussed. Are there similar programs? The Java program by [**?**] offers somewhat similar functionality to the previous version of OncoSimulR, but it is restricted to at most four drivers (whereas v.1 of OncoSimulR allowed for up to 64), you cannot use arbitrary CBNs or OTs (or XORs or semimonotone graphs) to specify restrictions, there is no allowance for passengers, and a single type of model (a discrete time Galton-Watson process) is implemented. The current functionality of OncoSimulR goes well beyond the the previous version (and, thus, also the TPT of [**?**]) allowing you to specify all types of fitness effects in other general forward genetic simulators such as FFPopSim [**?**], and some that, to our knowledge (e.g., order effects) are not available from any genetics simulator.

1.2 Steps in using OncoSimulR

Using this package will often involve the following steps:

1. Specify the fitness effects: sections **??** and **??**.
2. Simulate cancer progression: section **??**. You can simulate for a single subject or for a set of subjects. You will need to:
   - Decide on a model. This basically amounts to choosing a model with exponential growth ("Exp" or "Bozic") or a model with gompertz-like growth ("McFL"). If exponential growth, you can choose whether the the effects of mutations operate on the death rate ("Bozic") or the birth rate ("Exp")[1].

[1] It is of course possible to do this with the gompertz-like models, but there probably is little reason to do it. McFarland et al. [**?**] discuss this has little effect on their results, for example. In addition, decreasing the death rate will more easily lead to numerical problems as shown in section **??**

- Specify the other parameters of the simulation (when to stop, mutation rate, etc).

Of course, at least for initial playing around, you can use the defaults.

3. Sample from the simulated data: section **??**, and do something with those simulated data (e.g., fit an OT model to them). What you do with the data, however, is outside the scope of this package.

Before anything else, let us load the package. We also explicitly load *graph* and *igraph* for the vignette to work (you do not need that for your usual interactive work). And I set the default color for vertices in igraph.

```
library(OncoSimulR)
library(graph)
library(igraph)
```

```
##
## Attaching package:  'igraph'
```

```
## The following objects are masked from 'package:graph':
##
##    degree, edges, intersection, union
```

```
## The following objects are masked from 'package:stats':
##
##    decompose, spectrum
```

```
## The following object is masked from 'package:base':
##
##    union
```

```
igraph_options(vertex.color = "SkyBlue2")
```

To be explicit, what version are we running?

```
packageVersion("OncoSimulR")
```

```
## [1] '2.0.1'
```

1.3 A temporary note about versions

In this vignette and the documentation I often refer to version 1 (v.1) and version 2 of OncoSimulR. Version 1 is the version available up to, and including, BioConductor v. 3.1. Version 2 of OncoSimulR is available for the current BioConductor development branch, 3.2. If you look at the package version, however, it currently shows as 1.99.x (where x should be a number ≥ 4). That is because of the versioning scheme of BioConductor. This will become 2.0 in the next release of BioConductor.

Summary: if you are using the current development version of BioConductor, or you grab the sources from github (https://github.com/rdiaz02/OncoSimul) you are using what we call version 2. If you see a version in the package that says "1.99.x", where "x" is any number, you are too.

2 Specifying fitness effects

2.1 Introduction to the specification of fitness effects

With OncoSimulR you can specify different types of effects on fitness:

- A special type of epistatic effect that is particularly amenable to be represented as a graph. In this graph, having, say, "B" be a child of "A" means that B can only accumulate if A is already present. This is what OT [**?**, **?**], CBN [**?**, **?**, **?**], progression networks [**?**], and other similar models [**?**] mean. Details are provided in section **??**. Note that this is not an order effect (discussed below): the fitness of a genotype from this DAGs is a function of whether or not the restrictions in the graph are satisfied, not the historical sequence of how they were satisfied.
- Effects where the order in which mutations are acquired matters, as illustrated in section **??**. There is, in fact, empirical evidence of these effects [**?**]. For instance, the fitness of genotype "A, B" would differ depending on whether A or B was acquired first.
- General epistatic effects (e.g., section **??**), including synthetic viability (e.g., section **??**) and synthetic lethality/mortality (e.g., section **??**).
- Genes that have independent effects on fitness (section **??**).

Modules (see section **??**) allow you to specify any of the above effects (except those for genes without interactions, as it would not make sense there) in terms of modules (sets of genes), not individual genes. We will introduce them right after **??**, and continue using them thereafter.

2.1.1 How to specify fitness effects effects

A guiding design principle of OncoSimulR is to try to make the specification of those effects as simple as possible but also as flexible as possible.

Conceptually, the simplest way is to specify the mapping of all genotypes to fitness explicitly. This can be done with OncoSimulR (e.g., see sections **??**, **??** and **??** or the example in **??**), but this only makes sense for subsets of the genes or for very small genotypes, as you probably do not want to be explicit about the mapping of 2^k genotypes to fitness when k is larger than, say, four or five, and definitely not when k is 10.

An alternative general approach followed in many genetic simulators is to specify how particular combinations of alleles modify the wildtype genotype or the genotype that contains the individual effects of the interacting genes (e.g., see equation 1 in the supplementary material for FFPopSim). For example, if we specify that "A" contributes 0.04, "B" contributes 0.03, and "A:B" contributes 0.1, that means that the fitness of the "A, B" genotype is that of the wildtype (1, by default), plus (actually, times —see section **??**) the effects of A, plus (times) the effects of B, plus (times) the effects of "A:B".

As we will see in the examples (e.g., see sections **??**, **??**, **??**) OncoSimulR makes it simple to be explicit about the mapping of specific genotypes, while also using the "how this specific effects modifies previous effects" logic, leading to a flexible specification. This also means that in many cases the same fitness effects can be specified in several different ways.

2.2 Numeric values of fitness effects

We evaluate fitness using the usual (e.g. [**?**, **?**, **?**, **?**]) multiplicative model: fitness is $\prod(1 + s_i)$ where s_i is the fitness effect of gene (or gene interaction) i. In all models except Bozic, this fitness refers to the growth rate (the death rate being fixed to 1[2]). The original model of McFarland [**?**] has a slightly different parameterization, but you can go easily from one to the other (see section **??**).

For the Bozic model, however, the birth rate is set to 1, and the death rate then becomes $\prod(1 - s_i)$.

[2] You can change this if you really want to.

2.2.1 McFarland parameterization

In the original McFarland model [**?**], the effects of drivers contribute to the numerator of the birth rate, and those of the (deleterious) passengers to the denominator as: $\frac{(1+s)^D}{(1-s_p)^P}$, where D and P are, respectively, the total number of drivers and passengers in a genotype, and here the fitness effects of all drivers is the same (s) and that of all passengers the same too (s_p). However, we can map from this ratio to the usual product of terms by using a different value of s_p, that we will call $s_{pp} = -s_p/(1+s_p)$ (see [**?**], his eq. 2.1 in p. 9). This reparameterization applies to v.2. In v.1 we use the same parameterization as in the original one in McFarland [**?**].

2.2.2 No viability of clones and types of models

For all models where fitness affects directly the birth rate (for now, all except Bozic), if you specify that some event (say, mutating gene A) has $s_A \leq -1$, if that event happens then birth rate becomes zero which is taken to indicate that the clone is not even viable and thus disappears immediately without any chance for mutation[3].

Models based on Bozic, however, have a birth rate of 1[4] and mutations affect the death rate. In this case, a death rate larger than birth rate, per se, does not signal immediate extinction and, moreover, even for death rates that are a few times larger than birth rates, the clone could mutate before becoming extinct[5]. How do we signal immediate extinction or no viability in this case? You can set the value of $s = -\infty$.

In general, if you want to identify some mutations or some combinations of mutations as leading to immediate extinction, no viability, of the affected clone, set it to $-\infty$ as this would work even if we later change how birth rates of 0 are handled. Most examples below evaluate fitness by its effects on the birth rate. You can see one where we do it both ways in Section **??**.

2.3 Genes without interactions

This is a imple scenario. Each gene, i, has a fitness effect s_i if mutated. The s_i can come from any distribution you want. As an example let's use three genes. We know there are no order effects, but we will also see what happens if we examine genotypes as ordered.

```
ai1 <- evalAllGenotypes(allFitnessEffects(
    noIntGenes = c(0.05, -.2, .1)), order = FALSE)
```

We can easily verify the first results:

```
ai1

##   Genotype Fitness
```

[3] This is a shortcut that we take because we think that it is what you mean. Note, however, that technically a clone with birth rate of 0 might have a non-zero probability of mutating before becoming extinct because in the continuous time model we use mutation is not linked to reproduction. In the present code, we are not allowing for any mutation when birth rate is 0. There are other options, but none which I find really better. An alternative implementation makes a clone immediately extinct if and only if any of the $s_i = -\infty$. However, we still need to handle the case with $s_i < -1$ as a special case. We either make it identical to the case with any $s_i = -\infty$ or for any $s_i > -\infty$ we set $(1 + s_i) = \max(0, 1 + s_i)$ (i.e., if $s_i < -1$ then $(1 + s_i) = 0$), to avoid obtaining negative birth rates (that make no sense) and the problem of multiplying an even number of negative numbers. I think only the second would make sense as an alternative.

[4] In the C++ code there is a different model, not directly callable from R for now, called "bozic2" that is slightly different. These comments apply to the model that is right now callable from R

[5] We said "a few times". For a clone of population size 1 —which is the size at which all clones start from mutation—, if death rate is, say, 90 but birth rate is 1, the probability of mutating before becoming extinct is very, very close to zero for all reasonable values of mutation rate

```
## 1        1   1.050
## 2        2   0.800
## 3        3   1.100
## 4     1, 2   0.840
## 5     1, 3   1.155
## 6     2, 3   0.880
## 7  1, 2, 3   0.924
```

```
all(ai1[, "Fitness"]  == c( (1 + .05), (1 - .2), (1 + .1),
       (1 + .05) * (1 - .2),
       (1 + .05) * (1 + .1),
       (1 - .2) * (1 + .1),
       (1 + .05) * (1 - .2) * (1 + .1)))
```

```
## [1] TRUE
```

And we can see that considering the order of mutations makes no difference:

```
(ai2 <- evalAllGenotypes(allFitnessEffects(
    noIntGenes = c(0.05, -.2, .1)), order = TRUE))
```

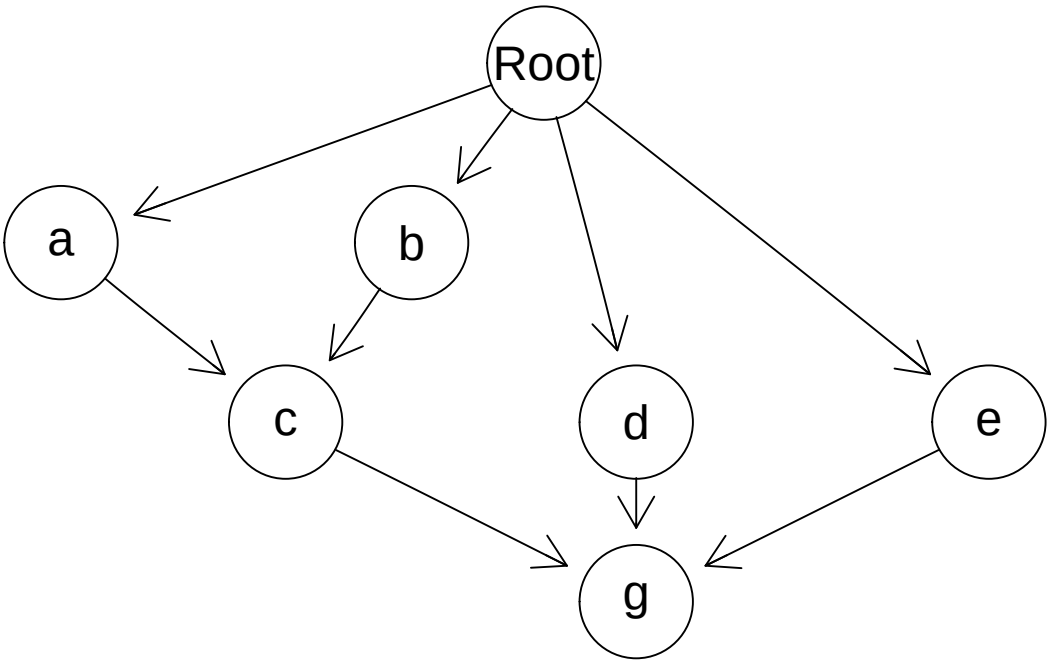
```
##     Genotype Fitness
## 1          1   1.050
## 2          2   0.800
## 3          3   1.100
## 4      1 > 2   0.840
## 5      1 > 3   1.155
## 6      2 > 1   0.840
## 7      2 > 3   0.880
## 8      3 > 1   1.155
## 9      3 > 2   0.880
## 10 1 > 2 > 3   0.924
## 11 1 > 3 > 2   0.924
## 12 2 > 1 > 3   0.924
## 13 2 > 3 > 1   0.924
## 14 3 > 1 > 2   0.924
## 15 3 > 2 > 1   0.924
```

2.4 Restrictions in the order of mutations as extended posets

2.4.1 AND, OR, XOR relationships

The literature on oncogenetic trees, CBNs, etc, has used graphs as a way of showing the restrictions in the order in which mutations can accumulate. The meaning of "convergent arrows" in these graphs, however, differs. In Figure 1 of [**?**] we are shown a simple diagram that illustrates the three basic different meanings of convergent arrows using two parental nodes. We will illustrate it here with three. Suppose we focus on node "g" in the following figure (we will create it shortly)

```
data(examplesFitnessEffects)
plot(examplesFitnessEffects[["cbn1"]])
```



- In relationships of the type used in **Conjunctive** Bayesian Networks (CBN) [**?**, e.g.], we are modeling an **AND** relationship, also called **CMPN** by [**?**] or **monotone** relationship by [**?**]. If the relationship in the graph is fully respected, then "g" will only appear if all of "c", "d", and "e" are already mutated.
- **Semimonotone** relationships *sensu* [**?**] or **DMPN** *sensu* [**?**] are **OR** relationships: "g" will appear if one or more of "c", "d", or "e" are already mutated.
- **XMPN** relationships ([**?**]) are **XOR** relationships: "g" will be present only if exactly one of "c", "d", or "e" is present.

Note that oncogenetic trees ([**?**, **?**]) need not deal with the above distinctions, since the DAGs are trees: no node has more than one incoming connection or more than one parent[6].

To have a flexible way of specifying all of these restrictions, we will want to be able to say what kind of dependency each child node has on its parents.

2.4.2 Fitness effects

Those DAGs specify dependencies and, as explained in [**?**], it is simple to map them to a simple evolutionary model: any set of mutations that does not conform to the restrictions encoded in the graph will have a fitness of 0. However, we might not want to require absolute compliance with the DAG. This means we might want to allow deviations from the DAG with a corresponding penalization that is, however, not identical to setting fitness to 0 (again, see [**?**]). This we can do by being explicit about the fitness effects of these deviations from the restrictions encoded in the DAG. We will use below a column of `s` for the fitness effect when the restrictions are satisfied and a column of `sh` when they are not. (See also **??** for the details about the meaning of the fitness effects).

That way of specifying fitness effects makes it also trivial to use the model in Hjelm et al. [**?**] where all mutations might be allowed to occur, but the presence of some mutations increases the probability of

[6] OTs and CBNs have some other technical differences about the underlying model they assume, such as the exponential waiting time in CBNs. We will not discuss them here.

occurrence of other mutations. For example, the values of `sh` could be all small positive ones (or for mildly deleterious effects, small negative numbers), while the values of `s` are much larger positive numbers.

2.4.3 Extended posets

In version 1 of this package we used posets in the sense of [**?**, **?**], as explained in section **??** and in the help for `poset`. Here, we continue using two columns, that specify parents and children, but we add columns for the specific values of fitness effects (both s and sh —i.e., fitness effects for what happens when restrictions are and are not satisfied) and for the type of dependency as explained in section **??**.

We can now illustrate the specification of different fitness effects.

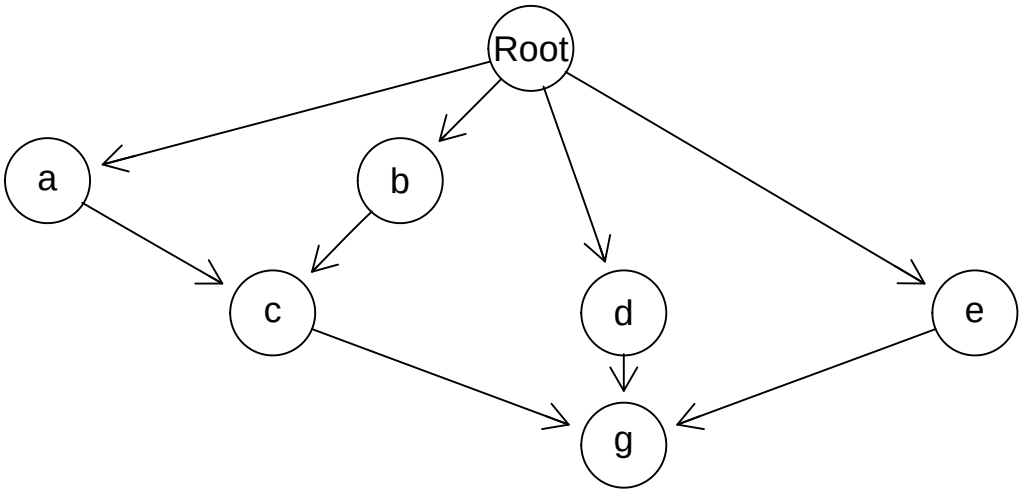
2.4.4 A first conjunction (AND) example

```
cs <-  data.frame(parent = c(rep("Root", 4), "a", "b", "d", "e", "c"),
                  child = c("a", "b", "d", "e", "c", "c", rep("g", 3)),
                  s = 0.1,
                  sh = -0.9,
                  typeDep = "MN")

cbn1 <- allFitnessEffects(cs)
```

(We skip one letter, just to show that names need not be consecutive or have any particular order.)

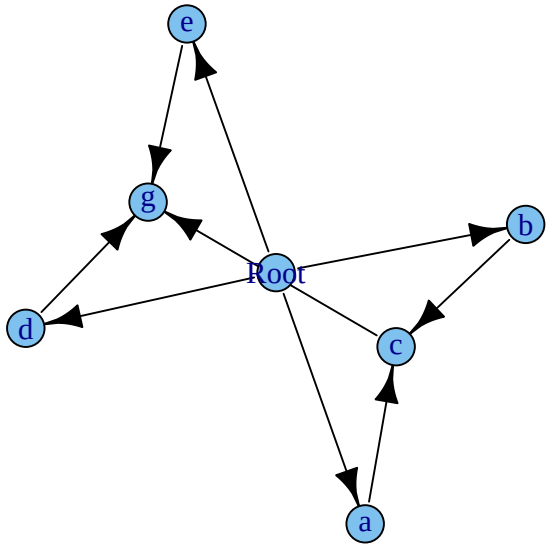
We can get a graphical representation using the default "graphNEL"

```
plot(cbn1)
```



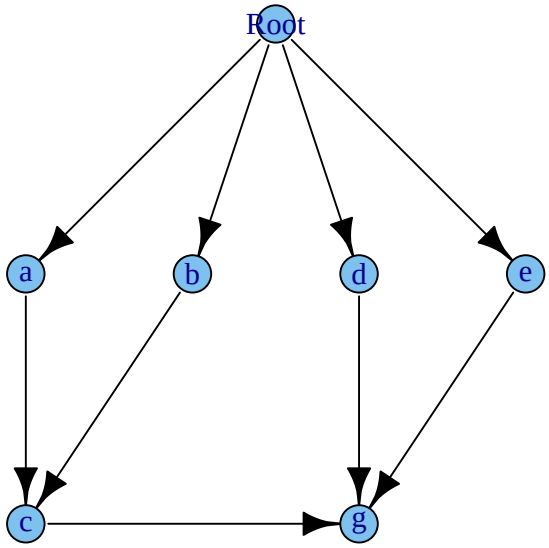
or one using "igraph":

```
plot(cbn1, "igraph")
```



Since this is a tree, the reingold.tilford layout is probably the best here, so you might want to use that:

```
library(igraph) ## to make the reingold.tilford layout available
plot(cbn1, "igraph", layout = layout.reingold.tilford)
```



And what is the fitness of all genotypes?

```
gfs <- evalAllGenotypes(cbn1, order = FALSE)

gfs[1:15, ]
```

```
##    Genotype Fitness
## 1         a    1.10
## 2         b    1.10
## 3         c    0.10
## 4         d    1.10
## 5         e    1.10
## 6         g    0.10
## 7      a, b    1.21
## 8      a, c    0.11
## 9      a, d    1.21
## 10     a, e    1.21
## 11     a, g    0.11
## 12     b, c    0.11
## 13     b, d    1.21
## 14     b, e    1.21
## 15     b, g    0.11
```

You can verify that for each genotype, if a mutation is present without all of its dependencies present, you get a $(1-0.9)$ multiplier, and you get a $(1+0.1)$ multiplier for all the rest with its direct parents satisfied. For example, genotypes "a", or "b", or "d", or "e" have fitness $(1+0.1)$, genotype "a, b, c" has fitness $(1+0.1)^3$, but genotype "a, c" has fitness $(1+0.1)(1-0.9) = 0.11$.

2.4.5 A second conjunction example

Let's try a first attempt at a somewhat more complex example, where the fitness consequences of different genes differ.

```
c1 <- data.frame(parent = c(rep("Root", 4), "a", "b", "d", "e", "c"),
                 child = c("a", "b", "d", "e", "c", "c", rep("g", 3)),
                 s = c(0.01, 0.02, 0.03, 0.04, 0.1, 0.1, rep(0.2, 3)),
                 sh = c(rep(0, 4), c(-.1, -.2), c(-.05, -.06, -.07)),
                 typeDep = "MN")

try(fc1 <- allFitnessEffects(c1))
```

That is an error because the "sh" varies within a child, and we do not allow that for a poset-type specification, as it is ambiguous. If you need arbitrary fitness values for arbitrary combinations of genotypes, you can specify them using epistatic effects as in section **??** and order effects as in section **??**.

Why do we need to specify as many "s" and "sh" as there are rows (or a single one, that gets expanded to those many) when the "s" and "sh" are properties of the child node, not of the edges? Because, for ease, we use a data.frame.

We fix the error in our specification. Notice that the "sh" is not set to -1 in these examples. If you want strict compliance with the poset restrictions, you should set $sh = -1$ or, better yet, $sh = -\infty$ (see section **??**), but having an $sh > -1$ will lead to fitnesses that are > 0 and, thus, is a way of modeling small deviations from the poset (see discussion in [**?**]).

Note that for those nodes that depend only on "Root" the type of dependency is irrelevant.

```
c1 <- data.frame(parent = c(rep("Root", 4), "a", "b", "d", "e", "c"),
                 child = c("a", "b", "d", "e", "c", "c", rep("g", 3)),
                 s = c(0.01, 0.02, 0.03, 0.04, 0.1, 0.1, rep(0.2, 3)),
                 sh = c(rep(0, 4), c(-.9, -.9), rep(-.95, 3)),
                 typeDep = "MN")

cbn2 <- allFitnessEffects(c1)
```

We could get graphical representations but the figures would be the same as in the example in section **??**, since the structure has not changed, only the numeric values.

What is the fitness of all possible genotypes? Here, order of events *per se* does not matter, beyond that considered in the poset. In other words, the fitness of genotype "a, b, c" is the same no matter how we got to "a, b, c". What matters is whether or not the genes on which each of "a", "b", and "c" depend are present or not (I only show the first 10 genotypes)

```
gcbn2 <- evalAllGenotypes(cbn2, order = FALSE)
gcbn2[1:10, ]
```

```
##    Genotype Fitness
## 1         a  1.0100
## 2         b  1.0200
## 3         c  0.1000
## 4         d  1.0300
## 5         e  1.0400
## 6         g  0.0500
## 7      a, b  1.0302
## 8      a, c  0.1010
```

```
## 9       a, d  1.0403
## 10      a, e  1.0504
```

Of course, if we were to look at genotypes but taking into account order of occurrence of mutations, we would see no differences

```
gcbn2o <- evalAllGenotypes(cbn2, order = TRUE, max = 1956)
gcbn2o[1:10, ]
```

```
##    Genotype Fitness
## 1         a  1.0100
## 2         b  1.0200
## 3         c  0.1000
## 4         d  1.0300
## 5         e  1.0400
## 6         g  0.0500
## 7     a > b  1.0302
## 8     a > c  0.1010
## 9     a > d  1.0403
## 10    a > e  1.0504
```

(The `max = 1956` is there so that we show all the genotypes, even if they are more than 256, the default.)

You can check the output and verify things are as they should. For instance:

```
all.equal(
        gcbn2[c(1:21, 22, 28, 41, 44, 56, 63 ) , "Fitness"],
        c(1.01, 1.02, 0.1, 1.03, 1.04, 0.05,
          1.01 * c(1.02, 0.1, 1.03, 1.04, 0.05),
          1.02 * c(0.10, 1.03, 1.04, 0.05),
          0.1 * c(1.03, 1.04, 0.05),
          1.03 * c(1.04, 0.05),
          1.04 * 0.05,
          1.01 * 1.02 * 1.1,
          1.01 * 0.1 * 0.05,
          1.03 * 1.04 * 0.05,
          1.01 * 1.02 * 1.1 * 0.05,
          1.03 * 1.04 * 1.2 * 0.1, ## notice this
          1.01 * 1.02 * 1.03 * 1.04 * 1.1 * 1.2
          ))
```

```
## [1] TRUE
```

A particular one that is important to understand is

```
gcbn2[56, ] ## this is d, e, g, c
```

```
##      Genotype  Fitness
## 56 c, d, e, g 0.128544
```

```
all.equal(gcbn2[56, "Fitness"], 1.03 * 1.04 * 1.2 * 0.10)
```

```
## [1] TRUE
```

where "g" is taken as if its dependencies are satisfied (as "c", "d", and "e" are present) even when the dependencies of "c" are not satisfied (and that is why the term for "c" is 0.9).

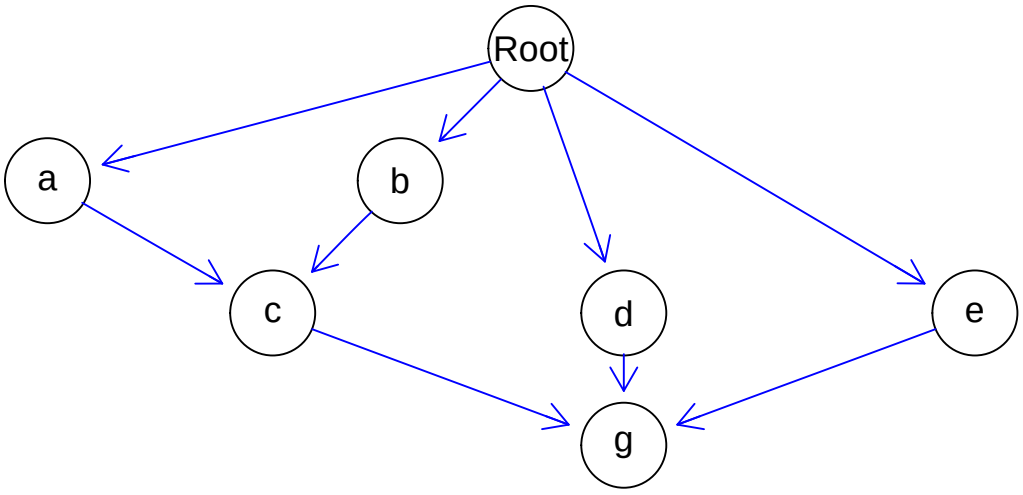
2.4.6 A semimonotone or "OR" example

We will reuse the above example, changing the type of relationship:

```
s1 <- data.frame(parent = c(rep("Root", 4), "a", "b", "d", "e", "c"),
                 child = c("a", "b", "d", "e", "c", "c", rep("g", 3)),
                 s = c(0.01, 0.02, 0.03, 0.04, 0.1, 0.1, rep(0.2, 3)),
                 sh = c(rep(0, 4), c(-.9, -.9), rep(-.95, 3)),
                 typeDep = "SM")

smn1 <- allFitnessEffects(s1)
```

It looks like this (where edges are shown in blue to denote the semimonotone relationship):

```
plot(smn1)
```




```
gsmn1 <- evalAllGenotypes(smn1, order = FALSE)
```

Having just one parental dependency satisfied is now enough, in contrast to what happened before. For instance:

```
gcbn2[c(8, 12, 22), ]
```

```
##    Genotype Fitness
## 8      a, c 0.10100
## 12     b, c 0.10200
## 22  a, b, c 1.13322
```

```
gsmn1[c(8, 12, 22), ]
```

```
##    Genotype Fitness
## 8      a, c 1.11100
## 12     b, c 1.12200
## 22  a, b, c 1.13322
```

```
gcbn2[c(20:21, 28), ]
```

```
##    Genotype Fitness
## 20     d, g 0.05150
## 21     e, g 0.05200
```

```
## 28  a, c, g 0.00505
```

```
gsmn1[c(20:21, 28), ]
```

```
##    Genotype Fitness
## 20     d, g  1.2360
## 21     e, g  1.2480
## 28  a, c, g  1.3332
```

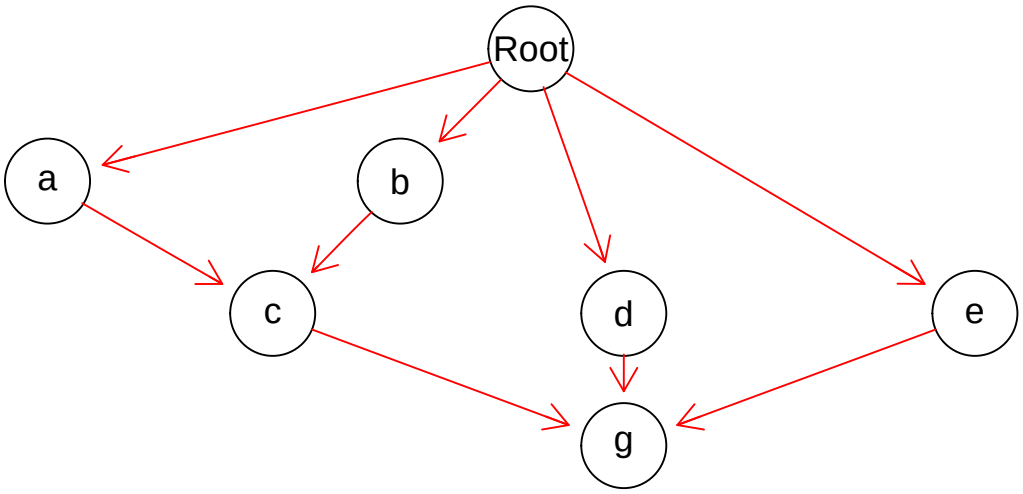
2.4.7 An "XMPN" or "XOR" example

Again, we reuse the example above, changing the type of relationship:

```
x1 <- data.frame(parent = c(rep("Root", 4), "a", "b", "d", "e", "c"),
                 child = c("a", "b", "d", "e", "c", "c", rep("g", 3)),
                 s = c(0.01, 0.02, 0.03, 0.04, 0.1, 0.1, rep(0.2, 3)),
                 sh = c(rep(0, 4), c(-.9, -.9), rep(-.95, 3)),
                 typeDep = "XMPN")

xor1 <- allFitnessEffects(x1)
```

It looks like this (edges in red to denote the "XOR" relationship):

```
plot(xor1)
```




```
gxor1 <- evalAllGenotypes(xor1, order = FALSE)
```

Whenever "c" is present with both "a" and "b", the fitness component for "c" will be $(1 - 0.1)$. Similarly for "g" (if more than one of "d", "e", or "c" is present, it will show as $(1 - 0.05)$). For example:

```
gxor1[c(22, 41), ]
```

```
##    Genotype Fitness
## 22  a, b, c 0.10302
## 41  d, e, g 0.05356
```

```
c(1.01 * 1.02 * 0.1, 1.03 * 1.04 * 0.05)
```

```
## [1] 0.10302 0.05356
```

However, having just both "a" and "b" is identical to the case with CBN and the monotone relationship (see sections **??** and **??**). If you want the joint presence of "a" and "b" to result in different fitness than the product of the individual terms, without considering the presence of "c", you can specify that using general epistatic effects (section **??**).

