BoolNet package vignette

Christoph Müssel, Martin Hopfensitz, Hans A. Kestler

May 20, 2010

Contents

1 Introduction

BoolNet is an R package that provides tools for assembling, analyzing and visualizing synchronous and asynchronous Boolean networks as well as probabilistic Boolean networks. This document gives an introduction to the usage of the software and includes examples for use cases.

BoolNet supports three types of networks:

Synchronous Boolean networks consist of a set of Boolean variables

$$X = \{X_1, \ldots, X_n\}$$

and a set of transition functions

$$F = \{f_1, \ldots, f_n\},$$

one for each variable. These transition functions map an input of the Boolean variables in X to a Boolean value (0 or 1). We call a Boolean vector $\mathbf{x}(t) = (x_1(t), \ldots, x_n(t))$ the *state* of the network at time t. Then, the next state of the network $\mathbf{x}(t{+}1)$ is calculated by applying *all* transition functions $f_i(\mathbf{x}(t))$.

In a biological context, genes can be modeled as Boolean variables (*active/expressed* or *inactive/not expressed*), and the transition functions model the dependencies among these genes. In the synchronous model, the assumption is that all genes are updated at the same time. This simplification facilitates the analysis of the networks.

Asynchronous Boolean networks have the same structure as synchronous Boolean networks. Yet, at each point of time t, only *one* of the transition functions $f_i \in F$ is chosen at random, and the corresponding Boolean variable is updated. This corresponds to the assumption that in a genetic network, gene expression levels are likely to change at different points of time. In the most common model, the gene to be updated is chosen uniformly among all genes. Moreover, BoolNet supports specifying non-uniform update probabilities for the genes.

Probabilistic Boolean networks (PBN) allow for specifying more than one transition function per variable/gene. Each of these functions has a probability to be chosen, where the probabilities of all functions for one variable sum up to 1. Formally

$$F = \{\{(f_{11}, p_{11}), \ldots, (f_{1k_1}, p_{1k_1})\}, \ldots, \{(f_{n1}, p_{n1}), \ldots, (f_{nk_n}, p_{nk_n})\}\}$$

where k_i is the number of alternative transition functions for variable i, and p_{ij} is the probability that function j is chosen for variable i. A state transition is performed by selecting one function for each gene based on the probabilities and applying the chosen functions synchronously.

In BoolNet, synchronous and asynchronous Boolean networks are modeled in the data structure *BooleanNetwork*, and probabilistic networks are modeled in the data structure *ProbabilisticBooleanNetwork*. The package provides several methods of constructing networks: Networks can be loaded from files in which human experts describe the dependencies between the genes. Furthermore, they can be reconstructed from time series of gene expression measurements. It is also possible to generate random networks. This can be helpful for the identification of distinct properties of biological networks by comparison to random structures. The different methods of assembling networks are described in Section˜2.

In Section˜3, tools for the analysis and visualization of network properties are introduced. For synchronous and asynchronous Boolean networks, the most important tool is the identification of attractors. Attractors are cycles of states and are assumed to be associated with the stable states of cell function. Another possibility of identifying relevant states is the included Markov chain simulation. This method is particularly suited for probabilistic networks and calculates the probability that a state is reached after a certain number of iterations. To test the robustness of structural properties of the networks to noise and mismeasurements, the software also includes extensive support for perturbing networks. In this way, it is possible to test these properties in noisy copies of a biological network.

In Section˜4, the interaction of `BoolNet` with related software is described. The necessary steps to import networks from BioTapestry and to export networks to Pajek are outlined.

For the examples in the following sections, we assume that the `BoolNet` package has been properly installed into the R environment. This can be done by typing

```
> install.packages("BoolNet")
```

into the R console or by the corresponding menu entries in an R GUI. For some of the plots, the `igraph` package is required and must be installed in your R environment as well. This is analogous to installing `BoolNet`. For the BioTapestry import, the `XML` package must be installed. Additionally, the `BoolNet` package must be loaded via

```
> library(BoolNet)
```

2 Assembling networks

2.1 Assembling a network from expert knowledge

A major advantage of Boolean networks is the fact that natural-language statements can easily be transferred into this representation. This allows researchers for building Boolean networks entirely from expert knowledge, for example by collecting statements on gene dependencies from literature and expressing them as Boolean rules.