We also see a very different pattern compared to CBN (section **??**) here:

```
gxor1[28, ]
```

```
##    Genotype Fitness
## 28  a, c, g  1.3332
```

```
1.01 * 1.1 * 1.2
```

```
## [1] 1.3332
```

as exactly one of the dependencies for both "c" and "g" are satisfied.

But

```
gxor1[44, ]
```

```
##      Genotype  Fitness
## 44 a, b, c, g 0.123624
```

```
1.01 * 1.02 * 0.1 * 1.2
```

```
## [1] 0.123624
```

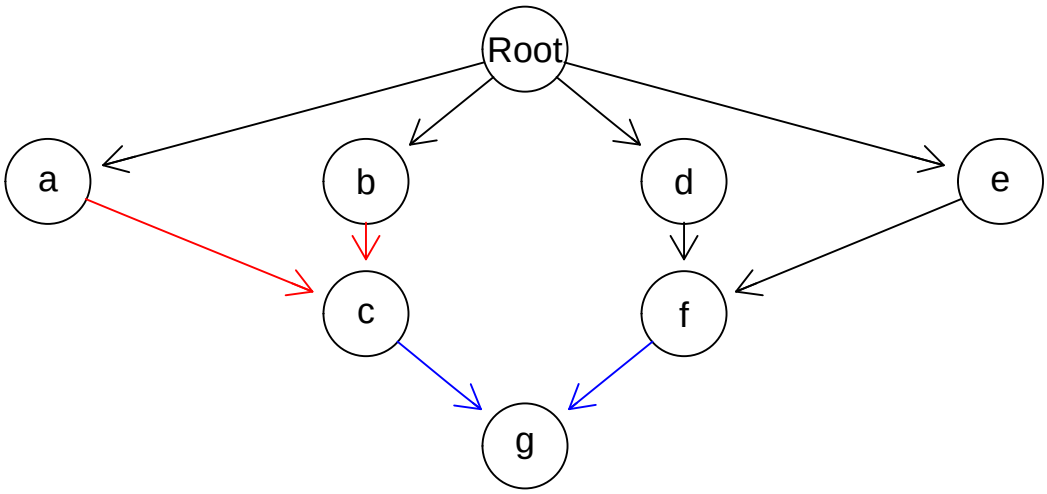
is the result of a 0.1 for "c" (and a 1.2 for "g" that has exactly one of its dependencies satisfied).

2.4.8 Posets: the three types of relationships

```
p3 <- data.frame(parent = c(rep("Root", 4), "a", "b", "d", "e", "c", "f"),
                 child = c("a", "b", "d", "e", "c", "c", "f", "f", "g", "g"),
                 s = c(0.01, 0.02, 0.03, 0.04, 0.1, 0.1, 0.2, 0.2, 0.3, 0.3),
                 sh = c(rep(0, 4), c(-.9, -.9), c(-.95, -.95), c(-.99, -.99)),
                 typeDep = c(rep("--", 4),
                     "XMPN", "XMPN", "MN", "MN", "SM", "SM"))
fp3 <- allFitnessEffects(p3)
```

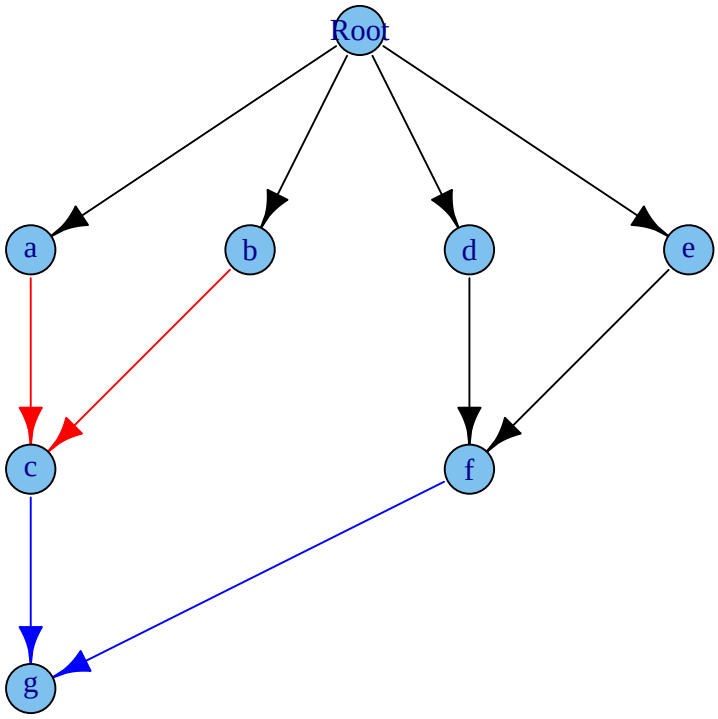
This is how it looks like:

```
plot(fp3)
```



We can also use "igraph":

```
plot(fp3, "igraph", layout.reingold.tilford)
```



```
gfp3 <- evalAllGenotypes(fp3, order = FALSE)
```

Let's look at a few:

```
gfp3[c(9, 24, 29, 59, 60, 66, 119, 120, 126, 127), ]
```

```
##                Genotype   Fitness
## 9                  a, c 1.1110000
## 24                 d, f 0.0515000
## 29              a, b, c 0.1030200
## 59              c, f, g 0.0065000
## 60              d, e, f 1.2854400
## 66           a, b, c, f 0.0051510
## 119       c, d, e, f, g 0.1671072
## 120    a, b, c, d, e, f 0.1324260
## 126    b, c, d, e, f, g 1.8749428
## 127 a, b, c, d, e, f, g 0.1721538
```

```
c(1.01 * 1.1, 1.03 * .05, 1.01 * 1.02 * 0.1, 0.1 * 0.05 * 1.3,
  1.03 * 1.04 * 1.2, 1.01 * 1.02 * 0.1 * 0.05,
  0.1 * 1.03 * 1.04 * 1.2 * 1.3,
  1.01 * 1.02 * 0.1 * 1.03 * 1.04 * 1.2,
  1.02 * 1.1 * 1.03 * 1.04 * 1.2 * 1.3,
  1.01 * 1.02 * 1.03 * 1.04 * 0.1 * 1.2 * 1.3)
```

```
##  [1] 1.1110000 0.0515000 0.1030200 0.0065000 1.2854400 0.0051510 0.1671072 0.1324260
##  [9] 1.8749428 0.1721538
```

As before, looking at the order of mutations makes no difference (look at the test directory to see a test that verifies this assertion).

2.5 Modules

As already mentioned, we can think in all the effects of fitness in terms not of individual genes but, rather, modules. This idea is discussed in, for example, [**?**, **?**]: the restrictions encoded in, say, the DAGs can be considered to apply not to genes, but to modules, where each module is a set of genes (and the intersection between modules is the empty set). Modules, then, play the role of a "union operation" over sets of genes. Of course, if we can use modules for the restrictions in the DAGs we should also be able to use them for epistasis and order effects, as we will see later (e.g., **??**).

2.5.1 What does a module provide

Modules can provide very compact ways of specifying relationships when you want to, well, model the existence of modules. For simplicity suppose there is a module, "A", made of genes "a1" and "a2", and a module "B", made of a single gene "b1". Module "B" can mutate if module "A" is mutated, but mutating both "a1" and "a2" provides no additional fitness advantage compared to mutating only a single one of them. We can specify this as:

```
s <- 0.2
sboth <- (1/(1 + s)) - 1
m0 <- allFitnessEffects(data.frame(
    parent = c("Root", "Root", "a1", "a2"),
    child = c("a1", "a2", "b", "b"),
```

```
    s = s,
    sh = -1,
    typeDep = "OR"),
                        epistasis = c("a1:a2" = sboth))
evalAllGenotypes(m0, order = FALSE, addwt = TRUE)
```

```
##    Genotype Fitness
## 1        wt    1.00
## 2        a1    1.20
## 3        a2    1.20
## 4         b    0.00
## 5    a1, a2    1.20
## 6     a1, b    1.44
## 7     a2, b    1.44
## 8 a1, a2, b    1.44
```

Note that we need to add an epistasis term, with value "sboth" to capture the idea of "mutating both "a1" and "a2" provides no additional fitness advantage compared to mutating only a single one of them"; see details in section **??**.

Now, specify it using modules:

```
s <- 0.2
m1 <- allFitnessEffects(data.frame(
    parent = c("Root", "A"),
    child = c("A", "B"),
    s = s,
    sh = -1,
    typeDep = "OR"),
                        geneToModule = c("Root" = "Root",
                                         "A" = "a1, a2",
                                         "B" = "b1"))
evalAllGenotypes(m1, order = FALSE, addwt = TRUE)
```

```
##     Genotype Fitness
## 1         wt    1.00
## 2         a1    1.20
## 3         a2    1.20
## 4         b1    0.00
## 5     a1, a2    1.20
## 6     a1, b1    1.44
## 7     a2, b1    1.44
## 8 a1, a2, b1    1.44
```

This captures the ideas directly. The typing savings here are small, but they can be large with modules with many genes.

2.5.2 Specifying modules

How do you specify modules? The general procedure is simple: you pass a vector that makes explicit the mapping from modules to sets of genes. We just saw an example. There are several additional examples such as **??**, **??**, **??**.

Why do we force you to specify "Root" = "Root"? We could check for it, and add it if it is not present.

But we want you to be explicit (and we want to avoid you shooting yourself in the foot having a gene that is not the root of the tree but is called "Root", etc).

It is important to note that, once you specify modules, we expect all of the relationships (except those that involve the non interacting genes) to be specified as modules. Thus, all elements of the epistasis, posets (the DAGs) and order effects components should be specified in terms of modules. But you can, of course, specify a module as containing a single gene (and a single gene with the same name as the module).

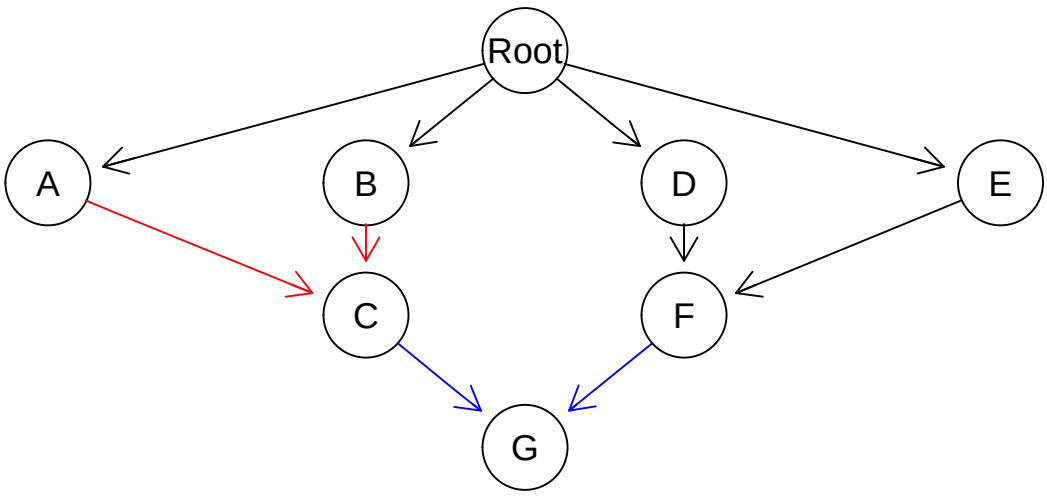
2.5.3 Modules and posets again: the three types of relationships and modules

We use the same specification of poset, but add modules. To keep it manageable, we only add a few genes for some modules, and have some modules with a single gene. Beware that the number of genotypes is starting to grow quite fast, though. We capitalize to differentiate modules (capital letters) from genes (lowercase with a number), but this is not needed.

```
p4 <- data.frame(parent = c(rep("Root", 4), "A", "B", "D", "E", "C", "F"),
                 child = c("A", "B", "D", "E", "C", "C", "F", "F", "G", "G"),
                 s = c(0.01, 0.02, 0.03, 0.04, 0.1, 0.1, 0.2, 0.2, 0.3, 0.3),
                 sh = c(rep(0, 4), c(-.9, -.9), c(-.95, -.95), c(-.99, -.99)),
                 typeDep = c(rep("--", 4),
                     "XMPN", "XMPN", "MN", "MN", "SM", "SM"))
fp4m <- allFitnessEffects(p4,
                          geneToModule = c("Root" = "Root", "A" = "a1",
                              "B" = "b1, b2", "C" = "c1",
                              "D" = "d1, d2", "E" = "e1",
                              "F" = "f1, f2", "G" = "g1"))
```

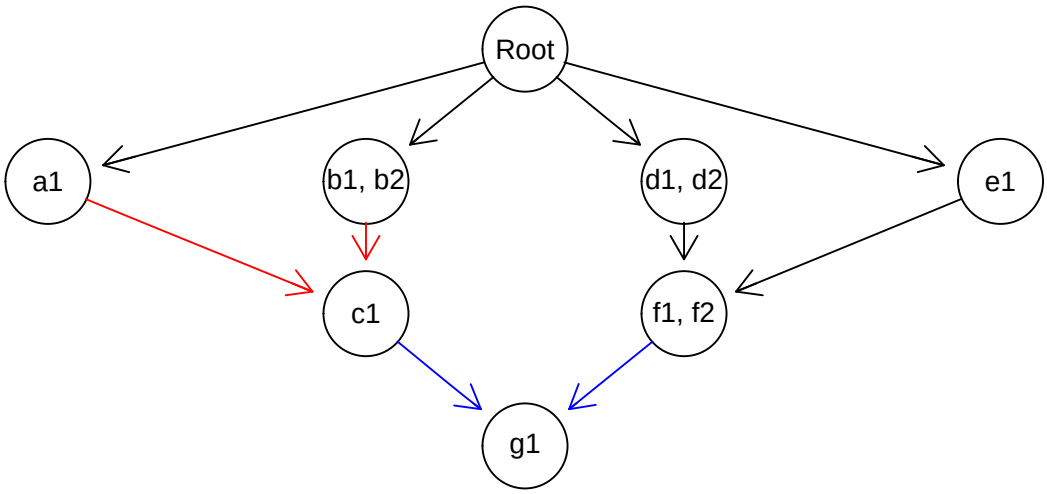
By default, plotting shows the modules:

```
plot(fp4m)
```



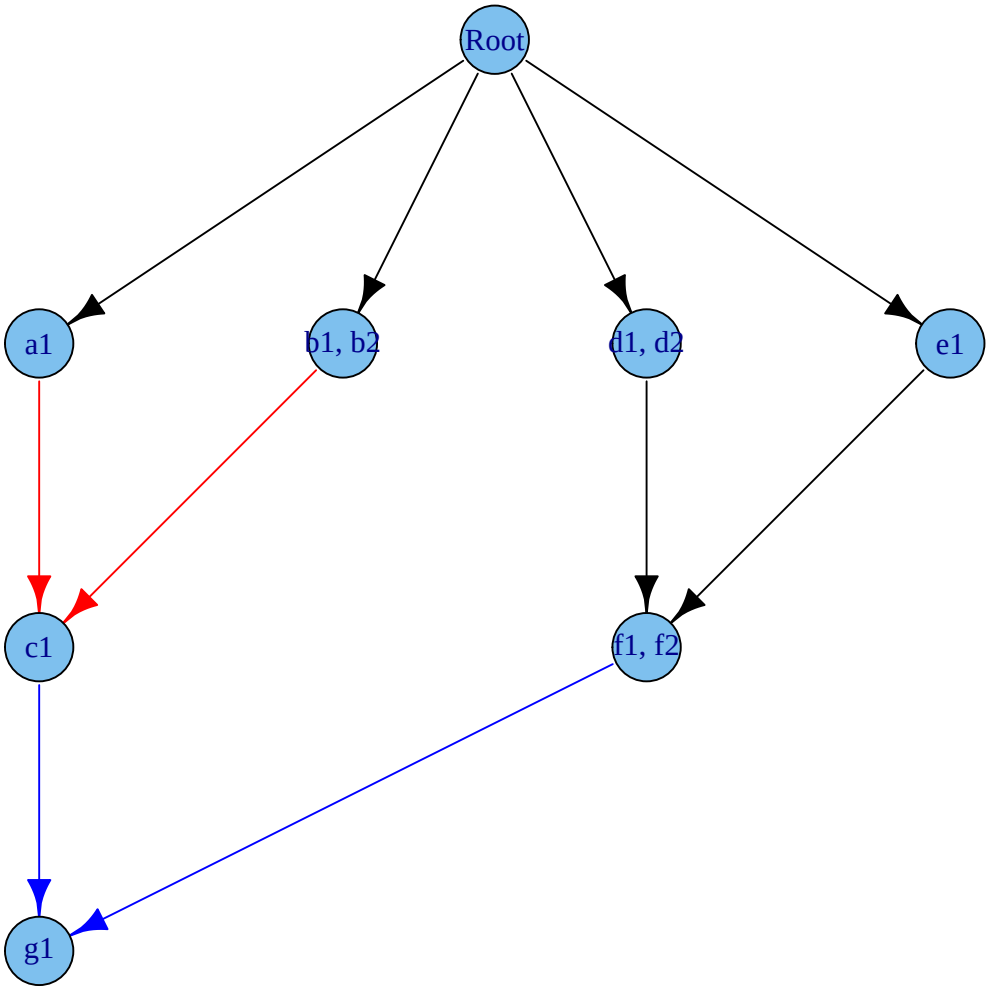
but we can show the gene names instead of the module names:

```
plot(fp4m, expandModules = TRUE)
```



or

```
plot(fp4m, "igraph", layout = layout.reingold.tilford,
     expandModules = TRUE)
```



We obtain the fitness of all genotypes in the usual way:

```
gfp4 <- evalAllGenotypes(fp4m, order = FALSE, max = 1024)
```

Let's look at a few of those:

```
gfp4[c(12, 20, 21, 40, 41, 46, 50, 55, 64, 92, 155, 157, 163, 372, 632, 828), ]
```

```
##                   Genotype  Fitness
## 12                  a1, b2 1.030200
## 20                  b1, b2 1.020000
## 21                  b1, c1 1.122000
```

```
## 40                  c1, g1 0.130000
## 41                  d1, d2 1.030000
## 46                  d2, e1 1.071200
## 50                  e1, f1 0.052000
## 55                  f2, g1 0.065000
## 64              a1, b2, c1 0.103020
## 92              b1, b2, c1 1.122000
## 155             c1, f2, g1 0.006500
## 157             d1, d2, f1 0.051500
## 163             d1, f1, f2 0.051500
## 372         d1, d2, e1, f2 1.285440
## 632     d1, d2, e1, f1, f2 1.285440
## 828 b2, c1, d1, e1, f2, g1 1.874943
```

```
c(1.01 * 1.02, 1.02, 1.02 * 1.1, 0.1 * 1.3, 1.03,
  1.03 * 1.04, 1.04 * 0.05, 0.05 * 1.3,
  1.01 * 1.02 * 0.1, 1.02 * 1.1, 0.1 * 0.05 * 1.3,
  1.03 * 0.05, 1.03 * 0.05, 1.03 * 1.04 * 1.2, 1.03 * 1.04 * 1.2,
  1.02 * 1.1 * 1.03 * 1.04 * 1.2 * 1.3)
```

```
##  [1] 1.030200 1.020000 1.122000 0.130000 1.030000 1.071200 0.052000 0.065000 0.103020
## [10] 1.122000 0.006500 0.051500 0.051500 1.285440 1.285440 1.874943
```

2.6 Order effects

As explained in the introduction (**??**), by order effects we mean a phenomenon such as the one shown empirically by [**?**]: the fitness of a double mutant "A", "B" is different depending on whether "A" was acquired before "B" or "B" before "A". This, of course, can be generalized to more than two genes.

Note that these order effects are different from the order restrictions discussed in section **??**. In there we might say that acquiring "B" depends or is facilitated by having "A" mutated (and, unless we allowed for multiple mutations, having "A" mutated means having "A" mutated before "B"). However, once you have the genotype "A, B", its fitness does not depend on the order in which "A" and "B" appeared.

2.6.1 Order effects: three-gene orders

Consider this case, where three specific three-gene orders and two two-gene orders (one of them a subset of one of the three) lead to different fitness compared to the wild-type. We add also modules, to show its usage (but just limit ourselves to using one gene per module here).

Order effects are specified using a $x > y$, that means that that order effect is satisfied when module x is mutated before module y.

```
o3 <- allFitnessEffects(orderEffects = c(
                            "F > D > M" = -0.3,
                            "D > F > M" = 0.4,
                            "D > M > F" = 0.2,
                            "D > M"     = 0.1,
                            "M > D"     = 0.5),
                        geneToModule =
                            c("Root" = "Root",
                              "M" = "m",
```

```
                                "F" = "f",
                                "D" = "d") )


(ag <- evalAllGenotypes(o3))
##     Genotype Fitness
## 1          d    1.00
## 2          f    1.00
## 3          m    1.00
## 4      d > f    1.00
## 5      d > m    1.10
## 6      f > d    1.00
## 7      f > m    1.00
## 8      m > d    1.50
## 9      m > f    1.00
## 10 d > f > m    1.54
## 11 d > m > f    1.32
## 12 f > d > m    0.77
## 13 f > m > d    1.50
## 14 m > d > f    1.50
## 15 m > f > d    1.50
```

The values for the first nine genotypes come directly from the fitness specifications. The 10th genotype matches $D > F > M$ $(= (1 + 0.4))$ but also $D > M$ $((1 + 0.1))$. The 11th matches $D > M > F$ and $D > M$. The 12th matches $F > D > M$ but also $D > M$. Etc.

2.6.2 Order effects and modules with multiple genes

Consider the following case:

```
ofe1 <- allFitnessEffects(orderEffects = c("F > D" = -0.3, "D > F" = 0.4),
                          geneToModule =
                              c("Root" = "Root",
                                "F" = "f1, f2",
                                "D" = "d1, d2") )

ag <- evalAllGenotypes(ofe1)
```

There are four genes, $d1, d2, f1, f2$, where each d belongs to module D and each f belongs to module F.

What to expect for cases such as $d1 > f1$ or $f1 > d1$ is clear, as shown in

```
ag[5:16,]
##    Genotype Fitness
## 5   d1 > d2     1.0
## 6   d1 > f1     1.4
## 7   d1 > f2     1.4
## 8   d2 > d1     1.0
## 9   d2 > f1     1.4
## 10  d2 > f2     1.4
## 11  f1 > d1     0.7
```

```
## 12  f1 > d2     0.7
## 13  f1 > f2     1.0
## 14  f2 > d1     0.7
## 15  f2 > d2     0.7
## 16  f2 > f1     1.0
```

Likewise, cases such as $d1 > d2 > f1$ or $f2 > f1 > d1$ are clear, because in terms of modules they map to $D > F$ or $F > D$: the observed order of mutation $d1 > d2 > f1$ means that module D was mutated first and module F was mutated second. Similar for $d1 > f1 > f2$ or $f1 > d1 > d2$: those map to $D > F$ and $F > D$. We can see the fitness of those four case in:

```
ag[c(17, 39, 19, 29), ]
```

```
##        Genotype Fitness
## 17 d1 > d2 > f1     1.4
## 39 f2 > f1 > d1     0.7
## 19 d1 > f1 > d2     1.4
## 29 f1 > d1 > d2     0.7
```

and they correspond to the values of those order effects, where $F > D = (1-0.3)$ and $D > F = (1+0.4)$:

```
ag[c(17, 39, 19, 29), "Fitness"] == c(1.4, 0.7, 1.4, 0.7)
```

```
## [1] TRUE TRUE TRUE TRUE
```

What if we match several patterns? For example, $d1 > f1 > d2 > f2$ and $d1 > f1 > f2 > d2$? The first maps to $D > F > D > F$ and the second to $D > F > D$. But since we are concerned with which one happened first and which happened second we should expect those two to correspond to the same fitness, that of pattern $D > F$, as is the case:

```
ag[c(43, 44),]
```

```
##             Genotype Fitness
## 43 d1 > f1 > d2 > f2     1.4
## 44 d1 > f1 > f2 > d2     1.4
```

```
ag[c(43, 44), "Fitness"] == c(1.4, 1.4)
```

```
## [1] TRUE TRUE
```

More generally, that applies to all the patterns that start with one of the "d" genes:

```
all(ag[41:52, "Fitness"] == 1.4)
```

```
## [1] TRUE
```

Similar arguments apply to the opposite pattern, $F > D$, which apply to all the possible gene mutation orders that start with one of the "f" genes. For example:

```
all(ag[53:64, "Fitness"] == 0.7)
```

```
## [1] TRUE
```

2.6.3 Order and modules with 325 genotypes

We can of course have more than two genes per module. This just repeats the above, with five genes (there are 325 genotypes, and that is why we pass the "max" argument to `evalAllGenotypes`, to allow for more than the default 256).

```
ofe2 <- allFitnessEffects(orderEffects = c("F > D" = -0.3, "D > F" = 0.4),
                          geneToModule =
                              c("Root" = "Root",
                                "F" = "f1, f2, f3",
                                "D" = "d1, d2") )
ag2 <- evalAllGenotypes(ofe2, max = 325)
```

We can verify that any combination that starts with a "d" gene and then contains at least one "f" gene will have a fitness of $1 + 0.4$. And any combination that starts with an "f" gene and contains at least one "d" genes will have a fitness of $1 - 0.3$. All other genotypes have a fitness of 1:

```
all(ag2[grep("^d.*f.*", ag2[, 1]), "Fitness"] == 1.4)
```

```
## [1] TRUE
```

```
all(ag2[grep("^f.*d.*", ag2[, 1]), "Fitness"] == 0.7)
```

```
## [1] TRUE
```

```
oe <- c(grep("^f.*d.*", ag2[, 1]), grep("^d.*f.*", ag2[, 1]))
all(ag2[-oe, "Fitness"] == 1)
```

```
## [1] TRUE
```

2.6.4 Order effects and genes without interactions

We will now look at both order effects and interactions. To make things more interesting, we name genes so that the ordered names do split nicely between those with and those without order effects (this, thus, also serves as a test of messy orders of names).

```
foi1 <- allFitnessEffects(
    orderEffects = c("D>B" = -0.2, "B > D" = 0.3),
    noIntGenes = c("A" = 0.05, "C" = -.2, "E" = .1))
```

You can get a verbose view of what the gene names and modules are (and their automatically created numeric codes) by:

```
foi1[c("geneModule", "long.geneNoInt")]
```

```
## $geneModule
##   Gene Module GeneNumID ModuleNumID
## 1 Root   Root         0           0
## 2    B      B         1           1
## 3    D      D         2           2
##
## $long.geneNoInt
##   Gene GeneNumID     s
## A    A         3  0.05
## C    C         4 -0.20
## E    E         5  0.10
```

We can get the fitness of all genotypes (we set $max = 325$ because that is the number of possible genotypes):

```
agoi1 <- evalAllGenotypes(foi1,  max = 325)
head(agoi1)
```

```
##   Genotype Fitness
## 1        B    1.00
## 2        D    1.00
## 3        A    1.05
## 4        C    0.80
## 5        E    1.10
## 6    B > D    1.30
```

Now:

```
rn <- 1:nrow(agoi1)
names(rn) <- agoi1[, 1]

agoi1[rn[LETTERS[1:5]], "Fitness"] == c(1.05, 1, 0.8, 1, 1.1)
```

```
## [1] TRUE TRUE TRUE TRUE TRUE
```

According to the fitness effects we have specified, we also know that any genotype with only two mutations, one of which is either "A", "C" or "E" and the other is "B" or "D" will have the fitness corresponding to "A", "C" or "E", respectively:

```
agoi1[grep("^A > [BD]$", names(rn)), "Fitness"] == 1.05
```

```
## [1] TRUE TRUE
```

```
agoi1[grep("^C > [BD]$", names(rn)), "Fitness"] == 0.8
```

```
## [1] TRUE TRUE
```

```
agoi1[grep("^E > [BD]$", names(rn)), "Fitness"] == 1.1
```

```
## [1] TRUE TRUE
```

```
agoi1[grep("^[BD] > A$", names(rn)), "Fitness"] == 1.05
```

```
## [1] TRUE TRUE
```

```
agoi1[grep("^[BD] > C$", names(rn)), "Fitness"] == 0.8
```

```
## [1] TRUE TRUE
```

```
agoi1[grep("^[BD] > E$", names(rn)), "Fitness"] == 1.1
```

```
## [1] TRUE TRUE
```

We will not be playing many additional games with regular expressions, but let us check those that start with "D" and have all the other mutations, which occupy rows 230 to 253; fitness should be equal (within numerical error, because of floating point arithmetic) to the order effect of having "D" before "B" times the other effects $(1 - 0.3) * 1.05 * 0.8 * 1.1 = 0.7392$

```
all.equal(agoi1[230:253, "Fitness"] , rep((1 - 0.2) * 1.05 * 0.8 * 1.1, 24))
```

```
## [1] TRUE
```

and that will also be the value of any genotype with the five mutations where "D" comes before "B" such as those in rows 260 to 265, 277, or 322 and 323, but it will be equal to $(1+0.3)*1.05*0.8*1.1 = 1.2012$ in those where "B" comes before "D". Analogous arguments apply to four, three, and two mutation genotypes.

2.7 Synthetic viability

Synthetic viability and synthetic lethality (e.g., [**?**, **?**]) are just special cases of epistasis (section **??**) but we deal with them here separately.

2.7.1 A simple synthetic viability example

A simple and extreme example of synthetic viability is shown in the following table, where the joint mutant has fitness larger than the wild type, but each single mutant is lethal.

A	B	Fitness
wt	wt	1
wt	M	0
M	wt	0
M	M	$(1 + s)$

where "wt" denotes wild type and "M" denotes mutant.

We can specify this (setting $s = 0.2$) as (I play around with spaces, to show there is a certain flexibility with them):

```
s <- 0.2
sv <- allFitnessEffects(epistasis = c("-A : B" = -1,
                                      "A : -B" = -1,
                                      "A:B" = s))
```

Now, let's look at all the genotypes (we use "addwt" to also get the wt, which by decree has fitness of 1), and disregard order:

```
(asv <- evalAllGenotypes(sv, order = FALSE, addwt = TRUE))
```

```
##   Genotype Fitness
## 1       wt     1.0
## 2        A     0.0
## 3        B     0.0
## 4     A, B     1.2
```

Asking the program to consider the order of mutations of course makes no difference:

```
evalAllGenotypes(sv, order = TRUE, addwt = TRUE)
```

```
##   Genotype Fitness
## 1       wt     1.0
## 2        A     0.0
## 3        B     0.0
## 4    A > B     1.2
## 5    B > A     1.2
```

Another example of synthetic viability is shown in section **??**.

Of course, if multiple simultaneous mutations are not possible in the simulations, it is not possible to go from the wildtype to the double mutant in this model where the single mutants are not viable.

2.7.2 Synthetic viability using Bozic model

If we were to use the above specification with Bozic's models, we might not get what we think we should get:

```
evalAllGenotypes(sv, order = FALSE, addwt = TRUE, model = "Bozic")
```

```
##   Genotype Death_rate
## 1       wt        1.0
## 2        A        2.0
## 3        B        2.0
## 4     A, B        0.8
```

What gives here? The simulation code would alert you of this (see section **??**) in this particular case because there are "-1", which might indicate that this is not what you want. The problem is that you probably want the Death rate to be infinity (the birth rate was 0, so no clone viability, when we used birth rates —section **??**).

Let us say so explicitly:

```
s <- 0.2
svB <- allFitnessEffects(epistasis = c("-A : B" = -Inf,
                                       "A : -B" = -Inf,
                                       "A:B" = s))
evalAllGenotypes(svB, order = FALSE, addwt = TRUE, model = "Bozic")
```

```
##   Genotype Death_rate
## 1       wt        1.0
## 2        A        Inf
## 3        B        Inf
## 4     A, B        0.8
```

Likewise, values of s larger than one have no effect beyond setting $s = 1$ (a single term of $(1 - 1)$ will drive the product to 0, and as we cannot allow negative death rates negative values are set to 0):

```
s <- 1
svB1 <- allFitnessEffects(epistasis = c("-A : B" = -Inf,
                                        "A : -B" = -Inf,
                                        "A:B" = s))

evalAllGenotypes(svB1, order = FALSE, addwt = TRUE, model = "Bozic")
```

```
##   Genotype Death_rate
## 1       wt          1
## 2        A        Inf
## 3        B        Inf
## 4     A, B          0
```

```
s <- 3
svB3 <- allFitnessEffects(epistasis = c("-A : B" = -Inf,
                                        "A : -B" = -Inf,
                                        "A:B" = s))

evalAllGenotypes(svB3, order = FALSE, addwt = TRUE, model = "Bozic")
```

```
##   Genotype Death_rate
## 1       wt          1
```

```
## 2         A         Inf
## 3         B         Inf
## 4      A, B           0
```

Of course, death rates of 0.0 are likely to lead to trouble down the road, when we actually conduct simulations (see section **??**).

2.7.3 Synthetic viability, non-zero fitness, and modules

This is a slightly more elaborate case, where there is one module and the single mutants have different fitness between themselves, which is non-zero. Without the modules, this is the same as in Misra et al. [**?**], Figure 1b, which we go over in section **??**.

A	B	Fitness
wt	wt	1
wt	M	$1 + s_b$
M	wt	$1 + s_a$
M	M	$1 + s_{ab}$

where $s_a, s_b < 0$ but $s_{ab} > 0$.

```
sa <- -0.1
sb <- -0.2
sab <- 0.25
sv2 <- allFitnessEffects(epistasis = c("-A : B" = sb,
                                       "A : -B" = sa,
                                       "A:B" = sab),
                         geneToModule = c(
                                       "Root" = "Root",
                                       "A" = "a1, a2",
                                       "B" = "b"))
evalAllGenotypes(sv2, order = FALSE, addwt = TRUE)
```

```
##    Genotype Fitness
## 1        wt    1.00
## 2        a1    0.90
## 3        a2    0.90
## 4         b    0.80
## 5    a1, a2    0.90
## 6     a1, b    1.25
## 7     a2, b    1.25
## 8 a1, a2, b    1.25
```

And if we look at order, of course it makes no difference:

```
evalAllGenotypes(sv2, order = TRUE, addwt = TRUE)
```

```
##       Genotype Fitness
## 1           wt    1.00
## 2           a1    0.90
## 3           a2    0.90
## 4            b    0.80
## 5      a1 > a2    0.90
## 6       a1 > b    1.25
```

```
## 7       a2 > a1    0.90
## 8        a2 > b    1.25
## 9        b > a1    1.25
## 10       b > a2    1.25
## 11 a1 > a2 > b    1.25
## 12 a1 > b > a2    1.25
## 13 a2 > a1 > b    1.25
## 14 a2 > b > a1    1.25
## 15 b > a1 > a2    1.25
## 16 b > a2 > a1    1.25
```

2.8 Synthetic mortality or synthetic lethality

In contrast to section **??**, here the joint mutant has decreased viability:

A	B	Fitness
wt	wt	1
wt	M	$1 + s_b$
M	wt	$1 + s_a$
M	M	$1 + s_{ab}$

where $s_a, s_b > 0$ but $s_{ab} < 0$.

```
sa <- 0.1
sb <- 0.2
sab <- -0.8
sm1 <- allFitnessEffects(epistasis = c("-A : B" = sb,
                                       "A : -B" = sa,
                                       "A:B" = sab))
evalAllGenotypes(sm1, order = FALSE, addwt = TRUE)
```

```
##   Genotype Fitness
## 1       wt     1.0
## 2        A     1.1
## 3        B     1.2
## 4     A, B     0.2
```

And if we look at order, of course it makes no difference:

```
evalAllGenotypes(sm1, order = TRUE, addwt = TRUE)
```

```
##   Genotype Fitness
## 1       wt     1.0
## 2        A     1.1
## 3        B     1.2
## 4    A > B     0.2
## 5    B > A     0.2
```

2.9 Epistasis

2.9.1 Epistasis: two alternative specifications

We want the following mapping of genotypes to fitness:

A	B	Fitness
wt	wt	1
wt	M	$1 + s_b$
M	wt	$1 + s_a$
M	M	$1 + s_{ab}$

Suppose that the actual numerical values are $s_a = 0.2, s_b = 0.3, s_{ab} = 0.7$.

We specify the above as follows:

```
sa <- 0.2
sb <- 0.3
sab <- 0.7

e2 <- allFitnessEffects(epistasis =
                            c("A: -B" = sa,
                              "-A:B" = sb,
                              "A : B" = sab))
evalAllGenotypes(e2, order = FALSE, addwt = TRUE)
```

```
##   Genotype Fitness
## 1       wt     1.0
## 2        A     1.2
## 3        B     1.3
## 4     A, B     1.7
```

That uses the "-" specification, so we explicitly exclude some patterns: with "A:-B" we say "A when there is no B".

But we can also use a specification where we do not use the "-". That requires a different numerical value of the interaction, because now, as we are rewriting the interaction term as genotype "A is mutant, B is mutant" the double mutant will incorporate the effects of "A mutant", "B mutant" and "both A and B mutants". We can define a new s_2 that satisfies $(1 + s_{ab}) = (1 + s_a)(1 + s_b)(1 + s_2)$ so $(1 + s_2) = (1 + s_{ab})/((1 + s_a)(1 + s_b))$ and therefore specify as:

```
s2 <- ((1 + sab)/((1 + sa) * (1 + sb))) - 1

e3 <- allFitnessEffects(epistasis =
                            c("A" = sa,
                              "B" = sb,
                              "A : B" = s2))
evalAllGenotypes(e3, order = FALSE, addwt = TRUE)
```

```
##   Genotype Fitness
## 1       wt     1.0
## 2        A     1.2
## 3        B     1.3
## 4     A, B     1.7
```

Note that this is the way you would specify effects with FFPopsim [**?**]. Whether this specification or the previous one with "-" is simpler will depend on the model. For synthetic mortality and viability, I think the one using "-" is simpler to map genotype tables to fitness effects. See also section **??** and **??** and the example in section **??**.

Finally, note that we can also specify some of these effects by combining the graph and the epistasis, as shown in section **??** or **??**.

2.9.2 Epistasis with three genes and two alternative specifications

Suppose we have

A	B	C	Fitness
M	wt	wt	$1 + s_a$
wt	M	wt	$1 + s_b$
wt	wt	M	$1 + s_c$
M	M	wt	$1 + s_{ab}$
wt	M	M	$1 + s_{bc}$
M	wt	M	$(1 + s_a)(1 + s_c)$
M	M	M	$1 + s_{abc}$

where missing rows have a fitness of 1 (they have been deleted for conciseness). Note that the mutant for exactly A and C has a fitness that is the product of the individual terms (so there is no epistasis in that case).

```
sa <- 0.1
sb <- 0.15
sc <- 0.2
sab <- 0.3
sbc <- -0.25
sabc <- 0.4

sac <- (1 + sa) * (1 + sc) - 1

E3A <- allFitnessEffects(epistasis =
                           c("A:-B:-C" = sa,
                             "-A:B:-C" = sb,
                             "-A:-B:C" = sc,
                             "A:B:-C" = sab,
                             "-A:B:C" = sbc,
                             "A:-B:C" = sac,
                             "A : B : C" = sabc)
                                         )

evalAllGenotypes(E3A, order = FALSE, addwt = FALSE)
```

```
##   Genotype Fitness
## 1        A    1.10
## 2        B    1.15
## 3        C    1.20
## 4     A, B    1.30
## 5     A, C    1.32
## 6     B, C    0.75
## 7  A, B, C    1.40
```

We needed to pass the s_{ac} coefficient explicitly, even if it that term was just the product. We can try to avoid using the "-", however (but we will need to do other calculations). For simplicity, I use capital "S" in what follows where the letters differ from the previous specification:

```
sa <- 0.1
sb <- 0.15
sc <- 0.2
```

```
sab <- 0.3
Sab <- ( (1 + sab)/((1 + sa) * (1 + sb))) - 1
Sbc <- ( (1 + sbc)/((1 + sb) * (1 + sc))) - 1
Sabc <- ( (1 + sabc)/( (1 + sa) * (1 + sb) * (1 + sc) * (1 + Sab) * (1 + Sbc) ) ) - 1

E3B <- allFitnessEffects(epistasis =
                             c("A" = sa,
                               "B" = sb,
                               "C" = sc,
                               "A:B" = Sab,
                               "B:C" = Sbc,
                               ## "A:C" = sac, ## not needed now
                               "A : B : C" = Sabc)
                                             )
evalAllGenotypes(E3B, order = FALSE, addwt = FALSE)
```

```
##   Genotype Fitness
## 1        A    1.10
## 2        B    1.15
## 3        C    1.20
## 4     A, B    1.30
## 5     A, C    1.32
## 6     B, C    0.75
## 7  A, B, C    1.40
```

The above two are, of course, identical:

```
all(evalAllGenotypes(E3A, order = FALSE, addwt = FALSE) ==
    evalAllGenotypes(E3B, order = FALSE, addwt = FALSE))
```

```
## [1] TRUE
```

We avoid specifying the "A:C", as it just follows from the individual "A" and "C" terms, but given a specified genotype table, we need to do a little bit of addition and multiplication to get the coefficients.

2.9.3 Why can we specify some effects with a "-"?

Let's suppose we want to specify the synthetic viability example seen before:

A	B	Fitness
wt	wt	1
wt	M	0
M	wt	0
M	M	$(1 + s)$

where "wt" denotes wild type and "M" denotes mutant.

If you want to directly map the above table to the fitness table for the program, to specify the genotype "A is wt, B is a mutant" you can specify it as ``-A,B'', not just as ``B''. Why? Because just the presence of a "B" is also compatible with genotype "A is mutant and B is mutant". If you use "-" you are explicitly saying what should not be there so that -A,B is NOT compatible with A, B. Otherwise, you need to carefully add coefficients. Depending on what you are trying to model, different specifications might be simpler. See the examples in section **??** and **??**. You have both options.

2.9.4 Epistasis: modules

There is nothing conceptually new, but we will show an example here:

```
sa <- 0.2
sb <- 0.3
sab <- 0.7

em <- allFitnessEffects(epistasis =
                            c("A: -B" = sa,
                              "-A:B" = sb,
                              "A : B" = sab),
                        geneToModule = c("Root" = "Root",
                                         "A" = "a1, a2",
                                         "B" = "b1, b2"))
evalAllGenotypes(em, order = FALSE, addwt = TRUE)
```

```
##          Genotype Fitness
## 1              wt     1.0
## 2              a1     1.2
## 3              a2     1.2
## 4              b1     1.3
## 5              b2     1.3
## 6           a1, a2     1.2
## 7           a1, b1     1.7
## 8           a1, b2     1.7
## 9           a2, b1     1.7
## 10          a2, b2     1.7
## 11          b1, b2     1.3
## 12      a1, a2, b1     1.7
## 13      a1, a2, b2     1.7
## 14      a1, b1, b2     1.7
## 15      a2, b1, b2     1.7
## 16 a1, a2, b1, b2     1.7
```

Of course, we can do the same thing without using the "-", as in section **??**:

```
s2 <- ((1 + sab)/((1 + sa) * (1 + sb))) - 1

em2 <- allFitnessEffects(epistasis =
                             c("A" = sa,
                               "B" = sb,
                               "A : B" = s2),
                         geneToModule = c("Root" = "Root",
                                          "A" = "a1, a2",
                                          "B" = "b1, b2")
                         )
evalAllGenotypes(em2, order = FALSE, addwt = TRUE)
```

```
##          Genotype Fitness
## 1              wt     1.0
## 2              a1     1.2
## 3              a2     1.2
## 4              b1     1.3
```

```
## 5              b2     1.3
## 6          a1, a2     1.2
## 7          a1, b1     1.7
## 8          a1, b2     1.7
## 9          a2, b1     1.7
## 10         a2, b2     1.7
## 11         b1, b2     1.3
## 12     a1, a2, b1     1.7
## 13     a1, a2, b2     1.7
## 14     a1, b1, b2     1.7
## 15     a2, b1, b2     1.7
## 16 a1, a2, b1, b2     1.7
```

2.10 Poset, epistasis, synthetic mortality and viability, order effects and genes without interactions, with some modules

We will now put together a complex example. We will use the poset from section **??** but will also add:

- Order effects that involve genes in the poset. In this case, if C happens before F, fitness decreases by $1 - 0.1$. If it happens the other way around, there is no effect on fitness beyond their individual contributions.
- Order effects that involve two new modules, "H" and "I" (with genes "h1, h2" and "i1", respectively), so that if H happens before I fitness increases by $1 + 0.12$.
- Synthetic mortality between modules "I" (already present in the epistatic interaction) and "J" (with genes "j1" and "j2"): the joint presence of these modules leads to cell death (fitness of 0).
- Synthetic viability between modules "K" and "M" (with genes "k1", "k2" and "m1", respectively), so that their joint presence is viable but adds nothing to fitness (i.e., mutation of both has fitness 1), whereas each single mutant has a fitness of $1 - 0.5$.
- A set of 5 driver genes $(n1, \ldots, n5)$ with fitness that comes from an exponential distribution with rate of 10.

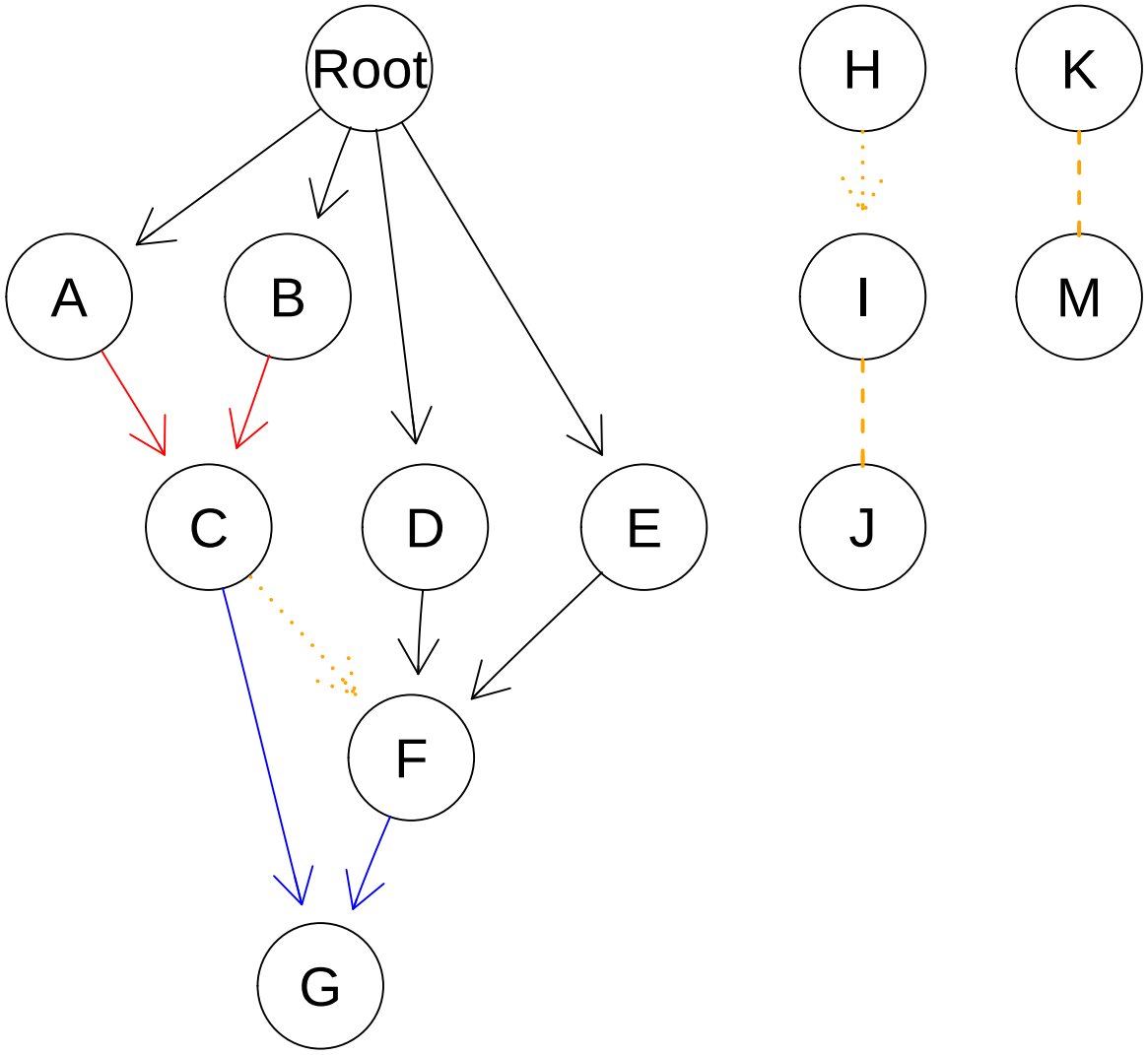
As we are specifying many different things, we will start by writing each set of effects separately:

```
p4 <- data.frame(parent = c(rep("Root", 4), "A", "B", "D", "E", "C", "F"),
                 child = c("A", "B", "D", "E", "C", "C", "F", "F", "G", "G"),
                 s = c(0.01, 0.02, 0.03, 0.04, 0.1, 0.1, 0.2, 0.2, 0.3, 0.3),
                 sh = c(rep(0, 4), c(-.9, -.9), c(-.95, -.95), c(-.99, -.99)),
                 typeDep = c(rep("--", 4),
                     "XMPN", "XMPN", "MN", "MN", "SM", "SM"))

oe <- c("C > F" = -0.1, "H > I" = 0.12)
sm <- c("I:J"  = -1)
sv <- c("-K:M" = -.5, "K:-M" = -.5)
epist <- c(sm, sv)

modules <- c("Root" = "Root", "A" = "a1",
             "B" = "b1, b2", "C" = "c1",
             "D" = "d1, d2", "E" = "e1",
             "F" = "f1, f2", "G" = "g1",
             "H" = "h1, h2", "I" = "i1",
             "J" = "j1, j2", "K" = "k1, k2", "M" = "m1")
```

```
set.seed(1) ## for repeatability
noint <- rexp(5, 10)
names(noint) <- paste0("n", 1:5)

fea <- allFitnessEffects(rT = p4, epistasis = epist, orderEffects = oe,
                         noIntGenes = noint, geneToModule = modules)
```

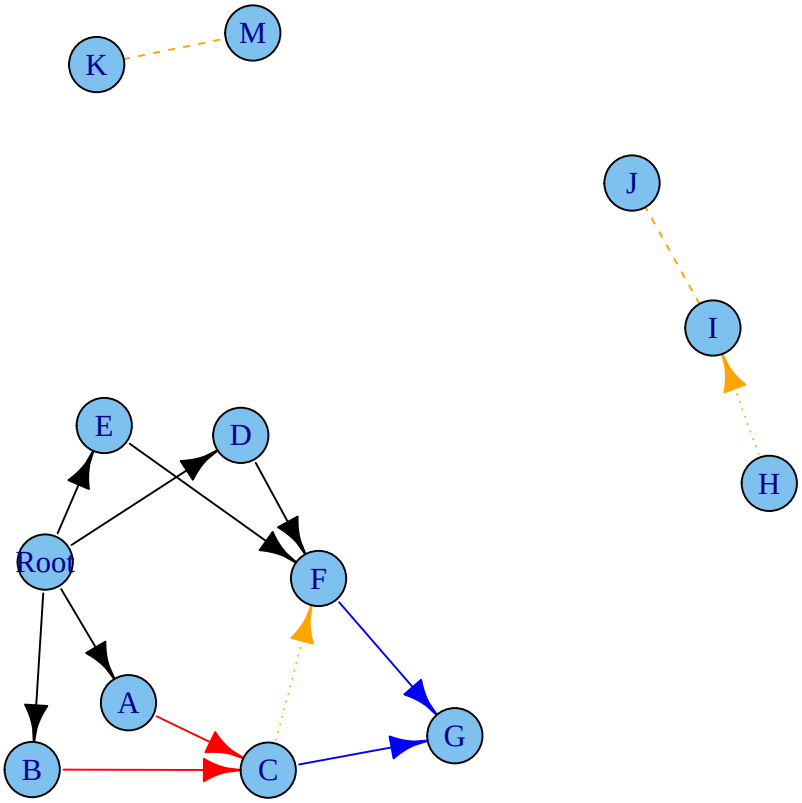
How does it look?

```
plot(fea)
```



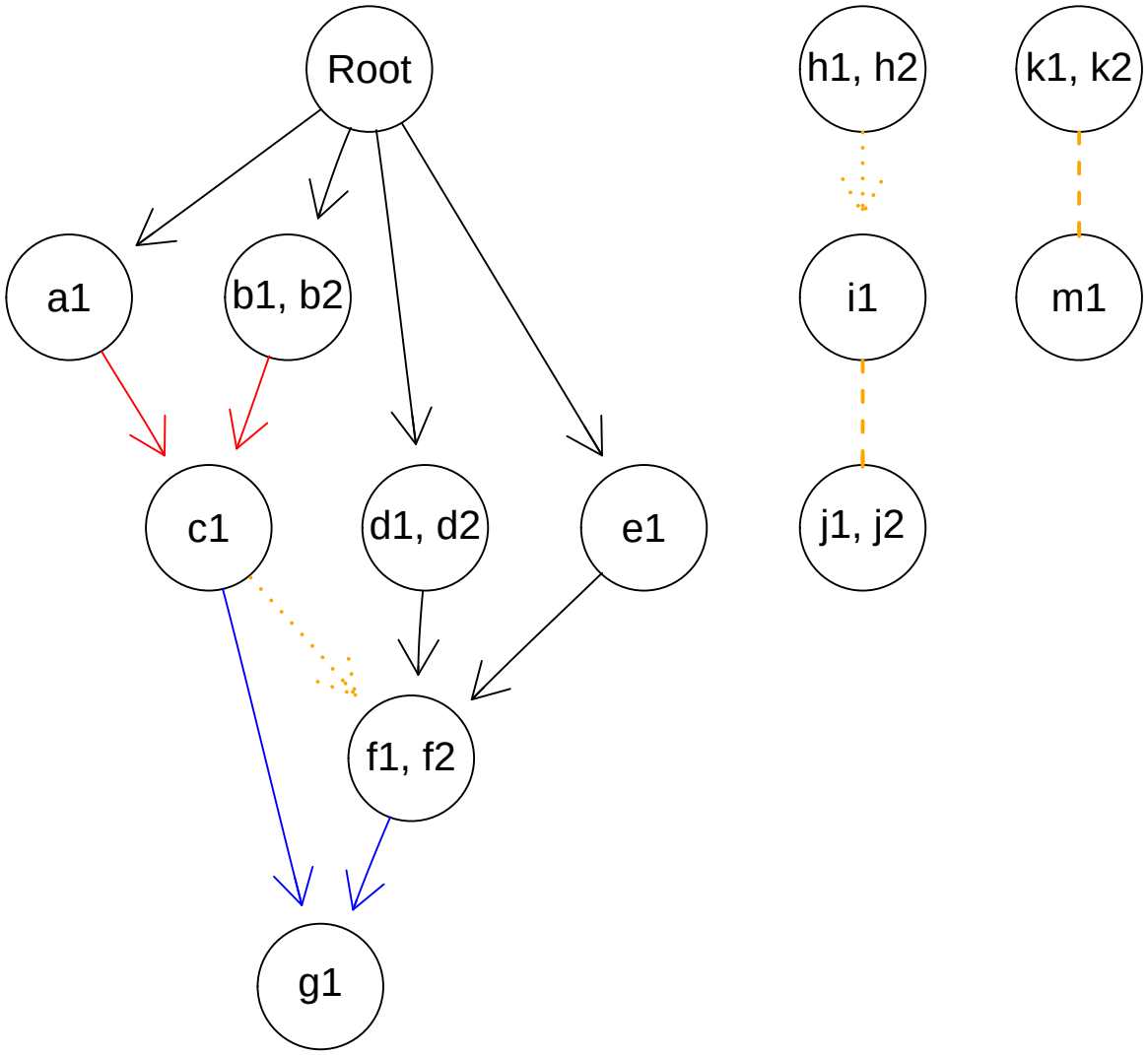
or

```
plot(fea, "igraph")
```



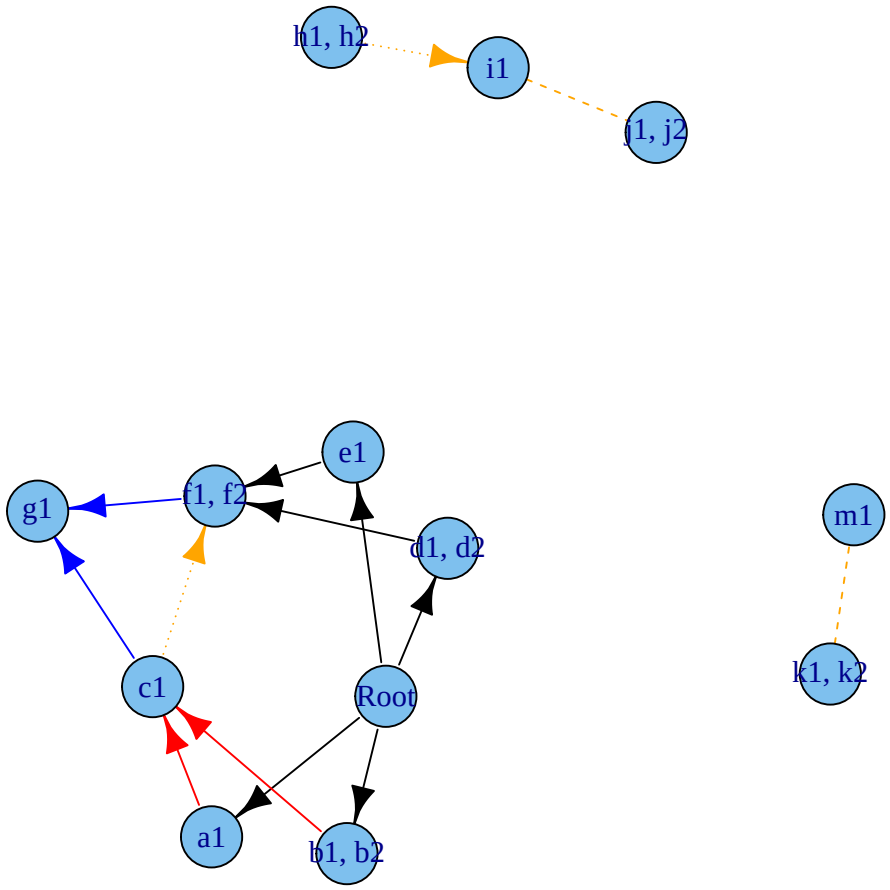
We can, if we want, expand the modules using a "graphNEL" graph

```
plot(fea, expandModules = TRUE)
```



or an "igraph" one

```
plot(fea, "igraph", expandModules = TRUE)
```



We will not evaluate the fitness of all genotypes, since the number of all ordered genotypes is $> 7 * 10^{22}$. We will look at some specific genotypes:

```
evalGenotype("k1 > i1 > h2", fea) ## 0.5
## [1] 0.5
evalGenotype("k1 > h1 > i1", fea) ## 0.5 * 1.12
## [1] 0.56
evalGenotype("k2 > m1 > h1 > i1", fea) ## 1.12
## [1] 1.12
evalGenotype("k2 > m1 > h1 > i1 > c1 > n3 > f2", fea)
## [1] 0.005113436
## 1.12 * 0.1 * (1 + noint[3]) * 0.05 * 0.9
```