`BoolNet` is able to read in networks consisting of such rule sets in a standardized text file format. In such a file, each line consists of a target gene and an update rule, usually separated by a comma. Optionally, it is also possible to add a probability for the rule if the file describes a probabilistic network. The first line of such a file is a header

```
targets, factors
```

or

```
targets, factors, probabilities
```

To illustrate the process of transforming natural-language statements into Boolean rules, we take a look at the mammalian cell cycle network introduced by Fauré et al. [5]. In Table 1 of this paper, the authors list natural-language statements of gene dependencies and the corresponding Boolean expressions. The following rules are taken from this table.
For gene CycD, Fauré et al. state:

> *CycD is an input, considered as constant.*

Transforming this into a Boolean rule is rather simple: CycD does not change its value, which means that its value after a transition only depends on its previous value. Thus, the transition rule is

```
CycD, CycD
```

Gene Rb has a more complex description:

> *Rb is expressed in the absence of the cyclins, which inhibit it by phosphorylation [...]; it can be expressed in the presence of CycE or CycA if their inhibitory activity is blocked by p27.*

As a general rule, inhibition can be represented by a Boolean negation. In the `BoolNet` file format, a negation is expressed by the `!` character. The referenced cyclins comprise the genes CycA, CycB, CycD, and CycE. If *all* these genes are absent, Rb is expressed – i.e. if CycA is not expressed *and* CycB is not expressed *and* CycD is not expressed *and* CycE is not expressed. A logical AND is embodied by the `&` character. Consequently, the first part of the rule is

```
 ! CycA & ! CycB & ! CycD & ! CycE
```

In combination with the above statement, the fact that Rb can be expressed in the presence of CycE and CycA if p27 is active means that CycB and CycD must not be active. Thus, the second part of the rule is

```
p27 & ! CycB & ! CycD
```

This statement is an exception (or alternative) to the first statement; this can be expressed as a logical OR, for which the `|` character is used.
The complete rule for gene Rb is thus

```
Rb, (! CycA & ! CycB & ! CycD & ! CycE) | (p27 & ! CycB & ! CycD)
```

After processing all genes in the table in this way, we get the following network description:

```
targets, factors
CycD, CycD
Rb, (! CycA & ! CycB & ! CycD & ! CycE) | (p27 & ! CycB & ! CycD)
E2F, (! Rb & ! CycA & ! CycB) | (p27 & ! Rb & ! CycB)
CycE, (E2F & ! Rb)
CycA, (E2F & ! Rb & ! Cdc20 & ! (Cdh1 & UbcH10)) | (CycA & ! Rb & ! Cdc20 & ! (Cdh1 & UbcH10))
p27, (! CycD & ! CycE & ! CycA & ! CycB) | (p27 & ! (CycE & CycA) & ! CycB &! CycD)
Cdc20, CycB
Cdh1,(! CycA & ! CycB) | (Cdc20) | (p27 & ! CycB)
UbcH10, ! Cdh1 | (Cdh1 & UbcH10 & (Cdc20 | CycA | CycB))
CycB, ! Cdc20 & ! Cdh1
```

Now save this description to a file "cellcycle.txt" in your R working directory. The network can be loaded via

```
> cellcycle <- loadNetwork("cellcycle.txt")
```

The same network is also included in `BoolNet` as an example and can be accessed via

```
> data(cellcycle)
```

As the above example does not cover all possibilities of the network description language, a full language specification is provided in Section~5.

2.2 Reconstructing a network from time series

An entirely different approach of assembling a network is to infer rules from series of expression measurements of the involved genes over time. For example, microarray experiments can be conducted at different points of time to cover the expression levels of different cell states. To reconstruct networks from such data, `BoolNet` includes two reconstruction algorithms, Best-Fit Extension~[9] and REVEAL [11]. REVEAL requires the inferred functions to match the input time series perfectly, hence it is not always able to reconstruct networks in the presence of noisy and inconsistent measurements. Best-Fit Extension retrieves a set of functions with minimum error on the input and is thus suited for noisy data.

In the following, we introduce a tool chain for the reconstruction of a Probabilistic Boolean Network from time series using Best-Fit extension.

Microarray measurements are usually represented as matrices of real-valued numbers which, for example, quantify the expression levels of genes. `BoolNet` includes a real-valued time series of gene measurements from a project to analyze the yeast cell cycle [14] which can be loaded using

```
> data(yeastTimeSeries)
```

This data contains four preselected genes and a series of 14 measurements for each of these genes.

In a first step, the real-valued dataset has to be converted to binary data as required by the reconstruction algorithm. `BoolNet` offers several binarization algorithms in the function binarizeTimeSeries(). We here employ the default method which is based on k-means clustering (with $k = 2$ for active and inactive):

```
> binSeries <- binarizeTimeSeries(yeastTimeSeries)
```

The returned structure in binSeries has an element $binarizedMeasurements containing the binary time series, and, depending on the chosen binarization method, some other elements describing parameters of the binarization.