Finally, let's generate some ordered genotypes randomly:

```
randomGenotype <- function(fe, ns = NULL) {
    gn <- setdiff(c(fe$geneModule$Gene,
                    fe$long.geneNoInt$Gene), "Root")
    if(is.null(ns)) ns <- sample(length(gn), 1)
    return(paste(sample(gn, ns), collapse = " > "))
}

set.seed(2) ## for reproducibility

evalGenotype(randomGenotype(fea), fea, echo = TRUE, verbose = TRUE)
```

```
## Genotype:  k2 > i1 > c1 > n1 > m1
##  Individual s terms are : 0.0755182 -0.9
##  Fitness:  0.1075518
## [1] 0.1075518
```

```
## Genotype:  k2 > i1 > c1 > n1 > m1
##  Individual s terms are : 0.0755182 -0.9
##  Fitness:  0.107552
evalGenotype(randomGenotype(fea), fea, echo = TRUE, verbose = TRUE)
```

```
## Genotype:  n2 > h1 > h2
##  Individual s terms are : 0.118164
##  Fitness:  1.118164
## [1] 1.118164
```

```
## Genotype:  n2 > h1 > h2
##  Individual s terms are : 0.118164
##  Fitness:  1.11816
evalGenotype(randomGenotype(fea), fea, echo = TRUE, verbose = TRUE)
```

```
## Genotype:  d2 > k2 > c1 > f2 > n4 > m1 > n3 > f1 > b1 > g1 > n5 > h1 > j2
##  Individual s terms are : 0.0145707 0.0139795 0.0436069 0.02 0.1 0.03 -0.95 0.3 -0.1
##  Fitness:  0.07258291
## [1] 0.07258291
```

```
## Genotype:  d2 > k2 > c1 > f2 > n4 > m1 > n3 > f1 > b1 > g1 > n5 > h1 > j2
##  Individual s terms are : 0.0145707 0.0139795 0.0436069 0.02 0.1 0.03 -0.95 0.3 -0.1
##  Fitness:  0.0725829
evalGenotype(randomGenotype(fea), fea, echo = TRUE, verbose = TRUE)
```

```
## Genotype:  h2 > c1 > f1 > n2 > b2 > a1 > n1 > i1
##  Individual s terms are : 0.0755182 0.118164 0.01 0.02 -0.9 -0.95 -0.1 0.12
##  Fitness:  0.006244181
## [1] 0.006244181
```

```
## Genotype:  h2 > c1 > f1 > n2 > b2 > a1 > n1 > i1
##  Individual s terms are : 0.0755182 0.118164 0.01 0.02 -0.9 -0.95 -0.1 0.12
##  Fitness:  0.00624418
evalGenotype(randomGenotype(fea), fea, echo = TRUE, verbose = TRUE)
```

```
## Genotype:  h2 > j1 > m1 > d2 > i1 > b2 > k2 > d1 > b1 > n3 > n1 > g1 > h1 > c1 > k1 > e1 >
##  Individual s terms are : 0.0755182 0.0145707 0.0436069 0.01 0.02 -0.9 0.03 0.04 0.2 0.3 -1
```

```
##  Fitness:  0
## [1] 0

## Genotype:  h2 > j1 > m1 > d2 > i1 > b2 > k2 > d1 > b1 > n3 > n1 > g1 > h1 > c1 > k1 > e1 >
##  Individual s terms are : 0.0755182 0.0145707 0.0436069 0.01 0.02 -0.9 0.03 0.04 0.2 0.3 -1
##  Fitness:  0
evalGenotype(randomGenotype(fea), fea, echo = TRUE, verbose = TRUE)

## Genotype:  n1 > m1 > n3 > i1 > j1 > n5 > k1
##  Individual s terms are : 0.0755182 0.0145707 0.0436069 -1
##  Fitness:  0
## [1] 0

## Genotype:  n1 > m1 > n3 > i1 > j1 > n5 > k1
##  Individual s terms are : 0.0755182 0.0145707 0.0436069 -1
##  Fitness:  0
evalGenotype(randomGenotype(fea), fea, echo = TRUE, verbose = TRUE)

## Genotype:  d2 > n1 > g1 > f1 > f2 > c1 > b1 > d1 > k1 > a1 > b2 > i1 > n4 > h2 > n2
##  Individual s terms are : 0.0755182 0.118164 0.0139795 0.01 0.02 -0.9 0.03 -0.95 0.3 -0.5
##  Fitness:  0.004205278
## [1] 0.004205278

## Genotype:  d2 > n1 > g1 > f1 > f2 > c1 > b1 > d1 > k1 > a1 > b2 > i1 > n4 > h2 > n2
##  Individual s terms are : 0.0755182 0.118164 0.0139795 0.01 0.02 -0.9 0.03 -0.95 0.3 -0.5
##  Fitness:  0.00420528
evalGenotype(randomGenotype(fea), fea, echo = TRUE, verbose = TRUE)

## Genotype:  j1 > f1 > j2 > a1 > n4 > c1 > n3 > k1 > d1 > h1
##  Individual s terms are : 0.0145707 0.0139795 0.01 0.1 0.03 -0.95 -0.5
##  Fitness:  0.02943085
## [1] 0.02943085

## Genotype:  j1 > f1 > j2 > a1 > n4 > c1 > n3 > k1 > d1 > h1
##  Individual s terms are : 0.0145707 0.0139795 0.01 0.1 0.03 -0.95 -0.5
##  Fitness:  0.0294308
evalGenotype(randomGenotype(fea), fea, echo = TRUE, verbose = TRUE)

## Genotype:  n5 > f2 > f1 > h2 > n4 > c1 > n3 > b1
##  Individual s terms are : 0.0145707 0.0139795 0.0436069 0.02 0.1 -0.95
##  Fitness:  0.06022978
## [1] 0.06022978

## Genotype:  n5 > f2 > f1 > h2 > n4 > c1 > n3 > b1
##  Individual s terms are : 0.0145707 0.0139795 0.0436069 0.02 0.1 -0.95
##  Fitness:  0.0602298
evalGenotype(randomGenotype(fea), fea, echo = TRUE, verbose = TRUE)

## Genotype:  h1 > d1 > f2
##  Individual s terms are : 0.03 -0.95
##  Fitness:  0.0515
## [1] 0.0515

## Genotype:  h1 > d1 > f2
##  Individual s terms are : 0.03 -0.95
##  Fitness:  0.0515
```

2.11 Homozygosity, heterozygosity, oncogenes, tumor suppressors

We are using what is conceptually a single linear chromosome. However, you can use it to model scenarios where the numbers of copies affected matter, by properly duplicating the genes.

Suppose we have a tumor suppressor gene, G, with two copies, one from Mom and one from Dad. We can have a table like:

G_M	G_D	Fitness
wt	wt	1
wt	M	1
M	wt	1
M	M	$(1+s)$

where $s > 0$, meaning that you need two hits, one in each copy, to trigger the clonal expansion.

What about oncogenes? A simple model is that one single hit leads to clonal expansion and additional hits lead to no additional changes, as in this table for gene O, where again the M or D subscript denotes the copy from Mom or from Dad:

O_M	O_D	Fitness
wt	wt	1
wt	M	$(1+s)$
M	wt	$(1+s)$
M	M	$(1+s)$

If you have multiple copies you can proceed similarly. As you can see, these are nothing but special cases of synthetic mortality (**??**), synthetic viability (**??**) and epistasis (**??**).

3 Specifying fitness effects: some examples from the literature

3.1 Bauer et al

In the model of bauer and collaborators [**?**, p. 54] "For cells without the primary driver mutation, each secondary driver mutation leads to a change in the cell's fitness by s_P. For cells with the primary driver mutation, the fitness advantage obtained with each secondary driver mutation is s_{DP}."

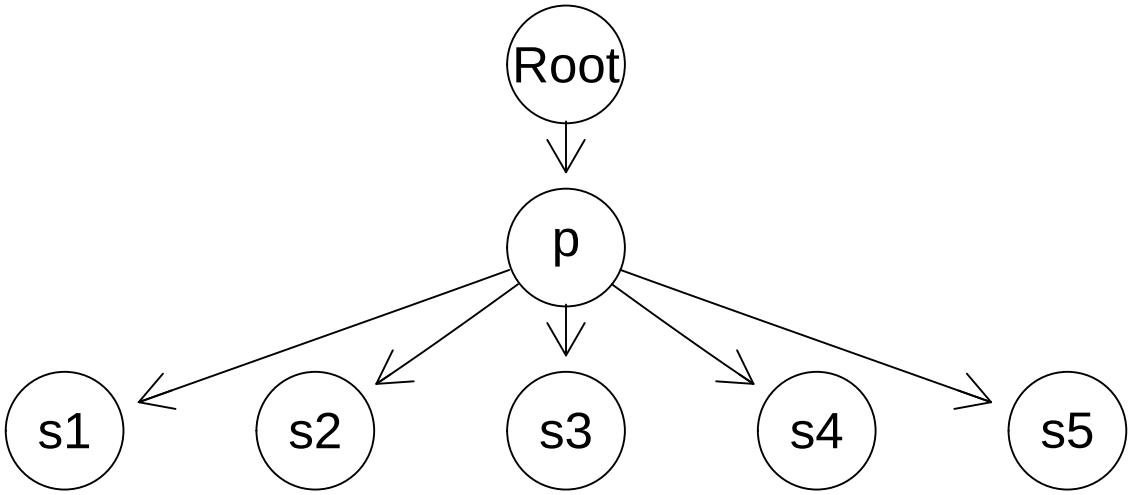
The proliferation probability is given as $(1+s_p)^k$ when there are k secondary drivers mutated and no primary diver. If the primary driver is mutated, then the expression is $\frac{1+S_D^+}{1+S_D^-}(1+S_{DP})^k$. They set apoptosis as $1 - proliferation$. So, ignoring constants such as $1/2$, and setting $P = \frac{1+S_D^+}{1+S_D^-}$ we can prepare a table as (for a largest k of 5 in this example, but can make it arbitrarily large):

```
K <- 5
sd <- 0.1
sdp <- 0.15
sp <- 0.05
bauer <- data.frame(parent = c("Root", rep("p", K)),
                    child = c("p", paste0("s", 1:K)),
                    s = c(sd, rep(sdp, K)),
                    sh = c(0, rep(sp, K)),
                    typeDep = "MN")
fbauer <- allFitnessEffects(bauer)
```

Note that what we specify as "typeDep" is irrelevant (MN, SMN, or XMPN make no difference).

The fitness effects figure looks like this:

```
plot(fbauer)
```




```
(b1 <- evalAllGenotypes(fbauer, order = FALSE))[1:10, ]
```

```
##    Genotype Fitness
## 1         p   1.100
## 2        s1   1.050
## 3        s2   1.050
## 4        s3   1.050
## 5        s4   1.050
## 6        s5   1.050
## 7     p, s1   1.265
## 8     p, s2   1.265
## 9     p, s3   1.265
## 10    p, s4   1.265
```

Order makes no difference

```
(b2 <- evalAllGenotypes(fbauer, order = TRUE, max = 2000))[1:15, ]
```

```
##    Genotype Fitness
## 1         p  1.1000
## 2        s1  1.0500
## 3        s2  1.0500
## 4        s3  1.0500
## 5        s4  1.0500
## 6        s5  1.0500
## 7    p > s1  1.2650
## 8    p > s2  1.2650
## 9    p > s3  1.2650
## 10   p > s4  1.2650
## 11   p > s5  1.2650
## 12   s1 > p  1.2650
## 13  s1 > s2  1.1025
## 14  s1 > s3  1.1025
```

```
## 15  s1 > s4  1.1025
```

And the number of levels is the right one: 11

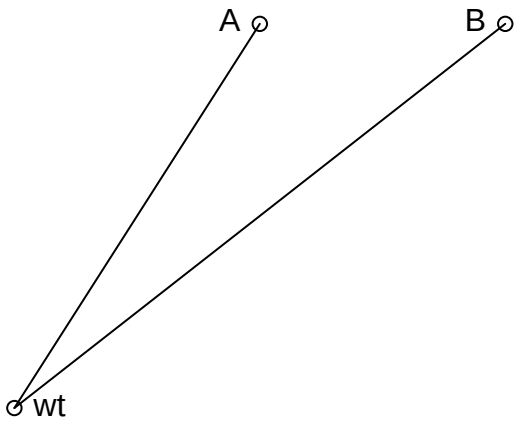
```
length(table(b1$Fitness))
## [1] 11
length(table(b2$Fitness))
## [1] 11
```

Can we use modules in this module? Sure, as in any other.

3.2 Misra et al., 2014

Figure 1 of Misra et al. [**?**] presents three scenarios which are different types of epistasis.

3.2.1 Example 1.a



In that figure it is evident that the fitness effect of "A" and "B" are the same. There are two different models depending on whether "AB" is just the product of both, or there is epistasis. In the first case probably the simplest is:

```
s <- 0.1 ## or whatever number
m1a1 <- allFitnessEffects(data.frame(parent = c("Root", "Root"),
                                     child = c("A", "B"),
                                     s = s,
                                     sh = 0,
                                     typeDep = "MN"))
evalAllGenotypes(m1a1, order = FALSE, addwt = TRUE)
```

```
##   Genotype Fitness
## 1       wt    1.00
## 2        A    1.10
## 3        B    1.10
## 4     A, B    1.21
```

If the double mutant shows epistasis, as we saw before (section **??**) we have a range of options. For example:

```
s <- 0.1
sab <- 0.3
m1a2 <- allFitnessEffects(epistasis = c("A:-B" = s,
                                        "-A:B" = s,
                                        "A:B" = sab))
evalAllGenotypes(m1a2, order = FALSE, addwt = TRUE)
```

```
##   Genotype Fitness
## 1       wt     1.0
## 2        A     1.1
## 3        B     1.1
## 4     A, B     1.3
```

But we could also modify the graph dependency structure, and we have to change the value of the coefficient, since that is what multiplies each of the terms for "A" and "B": $(1+s_{AB}) = (1+s)^2(1+s_{AB3})$

```
sab3 <- ((1 + sab)/((1 + s)^2)) - 1
m1a3 <- allFitnessEffects(data.frame(parent = c("Root", "Root"),
                                     child = c("A", "B"),
                                     s = s,
                                     sh = 0,
                                     typeDep = "MN"),
                          epistasis = c("A:B" = sab3))
evalAllGenotypes(m1a3, order = FALSE, addwt = TRUE)
```

```
##   Genotype Fitness
## 1       wt     1.0
## 2        A     1.1
## 3        B     1.1
## 4     A, B     1.3
```

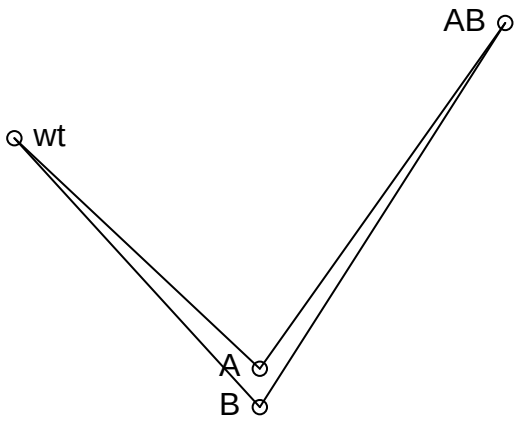
And, obviously

```
all.equal(evalAllGenotypes(m1a2, order = FALSE, addwt = TRUE),
          evalAllGenotypes(m1a3, order = FALSE, addwt = TRUE))
```

```
## [1] TRUE
```

3.2.2 Example 1.b

This is a specific case of synthetic viability (see also section **??**):



Here, $S_A, S_B < 0$, $S_B < 0$, $S_{AB} > 0$ and $(1 + S_{AB})(1 + S_A)(1 + S_B) > 1$.

As before, we can specify this in several different ways. The simplest is to specify all genotypes:

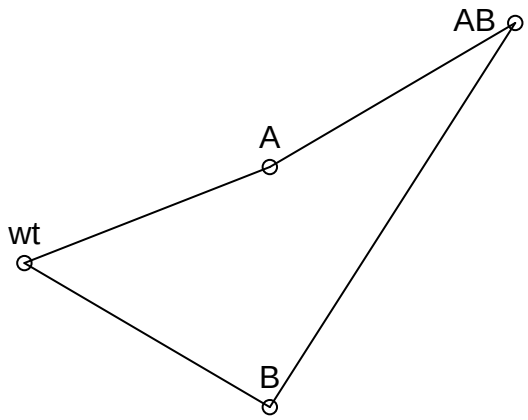
```
sa <- -0.6
sb <- -0.7
sab <- 0.3
m1b1 <- allFitnessEffects(epistasis = c("A:-B" = sa,
                                        "-A:B" = sb,
                                        "A:B" = sab))
evalAllGenotypes(m1b1, order = FALSE, addwt = TRUE)
```

```
##   Genotype Fitness
## 1       wt     1.0
## 2        A     0.4
## 3        B     0.3
## 4     A, B     1.3
```

We could also use a tree and modify the "sab" for the epistasis, as before (**??**).

3.2.3 Example 1.c

The final case, in figure 1.c of Misra et al., is just epistasis, where a mutation in one of the genes is deleterious (possibly only mildly), in the other is beneficial, and the double mutation has fitness larger than any of the other two.



Here we have that $s_A > 0$, $s_B < 0$, $(1 + s_{AB})(1 + s_A)(1 + s_B) > (1 + s_{AB})$ so $s_{AB} > \frac{-s_B}{1+s_B}$

As before, we can specify this in several different ways. The simplest is to specify all genotypes:

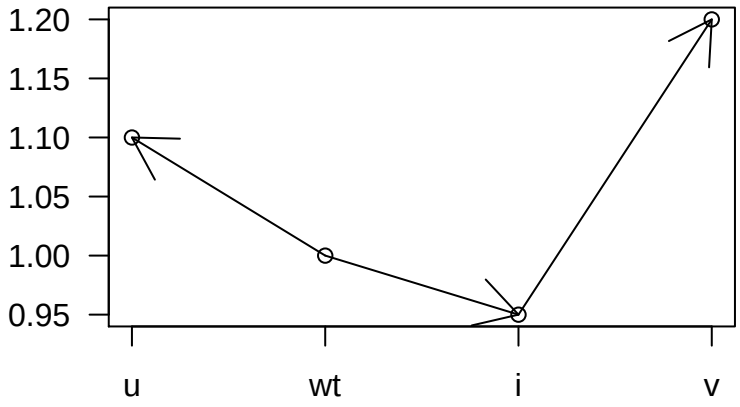
```
sa <- 0.2
sb <- -0.3
sab <- 0.5
m1c1 <- allFitnessEffects(epistasis = c("A:-B" = sa,
                                        "-A:B" = sb,
                                        "A:B" = sab))
evalAllGenotypes(m1c1, order = FALSE, addwt = TRUE)
```

```
##   Genotype Fitness
## 1       wt     1.0
## 2        A     1.2
## 3        B     0.7
## 4     A, B     1.5
```

We could also use a tree and modify the "sab" for the epistasis, as before (**??**).

3.3 Ochs and Desai, 2015

In [**?**] the authors present a model shown graphically as (the actual numerical values are arbitrarily set by me):



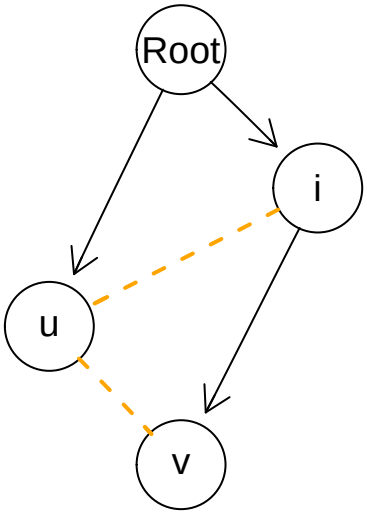
In their model, $s_u > 0$, $s_v > s_u$, $s_i < 0$, we can only arrive at v from i, and the mutants "ui" and "uv" can never appear as their fitness is 0, or $-\infty$, so $s_{ui} = s_{uv} = -1$ (or $-\infty$).

We can specify this combining a graph and epistasis specifications:

```
su <- 0.1
si <- -0.05
fvi <- 1.2 ## the fitnes of the vi mutant
sv <- (fvi/(1 + si)) - 1
sui <- suv <- -1
od <- allFitnessEffects(
    data.frame(parent = c("Root", "Root", "i"),
               child = c("u", "i", "v"),
               s = c(su, si, sv),
               sh = -1,
               typeDep = "MN"),
    epistasis = c(
        "u:i" = sui,
        "u:v" = suv))
```

A figure showing that model is

```
plot(od)
```



And the fitness of all genotype is

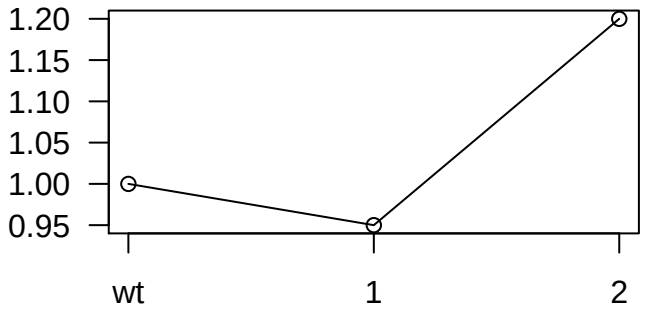
```
evalAllGenotypes(od, order = FALSE, addwt = TRUE)
```

```
##   Genotype Fitness
## 1       wt    1.00
## 2        i    0.95
## 3        u    1.10
## 4        v    0.00
## 5     i, u    0.00
## 6     i, v    1.20
## 7     u, v    0.00
## 8  i, u, v    0.00
```

3.4 Weissman et al., 2009

In their figure 1a, Weisman et al. [**?**] present this model (actual numeric values are set arbitrarily)

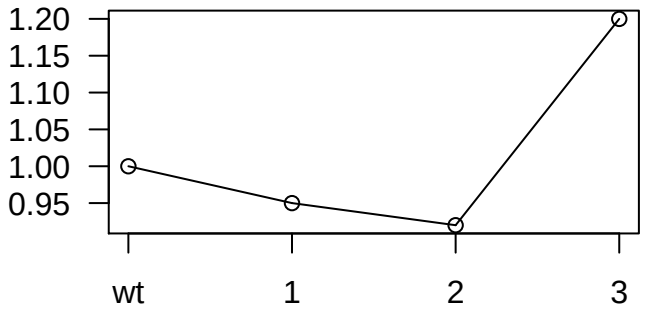
3.4.1 Figure 1.a



where the "1" and "2" refer to the total number of mutations in two different loci. This is, therefore, very similar to the example in section **??**. Here we have, in their notation, $\delta_1 < 0$, fitness of single "A" or single "B" $= 1 + \delta_1$, $S_{AB} > 0$, $(1 + S_{AB})(1 + \delta_1)^2 > 1$.

3.4.2 Figure 1.b

In their figure 1b they show



Where, as before, 1, 2, 3, denote the total number of mutations over three different loci and $\delta_1 < 0$, $\delta_2 < 0$, fitness of single mutant is $(1+\delta_1)$, of double mutant is $(1+\delta_2)$ so that $(1+\delta_2) = (1+\delta_1)^2(1+s_2)$ and of triple mutant is $(1 + \delta_3)$, so that $(1 + \delta_3) = (1 + \delta_1)^3(1 + s_2)^3(1 + s_3)$.

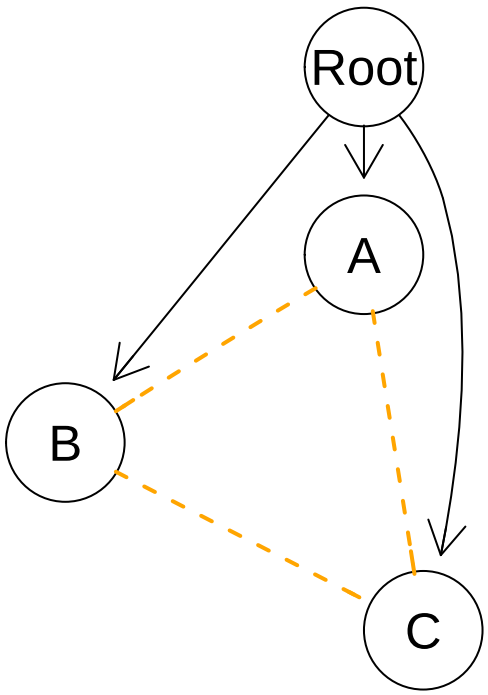
We can specify this combining a graph with epistasis:

```
d1 <- -0.05 ## single mutant fitness 0.95
d2 <- -0.08 ## double mutant fitness 0.92
d3 <- 0.2   ## triple mutant fitness 1.2

s2 <- ((1 + d2)/(1 + d1)^2) - 1
```

```
s3 <- ( (1 + d3)/((1 + d1)^3 * (1 + s2)^3) ) - 1

w <- allFitnessEffects(
    data.frame(parent = c("Root", "Root", "Root"),
               child = c("A", "B", "C"),
               s = d1,
               sh = -1,
               typeDep = "MN"),
    epistasis = c(
        "A:B" = s2,
        "A:C" = s2,
        "B:C" = s2,
        "A:B:C" = s3))
```

The model can be shown graphically as:

```
plot(w)
```



And fitness of all genotypes is:

```
evalAllGenotypes(w, order = FALSE, addwt = TRUE)
```

```
##   Genotype Fitness
## 1       wt    1.00
## 2        A    0.95
## 3        B    0.95
## 4        C    0.95
## 5     A, B    0.92
## 6     A, C    0.92
## 7     B, C    0.92
## 8  A, B, C    1.20
```

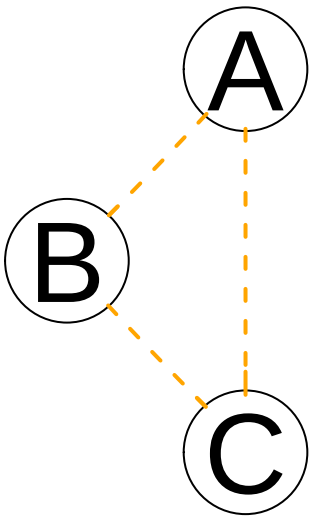
Alternatively, we can directly specify what each genotype adds to the fitness, given the included genotype. This is basically replacing the graph by giving each of "A", "B", and "C" directly:

```
wb <- allFitnessEffects(
    epistasis = c(
        "A" = d1,
        "B" = d1,
        "C" = d1,
        "A:B" = s2,
        "A:C" = s2,
        "B:C" = s2,
        "A:B:C" = s3))

evalAllGenotypes(wb, order = FALSE, addwt = TRUE)
```

```
##   Genotype Fitness
## 1       wt    1.00
## 2        A    0.95
## 3        B    0.95
## 4        C    0.95
## 5     A, B    0.92
## 6     A, C    0.92
## 7     B, C    0.92
## 8  A, B, C    1.20
```

The plot, of course, is not very revealing and we cannot show that there is a three-way interaction (only all three two-way interactions):

```
plot(wb)
```



As we have seen several times already (sections **??**, **??**, **??**) we can also give the genotypes directly and, consequently, the fitness of each genotype (not the added contribution):

```
wc <- allFitnessEffects(
    epistasis = c(
        "A:-B:-C" = d1,
        "B:-C:-A" = d1,
```

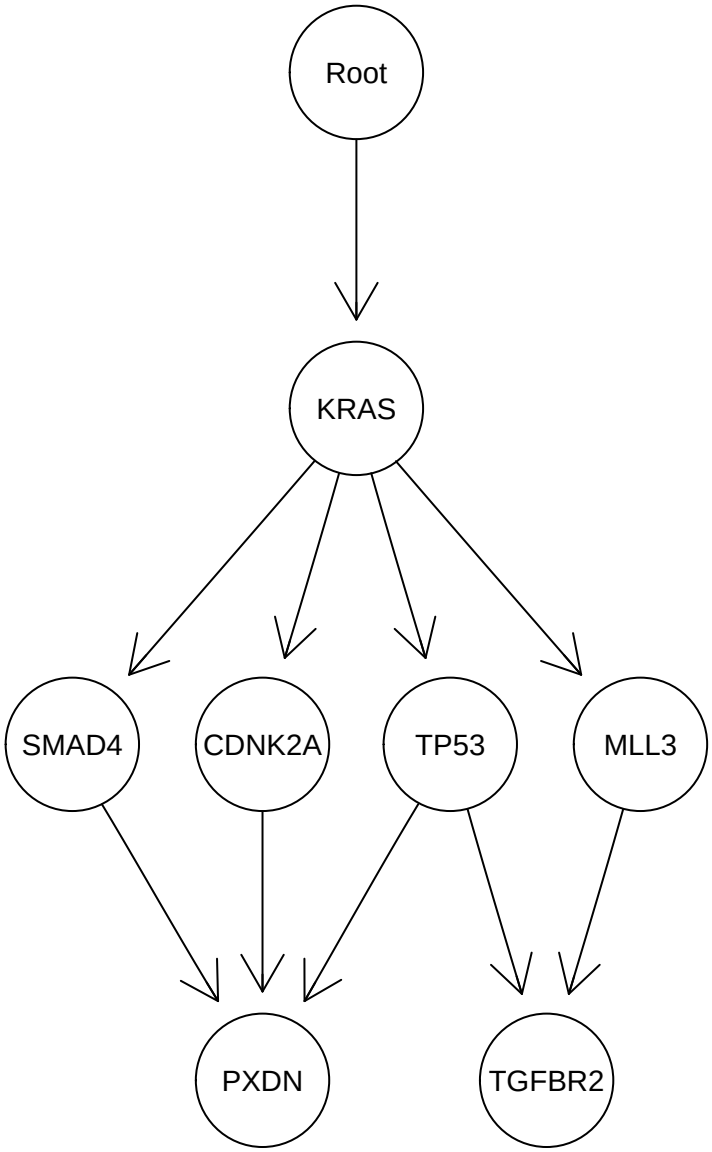
```
        "C:-A:-B" = d1,
        "A:B:-C" = d2,
        "A:C:-B" = d2,
        "B:C:-A" = d2,
        "A:B:C" = d3))
evalAllGenotypes(wc, order = FALSE, addwt = TRUE)
```

```
##   Genotype Fitness
## 1       wt    1.00
## 2        A    0.95
## 3        B    0.95
## 4        C    0.95
## 5     A, B    0.92
## 6     A, C    0.92
## 7     B, C    0.92
## 8  A, B, C    1.20
```

3.5 Gerstung et al., pancreatic cancer poset

Similar to what we did in v.1 (see section **??**) we can specify the pancreatic cancer poset in Gerstung et al. [**?**] (their figure 2B, left). We use directly the names of the genes, since that is immediately supported by the new version.

```
pancr <- allFitnessEffects(
    data.frame(parent = c("Root", rep("KRAS", 4),
                   "SMAD4", "CDNK2A",
                   "TP53", "TP53", "MLL3"),
               child = c("KRAS","SMAD4", "CDNK2A",
                   "TP53", "MLL3",
                   rep("PXDN", 3), rep("TGFBR2", 2)),
               s = 0.1,
               sh = -0.9,
               typeDep = "MN"))

plot(pancr)
```

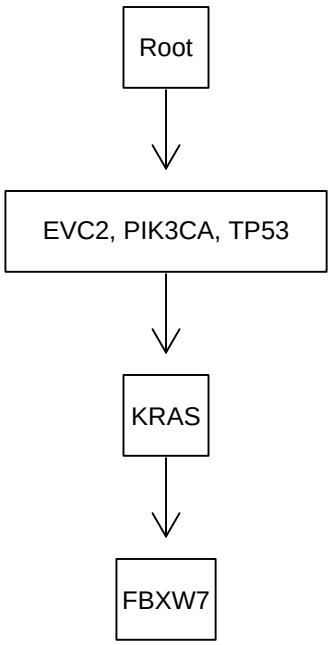


Of course the "s" and "sh" are set arbitrarily here.

3.6 Raphael and Vandin's modules

In [**?**], Raphael and Vandin show several progression models in terms of modules. We can code the extended poset for the colorectal cancer model in their Figure 4.a is (s and sh are arbitrary):

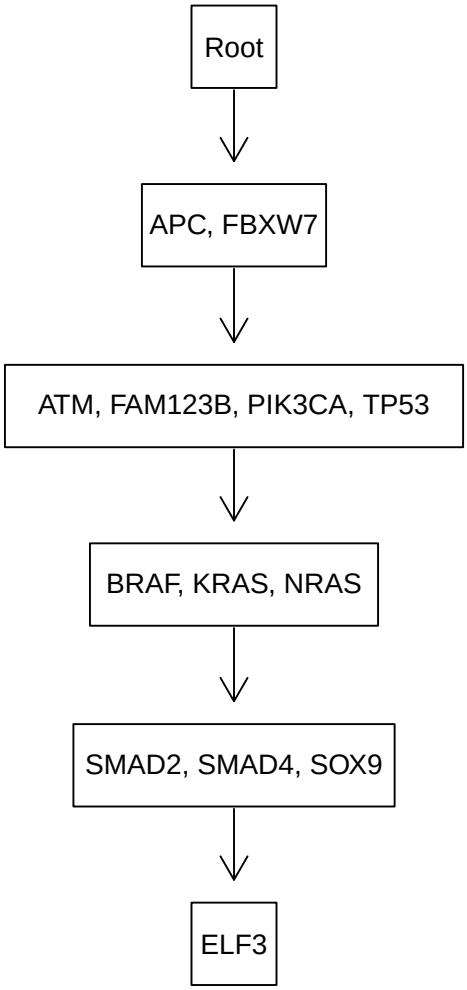
```
rv1 <- allFitnessEffects(data.frame(parent = c("Root", "A", "KRAS"),
                                    child = c("A", "KRAS", "FBXW7"),
                                    s = 0.1,
                                    sh = -0.01,
                                    typeDep = "MN"),
                         geneToModule = c("Root" = "Root",
                             "A" = "EVC2, PIK3CA, TP53",
                             "KRAS" = "KRAS",
                             "FBXW7" = "FBXW7"))

plot(rv1, expandModules = TRUE, autofit = TRUE)
```



We have used the (experimental) `autofit` option to fit the labels to the edges. Note how we can use the same name for genes and modules, but we need to specify all the modules.

Their Figure 5b is

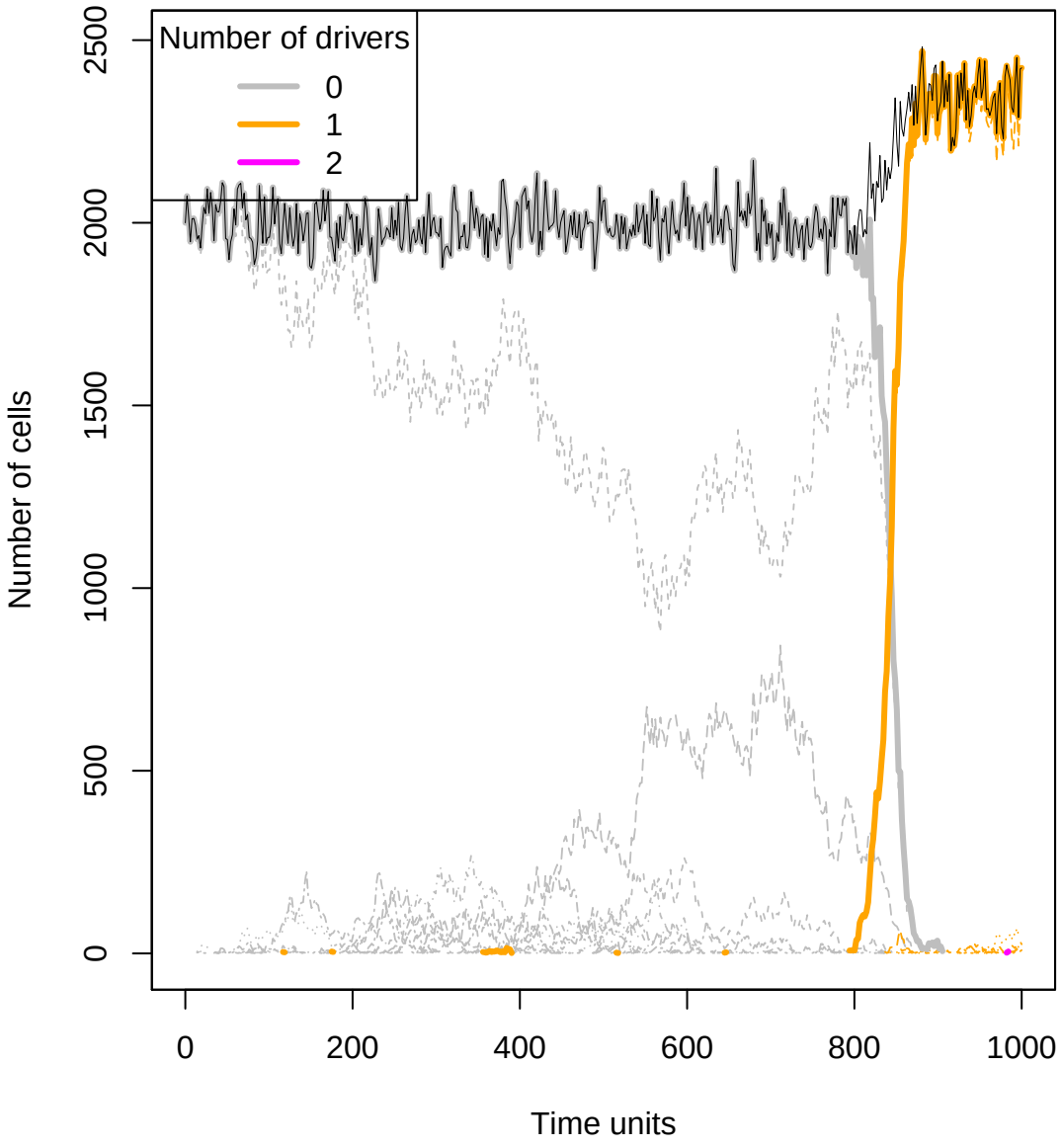
```
rv2 <- allFitnessEffects(data.frame(parent = c("Root", "1", "2", "3", "4"),
                                    child = c("1", "2", "3", "4", "ELF3"),
                                    s = 0.1,
                                    sh = -0.01,
                                    typeDep = "MN"),
                         geneToModule = c("Root" = "Root",
                             "1" = "APC, FBXW7",
                             "2" = "ATM, FAM123B, PIK3CA, TP53",
                             "3" = "BRAF, KRAS, NRAS",
                             "4" = "SMAD2, SMAD4, SOX9",
                             "ELF3" = "ELF3"))
plot(rv2, expandModules = TRUE,   autofit = TRUE)
```



4 Running the simulations

4.1 McFarland model with 5000 passengers and 70 drivers

```
set.seed(456)
nd <- 70
np <- 5000
s <- 0.1
sp <- 1e-3
spp <- -sp/(1 + sp)
mcf1 <- allFitnessEffects(noIntGenes = c(rep(s, nd), rep(spp, np)),
                          drv = seq.int(nd))
mcf1s <-  oncoSimulIndiv(mcf1,
                         model = "McFL",
                         mu = 1e-7,
                         detectionSize = 1e8,
                         detectionDrivers = 100,
                         sampleEvery = 0.02,
                         keepEvery = 2,
                         initSize = 2000,
                         finalTime = 1000,
                         onlyCancer = FALSE)
plot(mcf1s, addtot = TRUE, lwdClone = 0.9, log = "")
```




```
summary(mcf1s)
```

```
##   NumClones TotalPopSize LargestClone MaxNumDrivers MaxDriversLast NumDriversLargestPop
## 1       563         2424         2332             2              1                   1
##   TotalPresentDrivers FinalTime NumIter HittedWallTime    errorMF minDMratio minBMratio
## 1                  70      1000   51036          FALSE 0.01197637   1850.236   1966.068
##               OccurringDrivers
## 1 7, 8, 21, 22, 38, 43, 52, 68
```

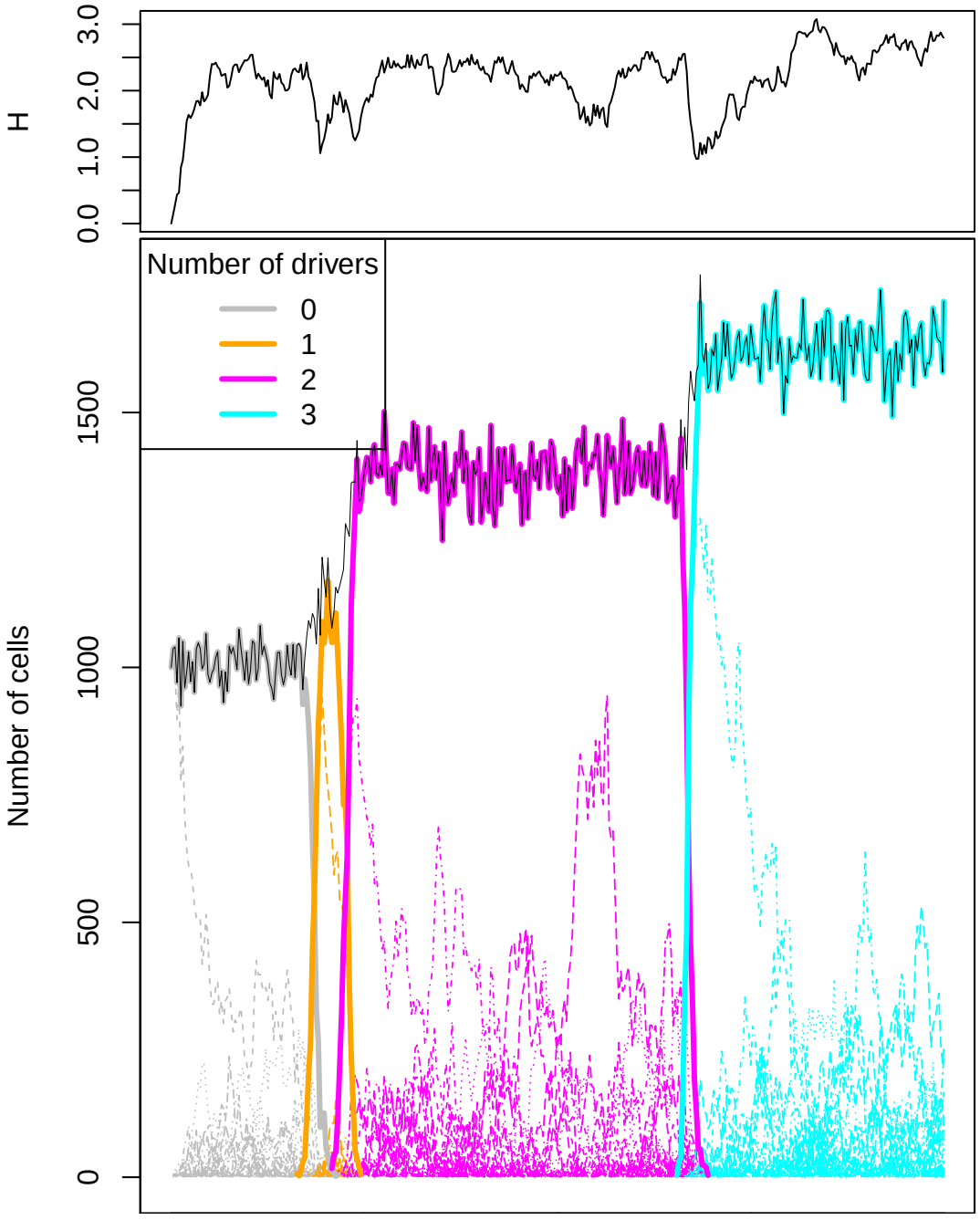
4.2 McFarland model with 50000 passengers and 70 drivers: clonal competition

The next is too slow (takes a couple of minutes in an i5 laptop) and too big to run in a vignette, because we keep track of over 4000 different clones (which leads to a result object of over 800 MB):

```
set.seed(123)
nd <- 70
np <- 50000
s <- 0.1
sp <- 1e-4 ## as we have many more passengers
spp <- -sp/(1 + sp)
mcfL <- allFitnessEffects(noIntGenes = c(rep(s, nd), rep(spp, np)),
                          drv = seq.int(nd))
mcfLs <-  oncoSimulIndiv(mcfL,
                         model = "McFL",
                         mu = 1e-7,
                         detectionSize = 1e8,
                         detectionDrivers = 100,
                         sampleEvery = 0.02,
                         keepEvery = 2,
                         initSize = 1000,
                         finalTime = 2000,
                         onlyCancer = FALSE)
```

But you can access the pre-stored results and plot them (beware: this object has been trimmed by removing empty passenger rows in the Genotype matrix)

```
data(mcfLs)
plot(mcfLs, addtot = TRUE, lwdClone = 0.9, log = "", plotDiversity = TRUE)
```



The argument `plotDiversity = TRUE` asks to show a small plot on top with Shannon's diversity index.

```
summary(mcfLs)
```

```
##   NumClones TotalPopSize LargestClone MaxNumDrivers MaxDriversLast NumDriversLargestPop
## 1      4458         1718          253             3              3                    3
##   TotalPresentDrivers FinalTime NumIter HittedWallTime    errorMF minDMratio minBMratio
## 1                  70      2000  113759          FALSE 0.01921737   184.1019   199.6085
##   OccurringDrivers
## 1   13, 38, 40, 69
```

```
## number of passengers per clone
summary(colSums(mcfLs$Genotypes[-(1:70), ]))
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
```

```
##   0.000   4.000   6.000   5.673   7.750  13.000
```

Note that we see clonal competition between clones with the same number of drivers (and with different drivers, of course). We will return to this (section **??**).

4.3 Loading fitnessEffects data for simulation examples

We will use several of the previous examples. Most of them are in file `examplesFitnessEffects`, where they are stored inside a list, with named components (names the same as in the examples above):

```
data(examplesFitnessEffects)
names(examplesFitnessEffects)
```

```
##  [1] "cbn1"   "cbn2"   "smn1"   "xor1"   "fp3"    "fp4m"   "o3"     "ofe1"   "ofe2"
## [10] "foi1"   "sv"     "svB"    "svB1"   "sv2"    "sm1"    "e2"     "E3A"    "em"
## [19] "fea"    "fbauer" "w"      "pancr"
```

4.4 Simulation with a conjunction example

We will simulate using the simple CBN-like restrictions of section **??** with two different models:

```
data(examplesFitnessEffects)
evalAllGenotypes(examplesFitnessEffects$cbn1, order = FALSE)[1:10, ]
```

```
##    Genotype Fitness
## 1         a    1.10
## 2         b    1.10
## 3         c    0.10
## 4         d    1.10
## 5         e    1.10
## 6         g    0.10
## 7      a, b    1.21
## 8      a, c    0.11
## 9      a, d    1.21
## 10     a, e    1.21
```

```
sm <-  oncoSimulIndiv(examplesFitnessEffects$cbn1,
                      model = "McFL",
                      mu = 5e-7,
                      detectionSize = 1e8,
                      detectionDrivers = 2,
                      sampleEvery = 0.025,
                      keepEvery = 5,
                      initSize = 2000,
                      onlyCancer = TRUE)
summary(sm)
```

```
##   NumClones TotalPopSize LargestClone MaxNumDrivers MaxDriversLast NumDriversLargestPop
## 1         4         2890         2752             2              2                    2
##   TotalPresentDrivers FinalTime NumIter HittedWallTime    errorMF minDMratio minBMratio
## 1                   6  1987.125   79499          FALSE 0.01596409   305323.7      40000
##   OccurringDrivers
## 1          a, b, d
```

```
evalAllGenotypes(examplesFitnessEffects$cbn1, order = FALSE,
                 model = "Bozic")[1:10, ]
```

```
##    Genotype Death_rate
## 1         a       0.90
## 2         b       0.90
## 3         c       1.90
## 4         d       0.90
## 5         e       0.90
## 6         g       1.90
## 7      a, b       0.81
## 8      a, c       1.71
## 9      a, d       0.81
## 10     a, e       0.81
```

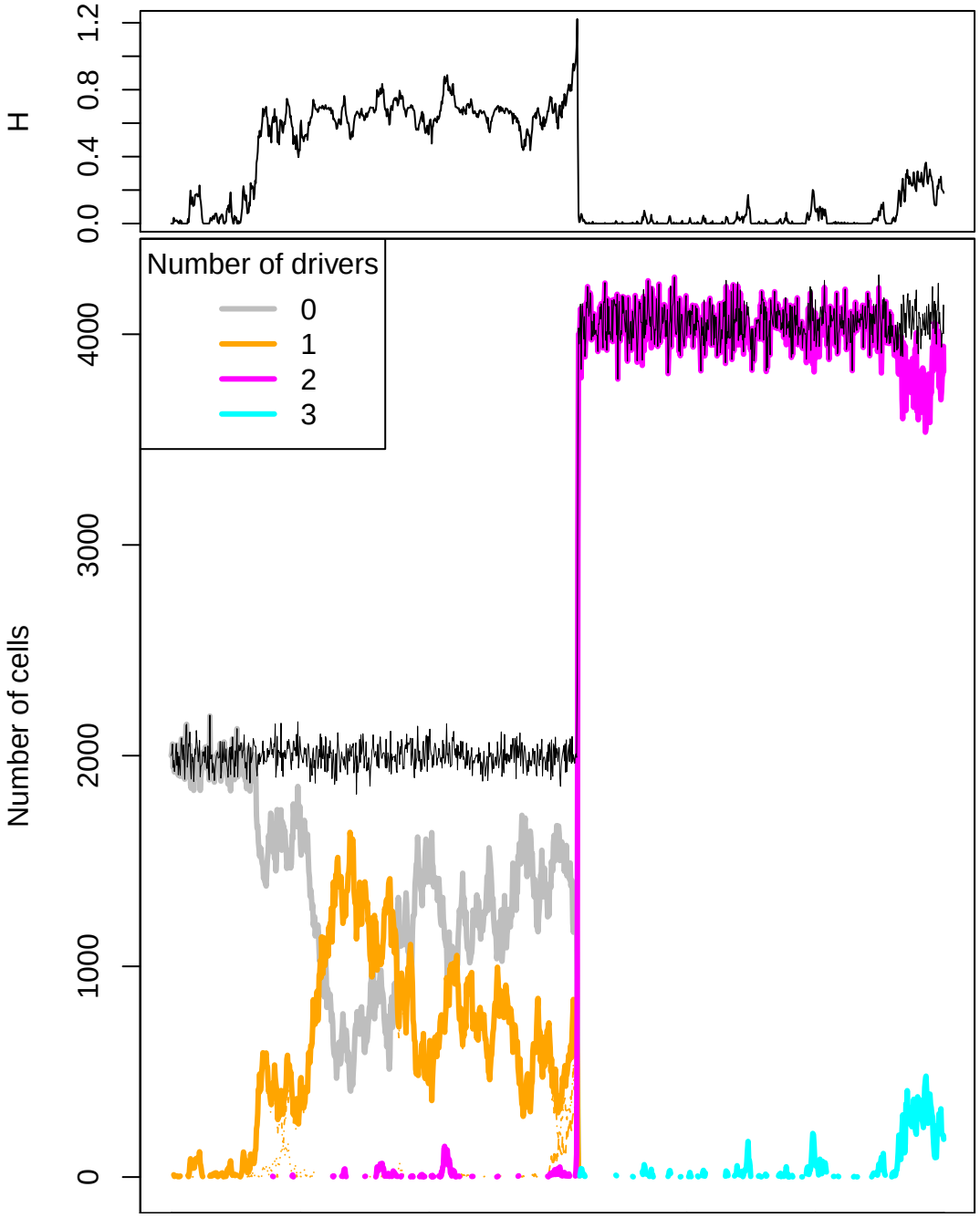
```
sb <-  oncoSimulIndiv(examplesFitnessEffects$cbn1,
                      model = "Bozic",
                      mu = 1e-6,
                      detectionSize = 1e8,
                      detectionDrivers = 4,
                      sampleEvery = 2,
                      initSize = 2000,
                      onlyCancer = TRUE)
summary(sb)
```

```
##   NumClones TotalPopSize LargestClone MaxNumDrivers MaxDriversLast NumDriversLargestPop
## 1         9    107734130    107716454             2              2                    1
##   TotalPresentDrivers FinalTime NumIter HittedWallTime errorMF minDMratio minBMratio
## 1                   6       392    5427          FALSE      NA   166666.7   166666.7
##   OccurringDrivers
## 1 a, b, c, d, e, g
```

4.5 Simulation with order effects and McFL model

(We use a somewhat large mutation rate than usual, so that the simulation runs quickly.)

```
set.seed(4321)
tmp <-  oncoSimulIndiv(examplesFitnessEffects[["o3"]],
                       model = "McFL",
                       mu = 5e-5,
                       detectionSize = 1e8,
                       detectionDrivers = 3,
                       sampleEvery = 0.025,
                       max.num.tries = 10,
                       keepEvery = 5,
                       initSize = 2000,
                       finalTime = 6000,
                       onlyCancer = FALSE);
plot(tmp, addtot = TRUE, log = "", plotDiversity = TRUE)
```



In this example (and at least under Linux, with both GCC and clang), we can see that the mutants with three drivers do not get established when we stop the simulation at time 6000. This is one case where the summary statistics about number of drivers says little of value, as fitness is very different for genotypes with the same number of mutations, and does not increase in a simple way with drivers:

```
evalAllGenotypes(examplesFitnessEffects[["o3"]])
```

```
##     Genotype Fitness
## 1          d    1.00
## 2          f    1.00
## 3          m    1.00
## 4      d > f    1.00
## 5      d > m    1.10
## 6      f > d    1.00
```

```
## 7      f > m    1.00
## 8      m > d    1.50
## 9      m > f    1.00
## 10 d > f > m    1.54
## 11 d > m > f    1.32
## 12 f > d > m    0.77
## 13 f > m > d    1.50
## 14 m > d > f    1.50
## 15 m > f > d    1.50
```

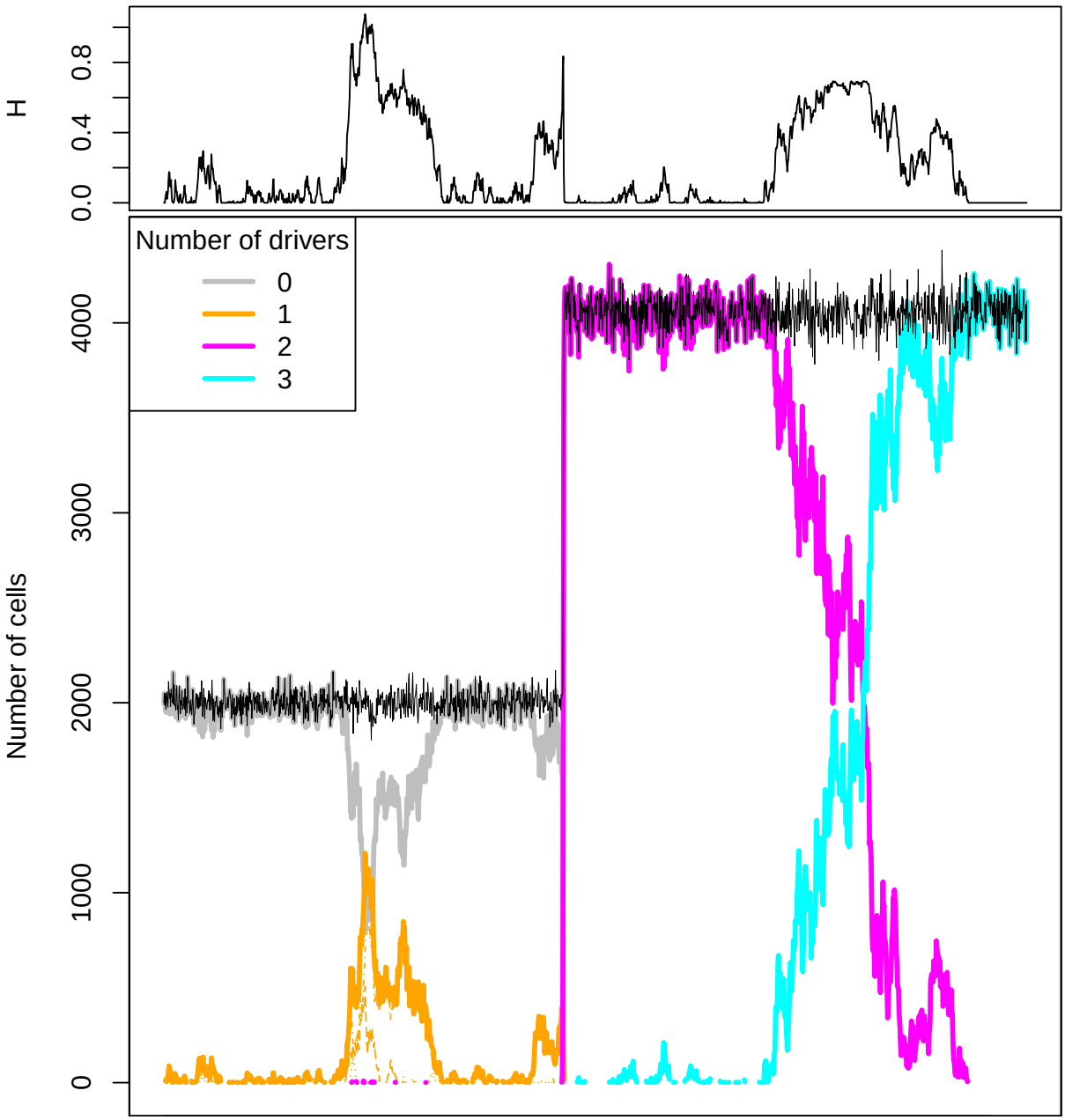
In this case, the clones with three drivers end up displacing those with two by the time we stop; moreover, notice how those with one driver never really grow to a large population size, so we basically go from a population with clones with zero drivers to a population made of clones with two or three drivers:

```
set.seed(15)
tmp <-  oncoSimulIndiv(examplesFitnessEffects[["o3"]],
                       model = "McFL",
                       mu = 5e-5,
                       detectionSize = 1e8,
                       detectionDrivers = 3,
                       sampleEvery = 0.015,
                       max.num.tries = 10,
                       keepEvery = 5,
                       initSize = 2000,
                       finalTime = 20000,
                       onlyCancer = FALSE,
                       extraTime = 1500)
tmp
```

```
##
## Individual OncoSimul trajectory with call:
##  oncoSimulIndiv(fp = examplesFitnessEffects[["o3"]], model = "McFL",
##     mu = 5e-05, detectionSize = 1e+08, detectionDrivers = 3,
##     sampleEvery = 0.015, initSize = 2000, keepEvery = 5, extraTime = 1500,
##     finalTime = 20000, onlyCancer = FALSE, max.num.tries = 10)
##
##   NumClones TotalPopSize LargestClone MaxNumDrivers MaxDriversLast NumDriversLargestPop
## 1        10         4113         4113             3              3                    3
##   TotalPresentDrivers FinalTime NumIter HittedWallTime    errorMF minDMratio minBMratio
## 1                   3  8354.805  558669          FALSE 0.01143976   6152.152   6666.667
##   OccurringDrivers
## 1          d, f, m
##
## Final population composition:
##    Genotype    N
## 1         _    0
## 2        d_    0
## 3     d, f_    0
## 4        f_    0
## 5     f, d_    0
## 6     f, m_    0
## 7        m_    0
```

```
## 8     m, d_    0
## 9  m, d, f_ 4113
## 10    m, f_    0
```

```
plot(tmp, addtot = TRUE, log = "", plotDiversity = TRUE)
```



As before, the argument `plotDiversity = TRUE` asks to show a small plot on top with Shannon's diversity index. Here, as before, the quick clonal expansion of the clone with two drivers leads to a sudden drop in diversity (for a while, the population is made virtually of a single clone). Note, however, that compared to section **??**, we are modeling here a scenario with very few genes, and correspondingly very few possible genotypes, and thus it is not strange that we observe very little diversity.

(We have used `extraTime` to continue the simulation well past the point of detection, here specified as three drivers. Instead of specifying `extraTime` we can set the `detectionDrivers` value to a number larger than the number of existing possible drivers, and the simulation will run until `finalTime` if `onlyCancer = FALSE`.)

4.6 Numerical issues with Bozic

As we mentioned above (section **??**) death rates of 0 can lead to trouble when using Bozic's model:

```
i1 <- allFitnessEffects(noIntGenes = c(1))
evalAllGenotypes(i1, order = FALSE, addwt = TRUE,
                 model = "Bozic")
```

```
##   Genotype Death_rate
## 1       wt          1
## 2        1          0
```

```
i1_b <- oncoSimulIndiv(i1, model = "Bozic")
```

```
## Warning in nr_oncoSimul.internal(rFE = fp, birth = birth, death = death, :  You are
using a Bozic model with the new restriction specification, and you have at least one
s of 1.  If that gene is mutated,  this will lead to a death rate of 0 and the simulations
will abort when you get a non finite value.
```

```
##
##  DEBUG2: Value of rnb = nan
##
##  DEBUG2: Value of m = 1
##
##  DEBUG2: Value of pe = 0
##
##  DEBUG2: Value of pm = 1
##
##  this is spP
##
##  popSize = 1
##  birth = 1
##  death = 0
##  W = 1
##  R = 1
##  mutation = 1e-09
##  timeLastUpdate = 1232.69
##  absfitness = -inf
##  numMutablePos =0
##
##  Unrecoverable exception: Algo 2: retval not finite. Aborting.
```

Of course, there is no problem in using the above with other models:

```
evalAllGenotypes(i1, order = FALSE, addwt = TRUE,
                 model = "Exp")
```

```
##   Genotype Fitness
## 1       wt       1
## 2        1       2
```

```
i1_e <- oncoSimulIndiv(i1, model = "Exp")
summary(i1_e)
```

```
##   NumClones TotalPopSize LargestClone MaxNumDrivers MaxDriversLast NumDriversLargestPop
## 1         2    265551875    265551583             0              0                   0
```

```
##   TotalPresentDrivers FinalTime NumIter HittedWallTime errorMF minDMratio minBMratio
## 1                   0       406     409          FALSE      NA      1e+06      1e+06
##   OccurringDrivers
## 1
```

5 Sampling multiple simulations

Often, you will want to simulate multiple runs of the same scenario, and do something with them. Conceptually, the first step is running multiple simulations and, then, sampling them.

We will use the "pancreas" example, above section **??**.

```
pancrPop <- oncoSimulPop(10, pancr,
                         detectionSize = 1e7,
                         keepEvery = 10,
                         mc.cores = 2)

summary(pancrPop)
```

```
##    NumClones TotalPopSize LargestClone MaxNumDrivers MaxDriversLast NumDriversLargestPop
## 1          8     10636686     10632422             2              2                    1
## 2         11     10820595     10809082             2              2                    1
## 3          8     10140318     10137750             2              2                    1
## 4          8     10544766     10543709             2              2                    1
## 5         11     10277052     10272947             2              2                    1
## 6          9     10843754     10842919             2              2                    1
## 7         11     10168698     10162920             3              3                    1
## 8          9     10844985     10841954             2              2                    1
## 9         10     10853107     10851383             3              3                    1
## 10         9     10191543     10187880             2              2                    1
##    TotalPresentDrivers FinalTime NumIter HittedWallTime errorMF minDMratio minBMratio
## 1                    7      1373    2141          FALSE      NA   142857.1   142857.1
## 2                    7      2032    2705          FALSE      NA   142857.1   142857.1
## 3                    7       267     927          FALSE      NA   142857.1   142857.1
## 4                    7      1142    1843          FALSE      NA   142857.1   142857.1
## 5                    7      2075    2819          FALSE      NA   142857.1   142857.1
## 6                    7      1250    1958          FALSE      NA   142857.1   142857.1
## 7                    7      1302    1998          FALSE      NA   142857.1   142857.1
## 8                    7      1937    2646          FALSE      NA   142857.1   142857.1
## 9                    7       247     998          FALSE      NA   142857.1   142857.1
## 10                   7      1141    1810          FALSE      NA   142857.1   142857.1
##                                 OccurringDrivers
## 1  CDNK2A, KRAS, MLL3, PXDN, SMAD4, TGFBR2, TP53
## 2  CDNK2A, KRAS, MLL3, PXDN, SMAD4, TGFBR2, TP53
## 3  CDNK2A, KRAS, MLL3, PXDN, SMAD4, TGFBR2, TP53
## 4  CDNK2A, KRAS, MLL3, PXDN, SMAD4, TGFBR2, TP53
## 5  CDNK2A, KRAS, MLL3, PXDN, SMAD4, TGFBR2, TP53
## 6  CDNK2A, KRAS, MLL3, PXDN, SMAD4, TGFBR2, TP53
## 7  CDNK2A, KRAS, MLL3, PXDN, SMAD4, TGFBR2, TP53
## 8  CDNK2A, KRAS, MLL3, PXDN, SMAD4, TGFBR2, TP53
## 9  CDNK2A, KRAS, MLL3, PXDN, SMAD4, TGFBR2, TP53
```

```
## 10 CDNK2A, KRAS, MLL3, PXDN, SMAD4, TGFBR2, TP53
```

The above runs the simulation process 10 times, and stores the results. We can then sample from them:

```
pancrSPop <- samplePop(pancrPop)
```

```
##
##  Subjects by Genes matrix of 10 subjects and 7 genes.
```

```
pancrSPop
```

```
##       CDNK2A KRAS MLL3 PXDN SMAD4 TGFBR2 TP53
##  [1,]      0    1    0    0     0      0    0
##  [2,]      0    1    0    0     0      0    0
##  [3,]      0    1    0    0     0      0    0
##  [4,]      0    1    0    0     0      0    0
##  [5,]      0    1    0    0     0      0    0
##  [6,]      0    1    0    0     0      0    0
##  [7,]      0    1    0    0     0      0    0
##  [8,]      0    1    0    0     0      0    0
##  [9,]      0    1    0    0     0      0    0
## [10,]      0    1    0    0     0      0    0
```

But if we are only interested in the final matrix of populations by mutations, the above is wasteful, because we store fully all of the simulations (in the call to `oncoSimulPop`) and then sample (in the call to `samplePop`). In particular, data from every sampling time (as given by `sampleEvery`) is preserved. It is in the call to `samplePop` when we actually sample the data.

An alternative approach is to use the function `oncoSimulSample`. The output is directly the matrix (and a little bit of summary from each run), and during the simulation it only stores one time point.

```
pancrSamp <- oncoSimulSample(10, pancr)
```

```
## Successfully sampled 10 individuals
```

```
##
##  Subjects by Genes matrix of 10 subjects and 7 genes.
```

```
pancrSamp
```

```
## $popSummary
##    NumClones TotalPopSize LargestClone MaxNumDrivers MaxDriversLast NumDriversLargestPop
## 1          8     22612333     22607385             2             2                    1
## 2          8     75002918     74994579             2             2                    1
## 3         11    103186869    103171946             3             3                    1
## 4         10     71719757     71705847             3             3                    1
## 5          9     22429976     22421307             3             3                    1
## 6          8     42338176     42333588             2             2                    1
## 7          3        16044        15048             2             2                    1
## 8          2        32455        32454             2             2                    1
## 9          6       268046       264433             3             3                    1
## 10         3        20053        18918             2             2                    1
##    TotalPresentDrivers FinalTime NumIter HittedWallTime errorMF minDMratio minBMratio
## 1                    7       514    2048          FALSE      NA   142857.1   142857.1
## 2                    7      2241    7222          FALSE      NA   142857.1   142857.1
## 3                    7      1486    8196          FALSE      NA   142857.1   142857.1
## 4                    7       627    5348          FALSE      NA   142857.1   142857.1
```

```
## 5                    7       154    1604          FALSE      NA   142857.1   142857.1
## 6                    7       248    3028          FALSE      NA   142857.1   142857.1
## 7                    7       217     221          FALSE      NA   142857.1   142857.1
## 8                    7       258     264          FALSE      NA   142857.1   142857.1
## 9                    7      1502    1527          FALSE      NA   142857.1   142857.1
## 10                   7       617     623          FALSE      NA   142857.1   142857.1
##                                    OccurringDrivers
## 1  CDNK2A, KRAS, MLL3, PXDN, SMAD4, TGFBR2, TP53
## 2  CDNK2A, KRAS, MLL3, PXDN, SMAD4, TGFBR2, TP53
## 3  CDNK2A, KRAS, MLL3, PXDN, SMAD4, TGFBR2, TP53
## 4  CDNK2A, KRAS, MLL3, PXDN, SMAD4, TGFBR2, TP53
## 5  CDNK2A, KRAS, MLL3, PXDN, SMAD4, TGFBR2, TP53
## 6  CDNK2A, KRAS, MLL3, PXDN, SMAD4, TGFBR2, TP53
## 7                                    KRAS, TGFBR2
## 8                                      KRAS, PXDN
## 9                      CDNK2A, KRAS, SMAD4, TP53
## 10                                     KRAS, TP53
##
## $popSample
##       CDNK2A KRAS MLL3 PXDN SMAD4 TGFBR2 TP53
##  [1,]      0    1    0    0     0      0    0
##  [2,]      0    1    0    0     0      0    0
##  [3,]      0    1    0    0     0      0    0
##  [4,]      0    1    0    0     0      0    0
##  [5,]      0    1    0    0     0      0    0
##  [6,]      0    1    0    0     0      0    0
##  [7,]      0    1    0    0     0      0    0
##  [8,]      0    1    0    0     0      0    0
##  [9,]      0    1    0    0     0      0    0
## [10,]      0    1    0    0     0      0    0
##
## $attemptsUsed
## [1] 146
##
## $probCancer
## [1] 0.06849315
##
## $HittedMaxTries
## [1] FALSE
##
## $HittedWallTime
## [1] FALSE
##
## $UnrecoverExcept
## [1] FALSE
```

5.1 Differences between `samplePop` and `oncoSimulSample`

`samplePop` provides two sampling times: "last" and "uniform". "last" means to sample each individual in the very last time period of the simulation. "uniform" means sampling each individual at a time choosen

uniformly from all the times recorded in the simulation between the time when the first driver appeared and the final time period. "unif" means that it is almost sure that different individuals will be sampled at different times. "last" does not guarantee that different individuals will be sampled at the same time unit, only that all will be sampled in the last time unit of their simulation.

With `oncoSimulSample` we obtain samples that correspond to `timeSample = ''last''` in `samplePop` by specifying a unique value for `detectionSize` and `detectionDrivers`. The data from each simulation will correspond to the time point at which those are reached (analogous to `timeSample = ''last''`). How about uniform sampling? We pass a vector of `detectionSize` and `detectionDrivers`, where each value of the vector comes from a uniform distribution. This is not identical to the "uniform" sampling of `oncoSimulSample`, as we are not sampling uniformly over all time periods, but are stopping at uniformly distributed values over the stopping conditions. Arguably, however, the procedure in `samplePop` might be closer to what we mean with "uniformly sampled over the course of the disease" if that course is measured in terms of drivers or size of tumor.

As an example, if you look at the output above, the object "pancrSamp" contains some simulations that have only a few drivers because those simulations were set to run only until they had just a small number of cells.

An additional advantage of `oncoSimulSample` is that we can specify arbitrary sampling schemes, just by passing the appropriate vector `detectionSize` and `detectionDrivers`. A disadvantage is that if we change the stopping conditions we can not just resample the data, but we need to run it again.

There is no difference between `oncoSimulSample` and `oncoSimulPop` + `samplePop` in terms of the `typeSample` argument (whole tumor or single cell).

Finally, there are some additional differences between the two functions. `oncoSimulPop` can run parallelized (it uses `mclapply`). This is not done with `oncoSimulSample` because this function is designed for simulation experiments where you want to examine many different scenarios simultaneously. Thus, we provide additional stopping criteria (`max.wall.time.total` and `max.num.tries.total`) to determine whether to continue running the simulations, that bounds the total running time of all the simulations in a call to `oncoSimulSample`. And, if you are running multiple different scenarios, you might want to make multiple, separate, independent calls (e.g., from different R processes) to `oncoSimulSample`, instead of relying in `mclapply`, since this is likely to lead to better usage of multiple cores/CPUs if you are examining a large number of different scenarios.

5.2 What can you do with the simulations?

This is up to you. Below (section **??**) we show an example where we infer an oncogenetic tree from simulated data.

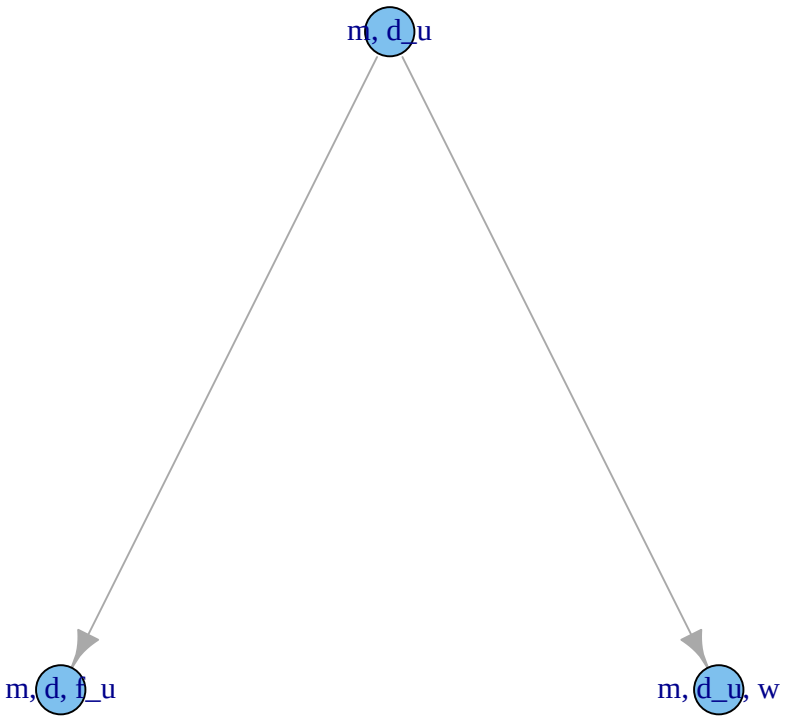
5.3 Whole tumor sampling and genotypes

You are obtaining genotypes, regardless of order. When we use "whole tumor sampling", it is the frequency of the mutations in each gene that counts, not the order. So, for instance, "c, d" and "c, d" both contribute to the counts of "c" and "d". Similarly, when we use single cell sampling, we obtain a genotype defined in terms of mutations, but there might be multiple orders that give this genotype. For example, $d > c$ and $c > d$ both give you a genotype with "c" and "d" mutated, and thus in the output you can have two columns with both genes mutated.

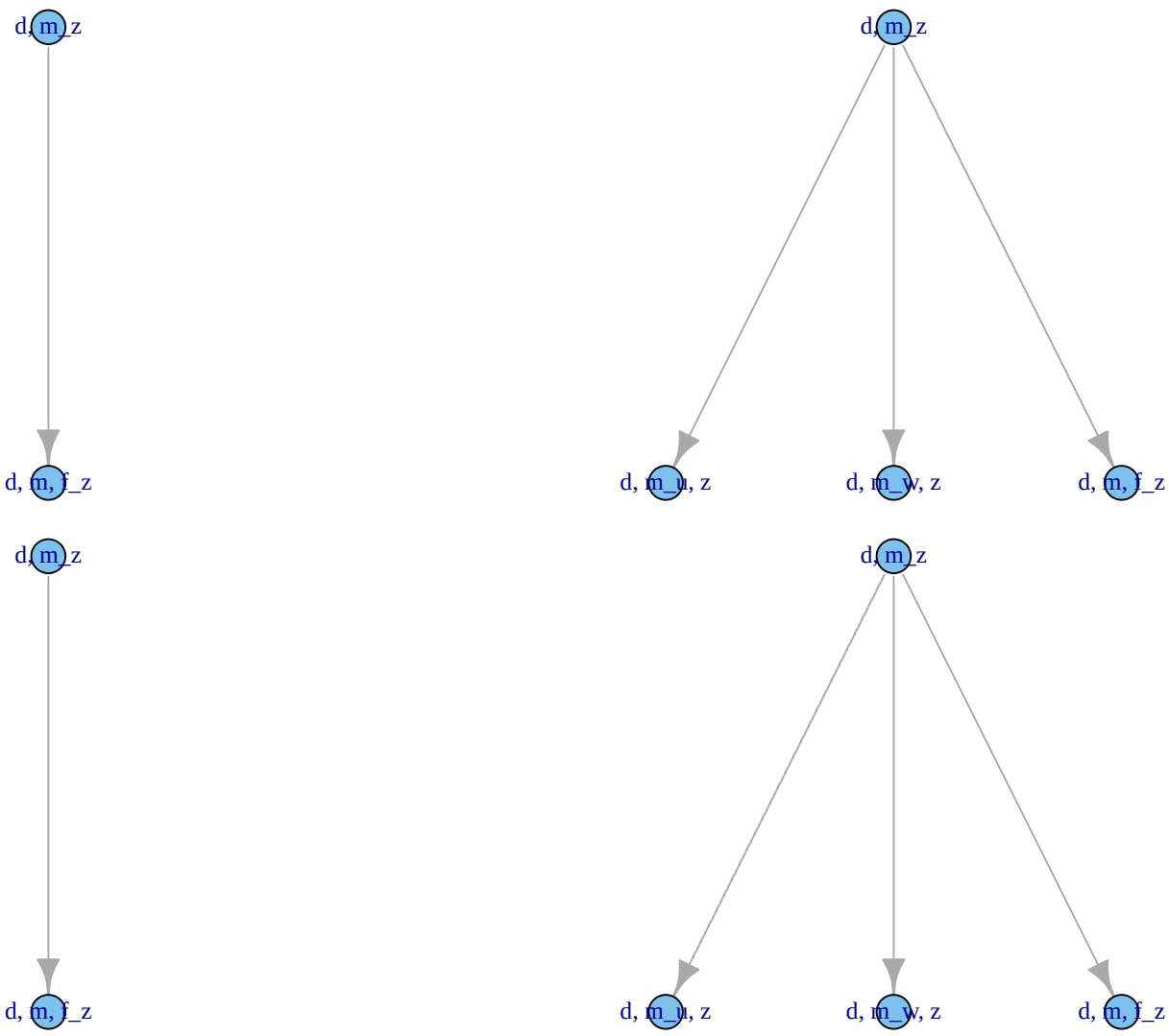
5.4 Can I start the simulation from a specific mutant?

You bet. In v.1 you can only give the initial mutant as one with a single mutated gene. In version 2, however, you can specify the genotype for the initial mutant with the same flexibility as in `evalGenotype`. Here we show a couple of examples (we use the representation of the phylogeny —discussed in section **??**— of the clones so that you can see which clones appear, and from which).

```
o3init <- allFitnessEffects(orderEffects = c(
                              "M > D > F" = 0.99,
                              "D > M > F" = 0.2,
                              "D > M"     = 0.1,
                              "M > D"     = 0.9),
                            noIntGenes = c("u" = 0.01,
                                           "v" = 0.01,
                                           "w" = 0.001,
                                           "x" = 0.0001,
                                           "y" = -0.0001,
                                           "z" = -0.001),
                            geneToModule =
                                c("Root" = "Root",
                                  "M" = "m",
                                  "F" = "f",
                                  "D" = "d") )

oneI <- oncoSimulIndiv(o3init, model = "McFL",
                       mu = 5e-5, finalTime = 500,
                       detectionDrivers = 3,
                       onlyCancer = FALSE,
                       initSize = 1000,
                       keepPhylog = TRUE,
                       initMutant = c("m > u > d")
                       )
plotClonePhylog(oneI, N = 0)
```




```
##
ospI <- oncoSimulPop(4,
                     o3init, model = "Exp",
                     mu = 5e-5, finalTime = 500,
                     detectionDrivers = 3,
                     onlyCancer = TRUE,
                     initSize = 10,
                     keepPhylog = TRUE,
                     initMutant = c("d > m > z"),
                     mc.cores = 2
                     )

op <- par(mar = rep(0, 4), mfrow = c(2, 2))
plotClonePhylog(ospI[[1]])
plotClonePhylog(ospI[[2]])
plotClonePhylog(ospI[[3]])
plotClonePhylog(ospI[[4]])
```




```
par(op)


ossI <- oncoSimulSample(4,
                        o3init, model = "Exp",
                        mu = 5e-5, finalTime = 500,
                        detectionDrivers = 2,
                        onlyCancer = TRUE,
                        initSize = 10,
                        initMutant = c("z > d"),
                        thresholdWhole = 1 ## check presence of initMutant
                    )

## Successfully sampled 4 individuals

##
##  Subjects by Genes matrix of 4 subjects and 9 genes.

## No phylogeny is kept with oncoSimulSample, but look at the
## OcurringDrivers and the sample

ossI$popSample
```

```
##      d f m u v w x y z
## [1,] 1 0 0 0 0 0 0 0 1
## [2,] 1 0 0 0 0 0 0 0 1
## [3,] 1 0 0 0 0 0 0 0 1
## [4,] 1 0 0 0 0 0 0 0 1
```

```
ossI$popSummary[, "OccurringDrivers", drop = FALSE]
```

```
##   OccurringDrivers
## 1             d, m
## 2             d, m
## 3             d, m
## 4             d, m
```

6 Showing the true phylogenetic relationships of clones

If you run simulations with the `keepPhylog = TRUE` argument, the simulations keep track of when every clone is generated, and that will allow us to see the true phylogenetic relationships of clones. (This is disabled by default: the code runs a little bit slower and the result is larger.)

Let us re-run a previous example:

```
set.seed(15)
tmp <-  oncoSimulIndiv(examplesFitnessEffects[["o3"]],
                       model = "McFL",
                       mu = 5e-5,
                       detectionSize = 1e8,
                       detectionDrivers = 3,
                       sampleEvery = 0.015,
                       max.num.tries = 10,
                       keepEvery = 5,
                       initSize = 2000,
                       finalTime = 20000,
                       onlyCancer = FALSE,
                       extraTime = 1500,
                       keepPhylog = TRUE)
tmp
```

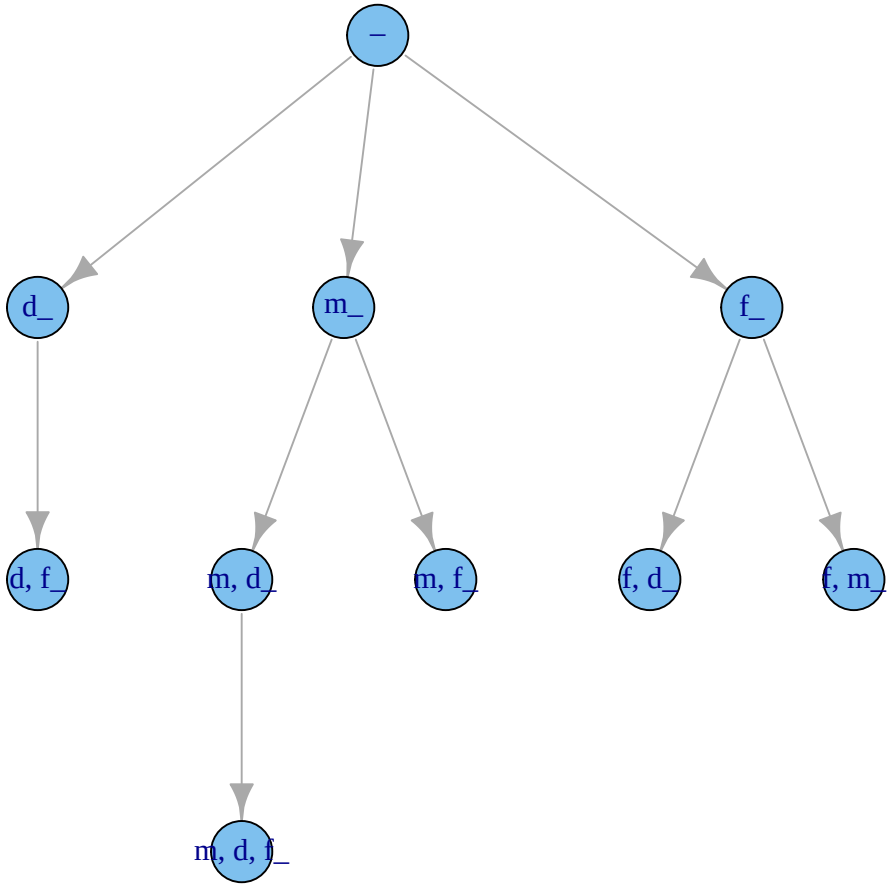
```
##
## Individual OncoSimul trajectory with call:
##  oncoSimulIndiv(fp = examplesFitnessEffects[["o3"]], model = "McFL",
##     mu = 5e-05, detectionSize = 1e+08, detectionDrivers = 3,
##     sampleEvery = 0.015, initSize = 2000, keepEvery = 5, extraTime = 1500,
##     finalTime = 20000, onlyCancer = FALSE, keepPhylog = TRUE,
##     max.num.tries = 10)
##
##   NumClones TotalPopSize LargestClone MaxNumDrivers MaxDriversLast NumDriversLargestPop
## 1        10         4113         4113             3              3                   3
##   TotalPresentDrivers FinalTime NumIter HittedWallTime    errorMF minDMratio minBMratio
## 1                   3  8354.805  558669          FALSE 0.01143976   6152.152   6666.667
##   OccurringDrivers
## 1          d, f, m
```

```
##
## Final population composition:
##    Genotype    N
## 1         _    0
## 2        d_    0
## 3     d, f_    0
## 4        f_    0
## 5     f, d_    0
## 6     f, m_    0
## 7        m_    0
## 8     m, d_    0
## 9  m, d, f_ 4113
## 10    m, f_    0
```

We can plot the phylogenetic relationships[7] of every clone ever created (with fitness larger than 0 —clones without viability are never shown):

```
plotClonePhylog(tmp, N = 0)
```

[7] There are several packages in R devoted to phylogenetic inference and related issues. For instance, *ape*. I have not used that infrastructure because of our very specific needs and circumstances; for instance, internal nodes are observed, we can have networks instead of trees, and we have no uncertainty about when events occurred.



However, we often only want to show clones that exist (have number of cells > 0) at a certain time (while of course showing all of their ancestors, even if those are now extinct —i.e., regardless of their current numbers).

```
plotClonePhylog(tmp, N = 1)
```

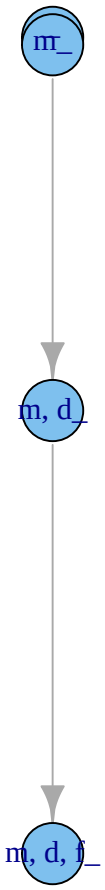
If we set `keepEvents = TRUE` the arrows show how many times each clone appeared:

(The next can take a while)

```
plotClonePhylog(tmp, N = 1, keepEvents = TRUE)
```

And we can plot the phylogeny so the vertical axis is proportional to time (though you might see overlap of nodes if a child node appeared shortly after the parent):

```
plotClonePhylog(tmp, N = 1, timeEvents = TRUE)
```



We can obtain the adjacency matrix doing

```
get.adjacency(plotClonePhylog(tmp, N = 1, returnGraph = TRUE))
```

```
## 4 x 4 sparse Matrix of class "dgCMatrix"
##          _ m_ m, d_ m, d, f_
## _        .  1     .        .
## m_       .  .     1        .
## m, d_    .  .     .        1
## m, d, f_ .  .     .        .
```

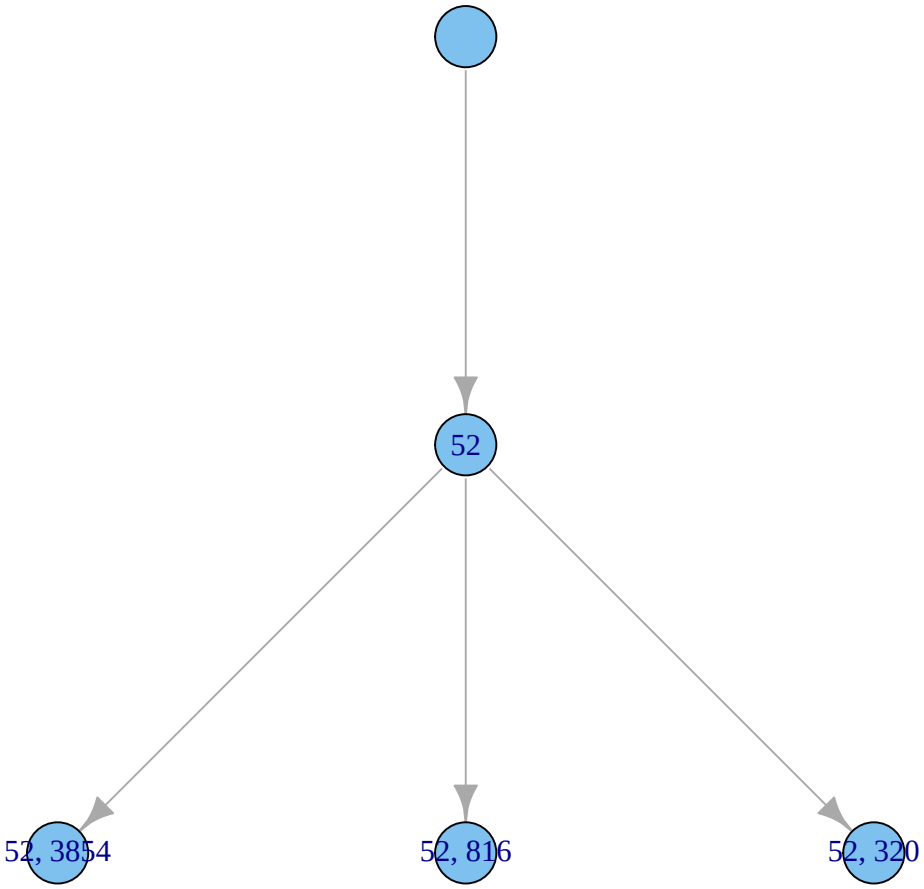
We can see another example here:

```
set.seed(456)
mcf1s <-  oncoSimulIndiv(mcf1,
                         model = "McFL",
                         mu = 1e-7,
```

```
                        detectionSize = 1e8,
                        detectionDrivers = 100,
                        sampleEvery = 0.02,
                        keepEvery = 2,
                        initSize = 2000,
                        finalTime = 1000,
                        onlyCancer = FALSE,
                        keepPhylog = TRUE)
```

Showing only clones that exist at the end of the simulation (and all their parents):

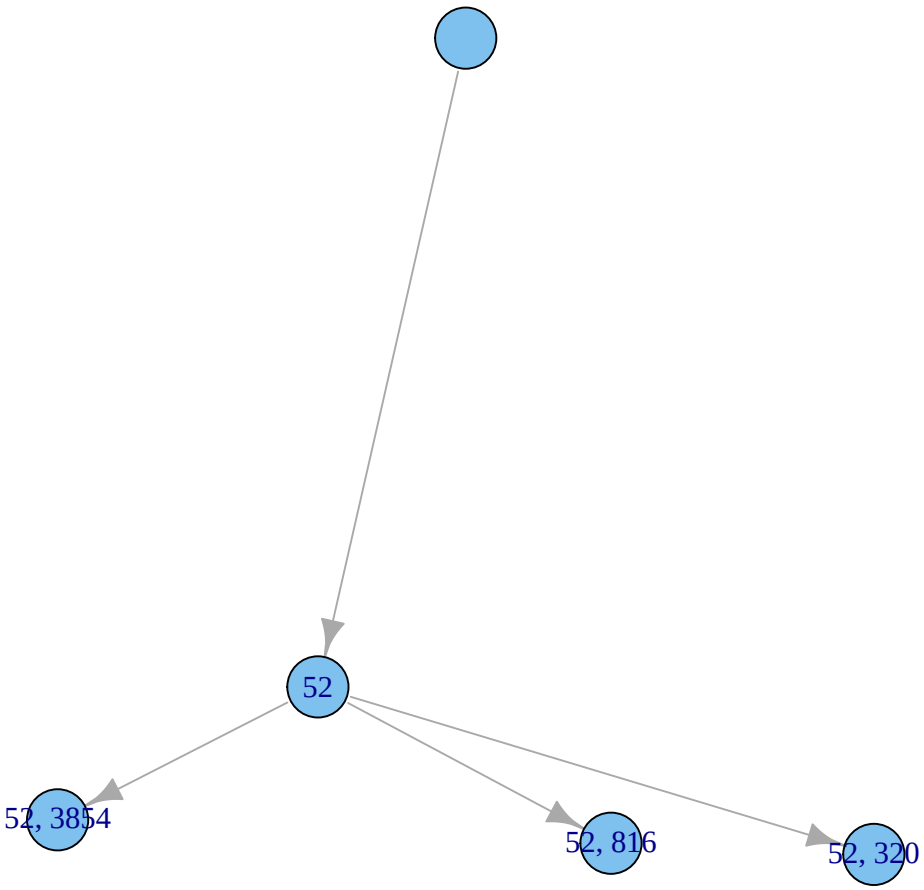
```
plotClonePhylog(mcf1s, N = 1)
```



Notice that the labels here do not have a "_", since there were no order effects in fitness. However, the labels show the genes that are mutated, just as before.

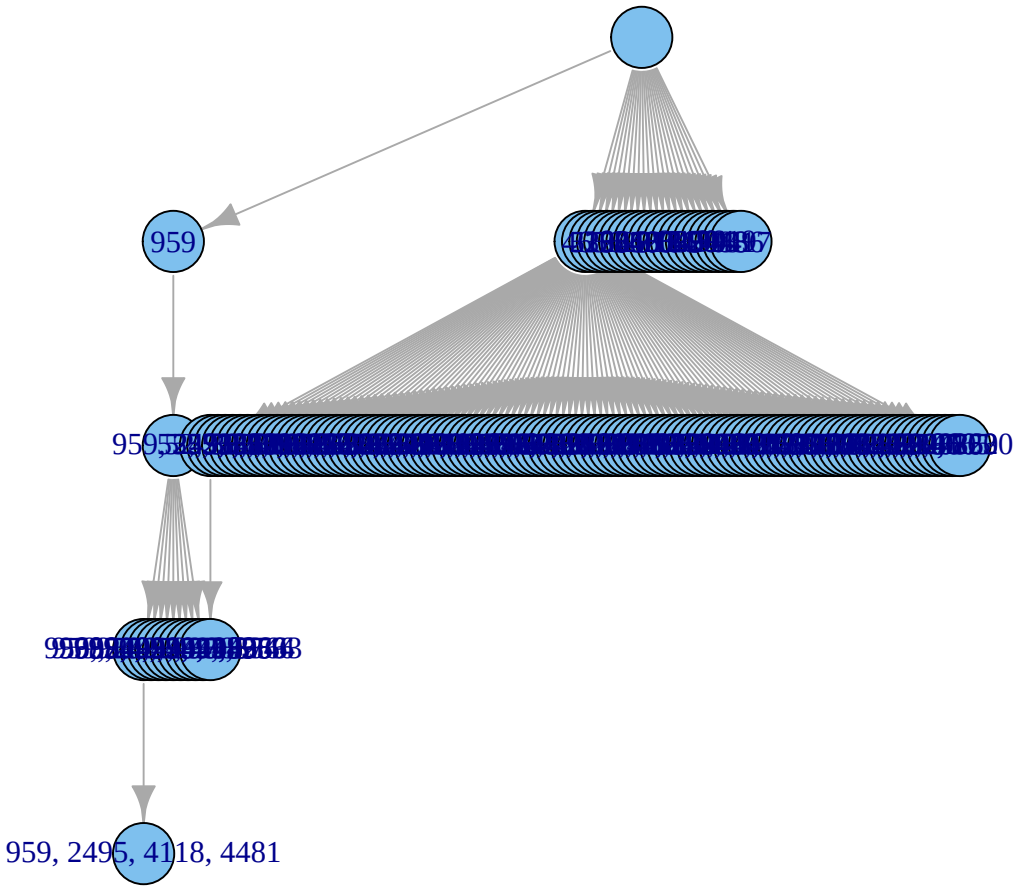
Similar, but with vertical axis proportional to time:

```
plotClonePhylog(mcf1s, N = 1, timeEvents = TRUE)
```



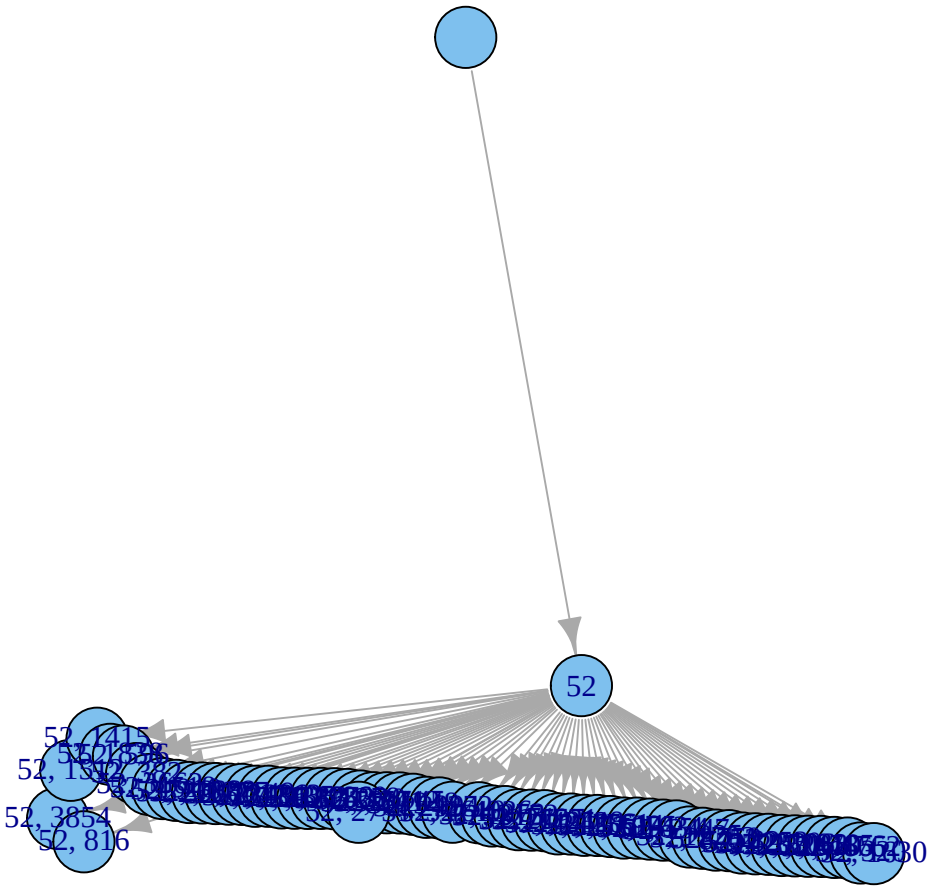
What about those that existed in the last 200 time units?

```
plotClonePhylog(mcf1s, N = 1, t = c(800, 1000))
```



And try now to show also when the clones appeared (we restrict the time to between 900 and 1000, to avoid too much clutter):

```
plotClonePhylog(mcf1s, N = 1, t = c(900, 1000), timeEvents = TRUE)
```

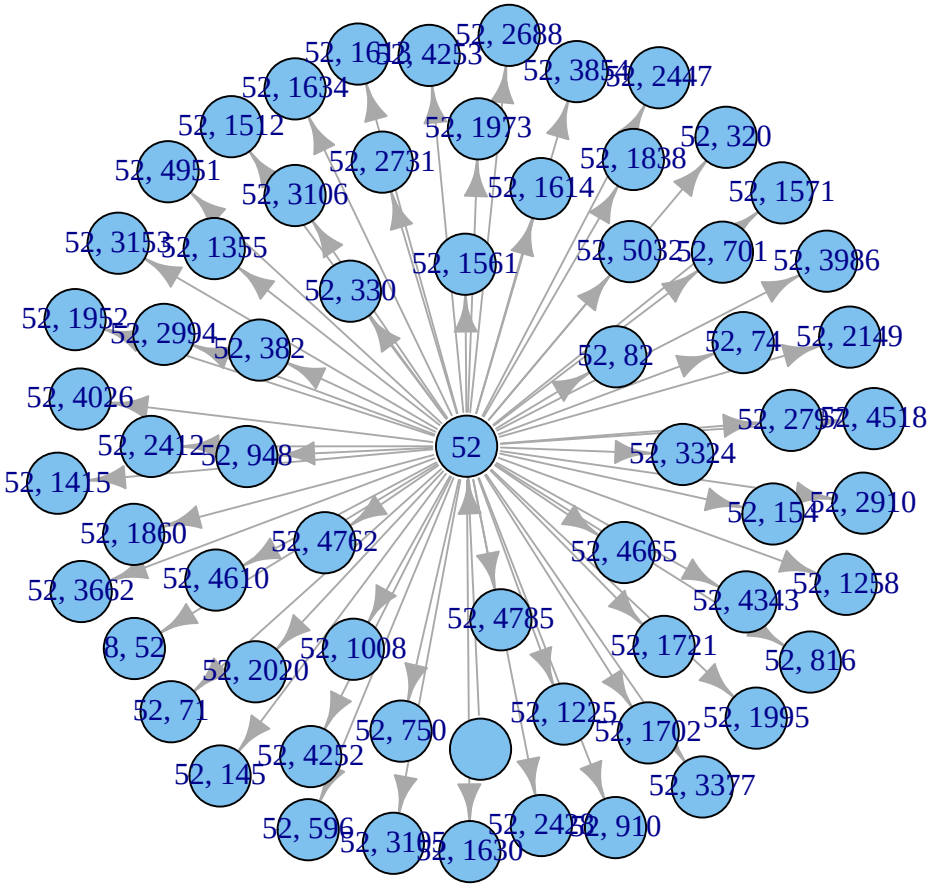


(By playing with `t`, it should be possible to obtain animations of the phylogeny. We will not pursue it here.)

If the previous graph seems cluttered, we can represent it in a different way by calling *igraph* directly after storing the graph and using the default layout:

```
g1 <- plotClonePhylog(mcf1s, N = 1, t = c(900, 1000), returnGraph = TRUE)
```

```
plot(g1)
```



which might be easier to show complex relationships or identify central or key clones.

It is of course quite possible that, especially if we consider few genes, our phylogeny will be a network, not a tree, as the same child node can have multiple parents. You can play with this example, modified from one we saw before (section **??**):

```
op <- par(ask = TRUE)
while(TRUE) {
    tmp <- oncoSimulIndiv(smn1, model = "McFL",
                          mu = 5e-5, finalTime = 500,
                          detectionDrivers = 3,
                          onlyCancer = FALSE,
                          initSize = 1000, keepPhylog = TRUE)
    plotClonePhylog(tmp, N = 0)
}
par(op)
```

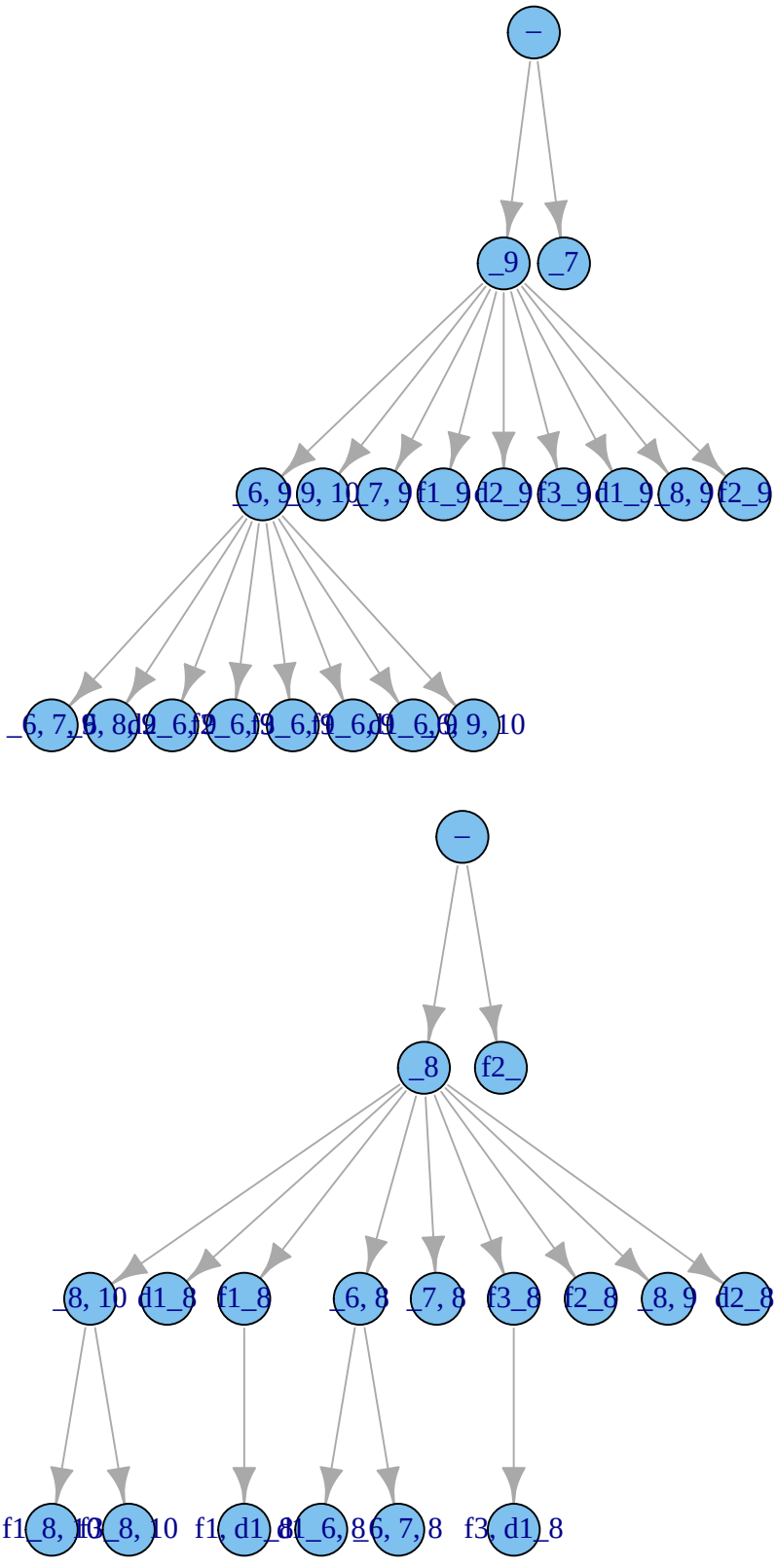
6.1 Phylogenies from multiple runs

If you use `oncoSimulPop` you can store and plot the phylogenies of the different runs:

```
oi <- allFitnessEffects(orderEffects =
               c("F > D" = -0.3, "D > F" = 0.4),
               noIntGenes = rexp(5, 10),
                           geneToModule =
                               c("Root" = "Root",
                                 "F" = "f1, f2, f3",
                                 "D" = "d1, d2") )
oiI1 <- oncoSimulIndiv(oi, model = "Exp")
oiP1 <- oncoSimulPop(4, oi,
                     keepEvery = 10,
                     mc.cores = 2,
                     keepPhylog = TRUE)
```

We will plot the first two:

```
op <- par(mar = rep(0, 4), mfrow = c(2, 1))
plotClonePhylog(oiP1[[1]])
plotClonePhylog(oiP1[[2]])
```



```
par(op)
```

This is so far disabled in function `oncoSimulSample`, since that function is optimized for other uses. This might change in the future.

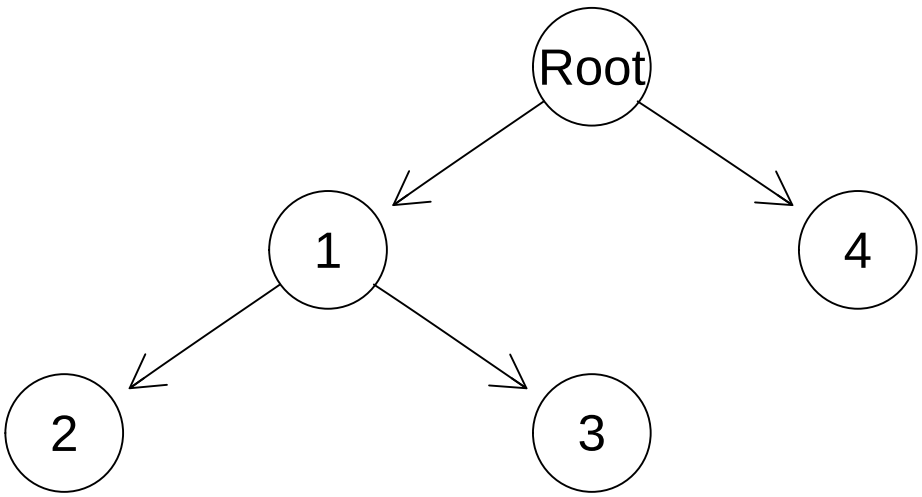
7 Using v.1 posets and simulations

It is strongly recommended that you use the new (v.2) procedures for specifying fitness effects. However, the former v.1 procedures are still available, with only very minor changes to function calls. What follows below is the former vignette. You might want to use v.1 because for certain models (e.g., small number of genes, with restrictions as specified by a simple poset) simulations might be faster with v.1 (fitness evaluation is much simpler —we are working on further improving speed).

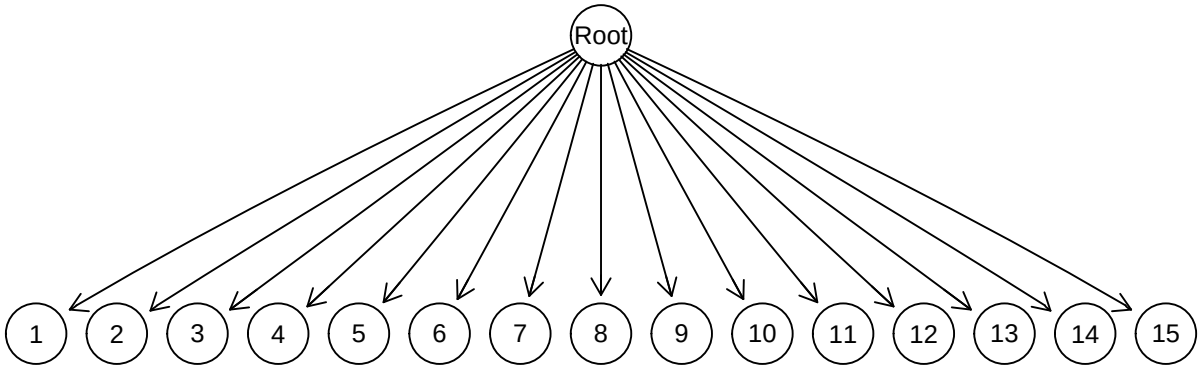
7.1 Specifying restrictions: posets

How to specify the restrictions is shown in the help for `poset`. It is often useful, to make sure you did not make any mistakes, to plot the poset. This is from the examples (we use an "L" after a number so that the numbers are integers, not doubles; we could alternatively have modified `storage.mode`).

```
## Node 2 and 3 depend on 1, and 4 depends on no one
p1 <- cbind(c(1L, 1L, 0L), c(2L, 3L, 4L))
plotPoset(p1, addroot = TRUE)
```

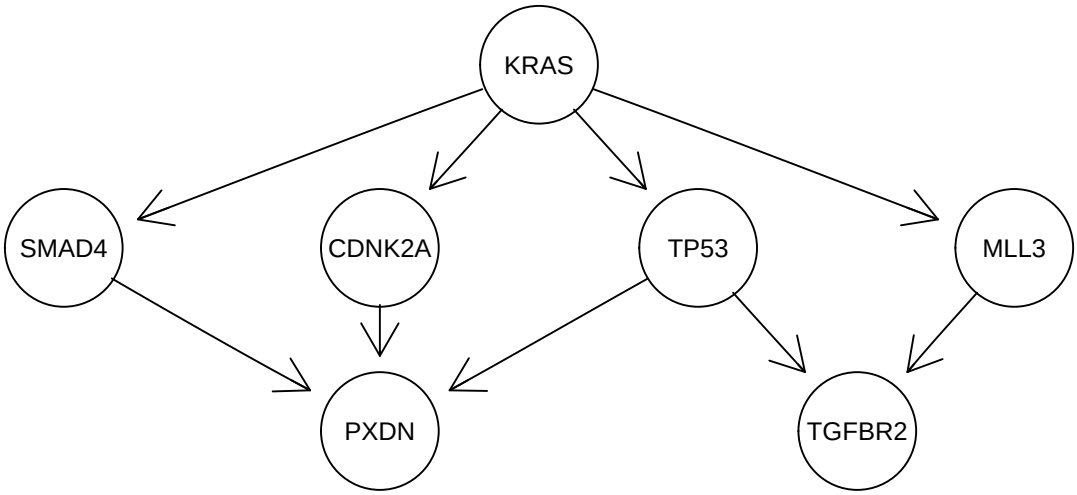



```
## A simple way to create a poset where no gene (in a set of 15) depends
## on any other.
p4 <- cbind(0L, 15L)
plotPoset(p4, addroot = TRUE)
```



Specifying posets is actually straightforward. For instance, we can specify the pancreatic cancer poset in Gerstung et al. [**?**] (their figure 2B, left). We specify the poset using numbers, but for nicer plotting we will use names (KRAS is 1, SMAD4 is 2, etc). This example is also in the help for `poset`:

```
pancreaticCancerPoset <- cbind(c(1, 1, 1, 1, 2, 3, 4, 4, 5),
                               c(2, 3, 4, 5, 6, 6, 6, 7, 7))
storage.mode(pancreaticCancerPoset) <- "integer"
plotPoset(pancreaticCancerPoset,
          names = c("KRAS", "SMAD4", "CDNK2A", "TP53",
                    "MLL3","PXDN", "TGFBR2"))
```



7.2 Simulating cancer progression

We can simulate the progression in a single subject. Using an example very similar to the one in the help:

```
## use poset p1101
data(examplePosets)
p1101 <- examplePosets[["p1101"]]

## Bozic Model
```

```
b1 <- oncoSimulIndiv(p1101, keepEvery = 15)
summary(b1)
```

```
##   NumClones TotalPopSize LargestClone MaxNumDrivers
## 1       221     33551822     19818466             4
##   MaxDriversLast NumDriversLargestPop TotalPresentDrivers
## 1              4                    1                   9
##   FinalTime NumIter HittedWallTime errorMF minDMratio
## 1       175   11607          FALSE      NA   18459.82
##   minBMratio          OccurringDrivers
## 1   24390.24 1, 2, 3, 4, 5, 6, 7, 8, 9
```

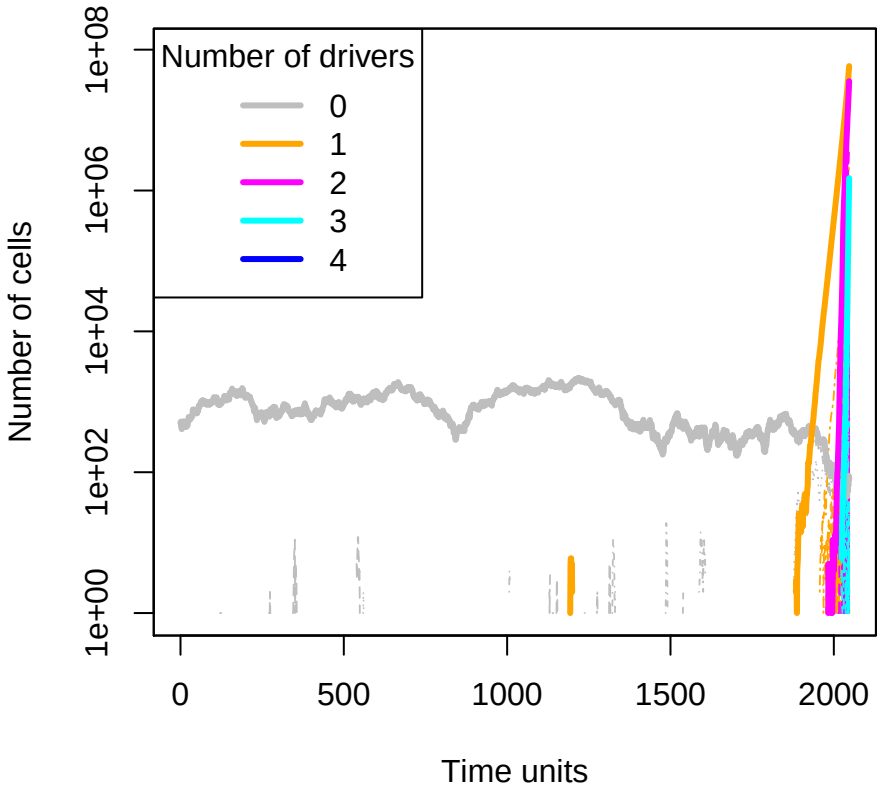
The first thing we do is make it simpler (for future examples) to use a set of restrictions. In this case, those encoded in poset p1101. Then, we run the simulations and look at a simple summary and a plot.

If you want to plot the trajectories, it is better to keep more frequent samples, so you can see when clones appear:

```
b2 <- oncoSimulIndiv(p1101, keepEvery = 1)

summary(b2)
```

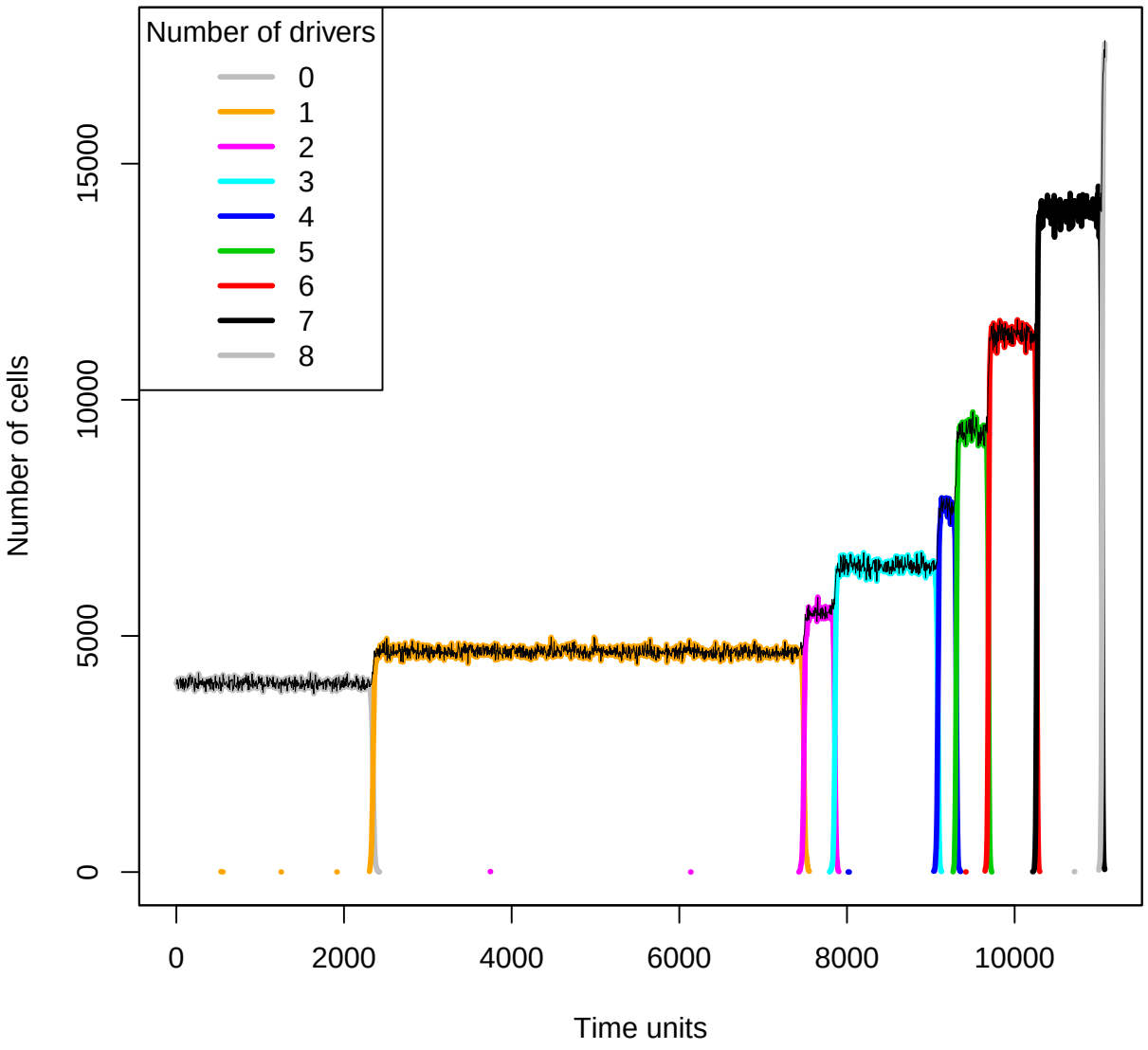
```
##   NumClones TotalPopSize LargestClone MaxNumDrivers
## 1       628     95607340     41565282             4
##   MaxDriversLast NumDriversLargestPop TotalPresentDrivers
## 1              4                    1                   9
##   FinalTime NumIter HittedWallTime errorMF minDMratio
## 1      2047   30370          FALSE      NA   18459.82
##   minBMratio          OccurringDrivers
## 1   24390.24 1, 2, 3, 4, 5, 6, 7, 8, 9
```

```
plot(b2)
```



The following is an example where we do not care about passengers, but we want to use a different graph, and we want a few more drivers before considering cancer has been reached. And we allow it to run for longer. Note that in the McF model `detectionSize` really plays no role. Note also how we pass the poset: it is the same as before, but now we directly access the poset in the list of posets.

```
m2 <- oncoSimulIndiv(examplePosets[["p1101"]], model = "McFL",
                     numPassengers = 0, detectionDrivers = 8,
                     mu = 5e-7, initSize = 4000,
                     sampleEvery = 0.025,
                     finalTime = 25000, keepEvery = 5,
                     detectionSize = 1e6)
plot(m2, addtot = TRUE, log = "")
```



The default is to simulate progression until a simulation reaches cancer (i.e., only simulations that satisfy the detectionDrivers or the detectionSize will be returned). If you use the McF model with large enough `initSize` this will often be the case but not if you use very small `initSize`. Likewise, most of the Bozic runs do not reach cancer. Lets try a few:

```
b3 <- oncoSimulIndiv(p1101, onlyCancer = FALSE)
summary(b3)
```

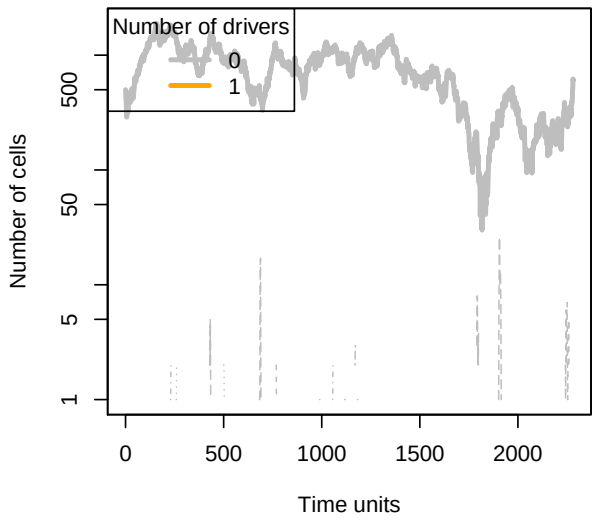
```
##   NumClones TotalPopSize LargestClone MaxNumDrivers
## 1        21          603          603             1
##   MaxDriversLast NumDriversLargestPop TotalPresentDrivers
## 1              0                    0                   1
##   FinalTime NumIter HittedWallTime errorMF minDMratio
## 1   2281.25    2347          FALSE      NA   22727.27
##   minBMratio OccurringDrivers
## 1   24390.24               7
```

```
b4 <- oncoSimulIndiv(p1101, onlyCancer = FALSE)
summary(b4)
```

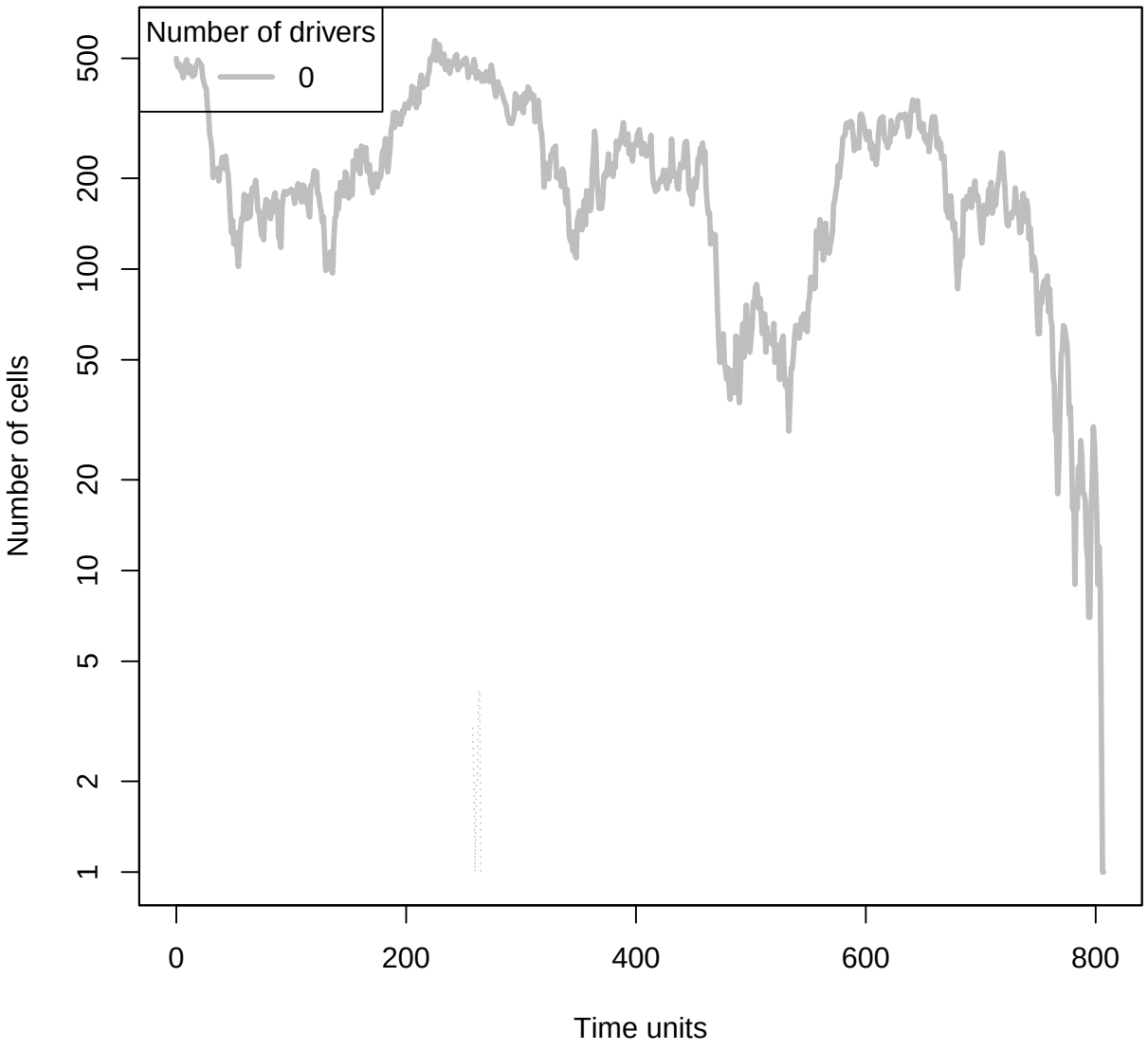
```
##   NumClones TotalPopSize LargestClone MaxNumDrivers
## 1         2            0            0             0
##   MaxDriversLast NumDriversLargestPop TotalPresentDrivers
## 1              0                    0                   0
##   FinalTime NumIter HittedWallTime errorMF minDMratio
## 1       808     812          FALSE      NA   22727.27
##   minBMratio OccurringDrivers
## 1   24390.24               NA
```

Plot those runs:

```
par(mfrow = c(1, 2))
par(cex = 0.8) ## smaller font
plot(b3)
```



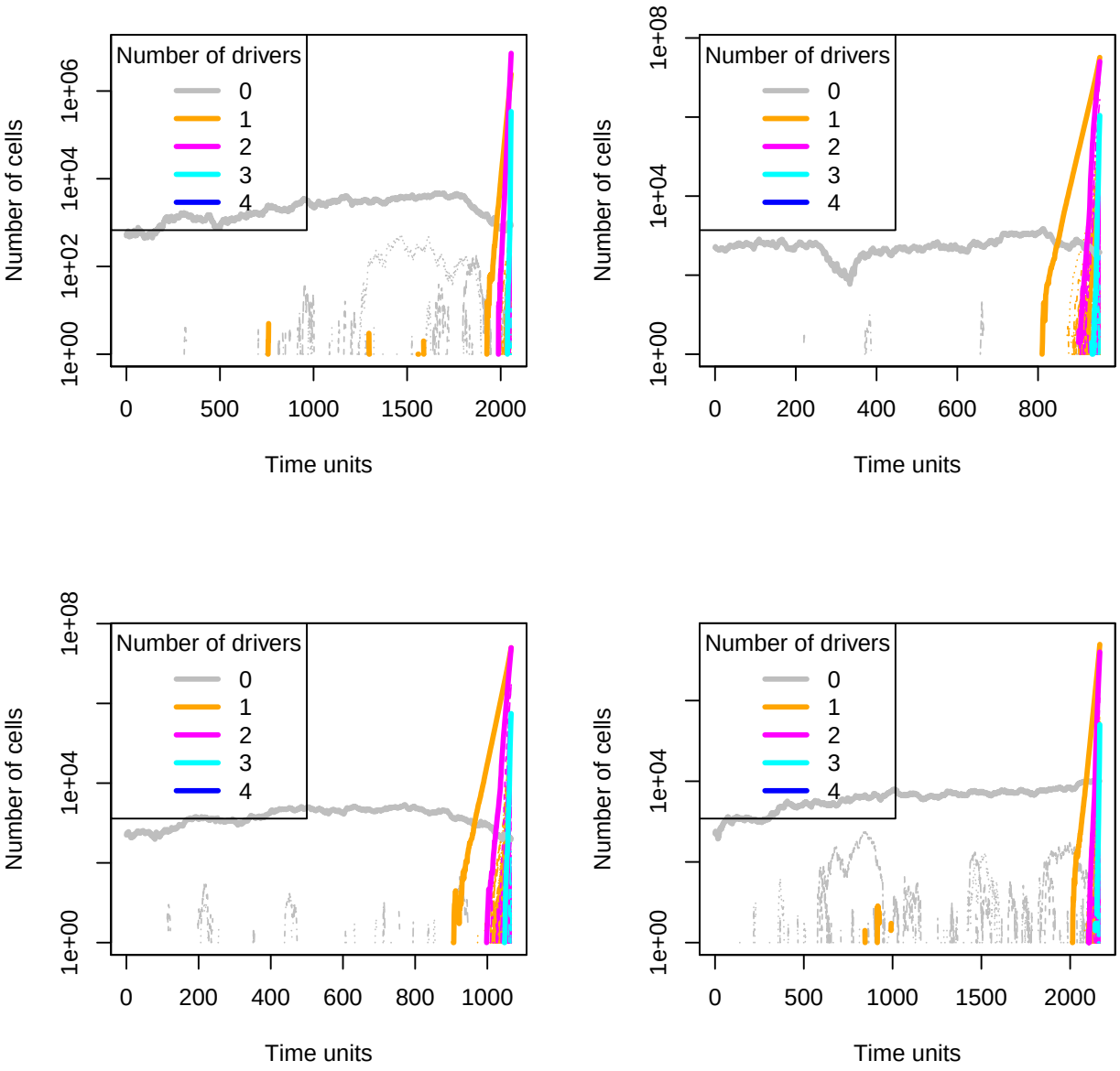

```
plot(b4)
```



7.2.1 Simulating progression in several subjects

To simulate the progression in a bunch of subjects (we will use only four, so as not to fill the vignette with plots) we can do, with the same settings as above:

```
p1 <- oncoSimulPop(4, p1101)
par(mfrow = c(2, 2))
plot(p1, ask = FALSE)
```



7.3 Sampling from a set of simulated subjects

You will often want to do something with the simulated data. For instance, sample the simulated data. Here we will obtain the trajectories for 100 subjects in a scenario without passengers. Then we will sample with the default options and store that as a vector of genotypes (or a matrix of subjects by genes):

```
m1 <- oncoSimulPop(100, examplePosets[["p1101"]],
                   numPassengers = 0)
```

The function samplePop samples that object, and also gives you some information about the output:

```
genotypes <- samplePop(m1)

##
##  Subjects by Genes matrix of 100 subjects and 11 genes.
```

What can you do with it? That is up to you. As an example, let us try to infer an oncogenetic tree (and plot it) using the *Oncotree* package [**?**] after getting a quick look at the marginal frequencies of events:

```
colSums(genotypes)/nrow(genotypes)
```

```
##    1    2    3    4    5    6    7    8    9   10   11
## 0.57 0.03 0.02 0.00 0.00 0.00 0.51 0.06 0.01 0.00 0.00
```

```
require(Oncotree)
```

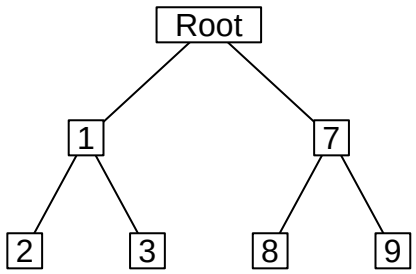
```
## Loading required package:  Oncotree
```

```
## Loading required package:  boot
```

```
ot1 <- oncotree.fit(genotypes)
```

```
## The following events had no observed occurances,so they will not be included in the constru
## 4 5 6 10 11
```

```
plot(ot1)
```



Your run will likely differ from mine, but with the defaults (detection size of 10^8) it is likely that events down the tree will never appear. You can set `detectionSize = 1e9` and you will see that events down the tree are now found in the cross-sectional sample.

Alternatively, you can use single cell sampling and that, sometimes, recovers one or a couple more events.

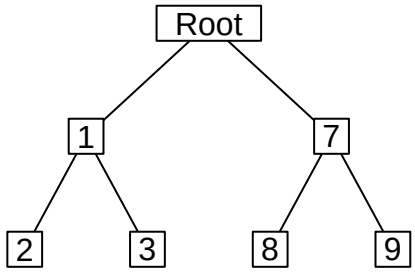
```
genotypesSC <- samplePop(m1, typeSample = "single")
```

```
##
##  Subjects by Genes matrix of 100 subjects and 11 genes.
```

```
colSums(genotypesSC)/nrow(genotypesSC)
```

```
##    1    2    3    4    5    6    7    8    9   10   11
## 0.63 0.10 0.12 0.00 0.00 0.00 0.62 0.13 0.12 0.00 0.00
```

```
ot2 <- oncotree.fit(genotypesSC)

## The following events had no observed occurances,so they will not be included in the constru
## 4 5 6 10 11

plot(ot2)
```



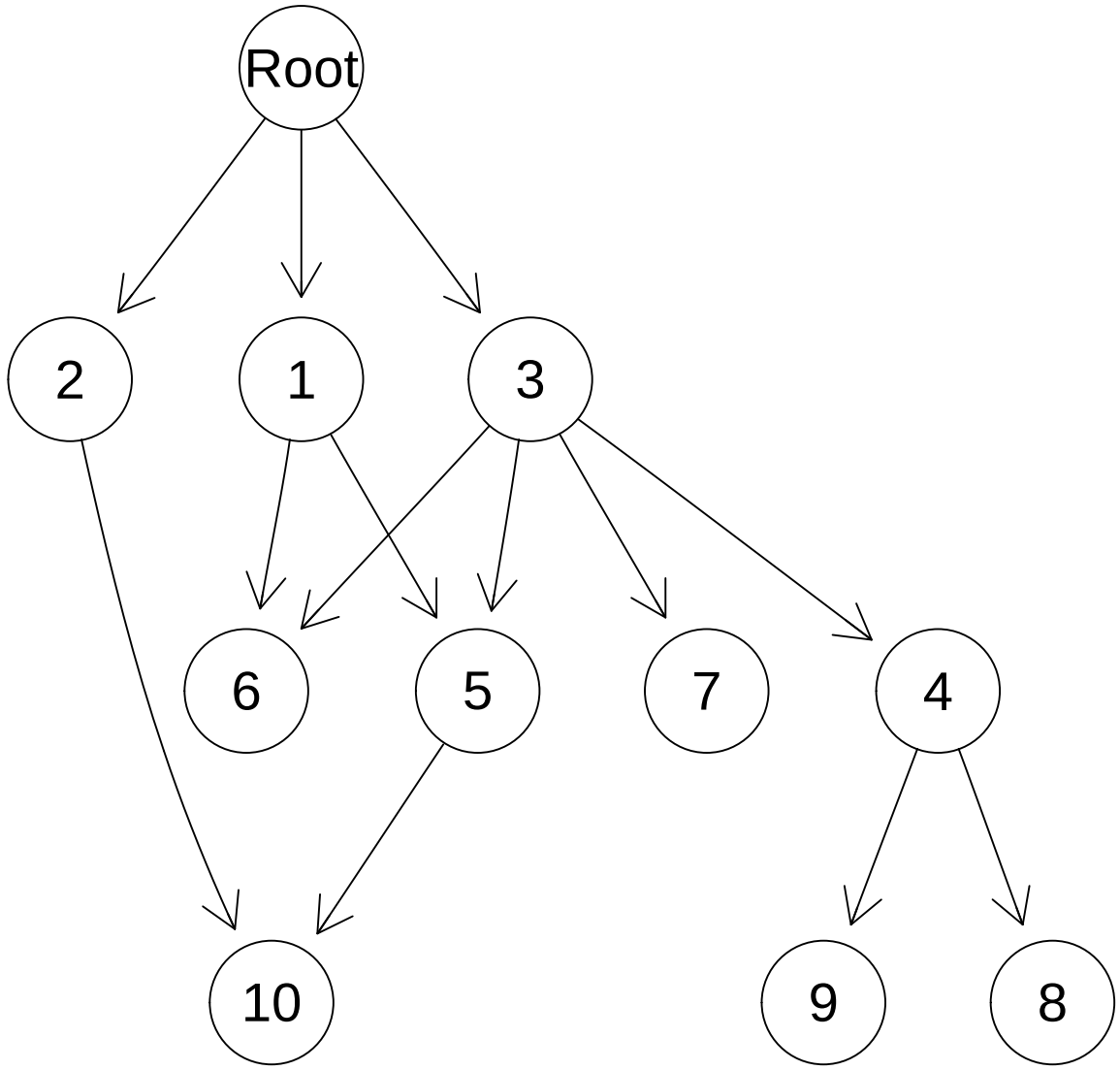
You can of course rename the columns of the output matrix to something else if you want so the names of the nodes will reflect those potentially more meaningful names.

8 Generating random DAGs for restrictions

You might want to randomly generate DAGs like those often found in the literature on oncogenetic trees et al. Function `simOGraph` might help here.

```
## No seed fixed, so reruns will give different DAGs.
(a1 <- simOGraph(10))

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

```
library(graph) ## for simple plotting
plot(as(a1, "graphNEL"))
```



Once you obtain the adjacency matrices, it is for now up to you to convert them into appropriate posets or fitnessEffects objects.

Why this function? I searched for, and could not find any that did what I wanted, in particular bounding the number of parents, being able to specify the approximate depth[8] of the graph, and optionally being able to have DAGs where no node is connected to another both directly (an edge between the two) and indirectly (there is a path between the two through other nodes). So I wrote my own code. The code is fairly simple to understand (all in file `generate-random-trees.R`). I would not be surprised if this way of generating random graphs has been proposed and named before; please let me know, best if with a reference.

[8]Where depth is defined in the usual way to mean smallest number of nodes —or edges— to traverse to get from the bottom to the top of the DAG.

Should we remove direct connections if there are indirect? Or, should we set `removeDirectIndirect = TRUE`? Except for [**?**], none of the DAGs I've seen in the context of CBNs, oncogenetic trees, etc, include both direct and indirect connections between nodes. If these exist, reasoning about the model can be harder. For example, with CBN (AND or CMPN or monotone relationships) adding a direct connection makes no difference iff we assume that the relationships encoded in the DAG are fully respected (e.g., all $s_h = -\infty$). But it can make a difference if we allow for deviations from the monotonicity, specially if we only check for the satisfaction of the presence of the immediate ancestors. And things get even trickier if we combine XOR with AND. The code for computing fitness, however, should deal with all of this just fine.

9 Session info and packages used

This is the information about the version of R and packages used:

```
sessionInfo()
```

```
## R version 3.2.4 (2016-03-10)
## Platform: x86_64-apple-darwin13.4.0 (64-bit)
## Running under: OS X 10.9.5 (Mavericks)
##
## locale:
## [1] C/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets
## [6] methods   base
##
## other attached packages:
## [1] Oncotree_0.3.3   boot_1.3-18      igraph_1.0.1
## [4] graph_1.48.0     OncoSimulR_2.0.1 knitr_1.12.3
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_0.12.4         lattice_0.20-33
##  [3] gtools_3.5.0        grid_3.2.4
##  [5] chron_2.3-47        stats4_3.2.4
##  [7] formatR_1.3         magrittr_1.5
##  [9] evaluate_0.8.3      highr_0.5.1
## [11] stringi_1.0-1       data.table_1.9.6
## [13] Rgraphviz_2.14.0    Matrix_1.2-4
## [15] BiocStyle_1.8.0     tools_3.2.4
## [17] stringr_1.0.0       parallel_3.2.4
## [19] BiocGenerics_0.16.1
```