To reconstruct the network from this data, we call the Best-Fit Extension algorithm:

```
> net <- reconstructNetwork(binSeries$binarizedMeasurements,
+ method="bestfit", maxK=4)
```

Here, maxK is the maximum number of input genes for a gene examined by the algorithm. The higher this number, the higher is the runtime and memory consumption of the reconstruction.

We can now take a look at the network using

```
> net

Probabilistic Boolean network with 4 genes

Involved genes:
Fkh2 Swi5 Sic1 Clb1

Transition functions:

Alternative transition functions for gene Fkh2:
Fkh2 = <f(Clb1){01}> (probability: 0.5, error: 1)
Fkh2 = <f(Fkh2){01}> (probability: 0.5, error: 1)

Alternative transition functions for gene Swi5:
Swi5 = <f(Clb1){01}> (probability: 0.5, error: 1)
Swi5 = <f(Fkh2){01}> (probability: 0.5, error: 1)

Alternative transition functions for gene Sic1:
Sic1 = <f(Sic1,Clb1){0001}> (probability: 0.3333333, error: 1)
Sic1 = <f(Swi5,Sic1){0001}> (probability: 0.3333333, error: 1)
Sic1 = <f(Fkh2,Sic1){0001}> (probability: 0.3333333, error: 1)

Alternative transition functions for gene Clb1:
Clb1 = <f(Clb1){01}> (probability: 0.5, error: 1)
Clb1 = <f(Fkh2){01}> (probability: 0.5, error: 1)
```

The dependencies among the genes in the network can be visualized using the plotNetworkWiring() function. In this graph, each gene corresponds to a vertex, and the inputs of transition functions correspond to edges.

```
> plotNetworkWiring(net)
```

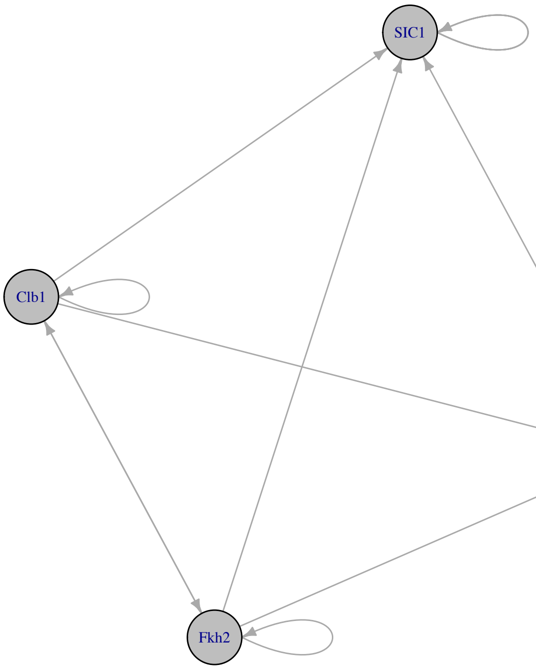


Figure 1: The wiring graph of the reconstructed network. Each node of the graph represents one gene, and each arrow represents a gene dependency.

plots a graph similar to that in Figure˜1. To use this function, you must install the `igraph` package.

A network that involved the same genes was examined by Kim et al. [8]. When comparing the wiring graph of our reconstructed network with the reference network presented in Figure˜2 of this paper, one observes a very high similarity between the two networks.

When `reconstructNetwork()` discovers multiple functions for a gene with the minimum error on the input data, it includes all of these functions as alternative functions with equal probability. Consequently, the function returns a `ProbabilisticBooleanNetwork` structure.

If you would like to obtain a `BooleanNetwork` object with only one function per gene from a probabilistic network, you can extract such a network by telling the software which of the functions you would like to use. For example,

```
> singleNet <- chooseNetwork(net, c(1,2,3,2))
```

creates a Boolean network by extracting the first function of gene Fkh2, the second function of genes Swi5 and Clb1, and the third function of gene Sic1 from the above probabilistic network:

```
> singleNet

Boolean network with 4 genes

Involved genes:
Fkh2 Swi5 Sic1 Clb1

Transition functions:
Fkh2 = <f(Clb1){01}>
Swi5 = <f(Fkh2){01}>
Sic1 = <f(Fkh2,Sic1){0001}>
Clb1 = <f(Fkh2){01}>
```

2.3 Creating random networks

To study structural properties of Boolean networks and to determine the specific properties of biological networks in comparison to arbitrary networks, it is often desirable to generate artificial networks. `BoolNet` comprises a facility for the generation of random N-K networks [6, 7]. In the standard N-K networks, N is the total number of genes, and K is the number of input genes for each gene transition function. Such a network can be generated using

```
> net <- generateRandomNKNetwork(n=10, k=3)
```

This creates a network with 10 genes, each of which has a transition function that depends on 3 genes and whose output is generated uniformly at random. Similarly, one can also specify different numbers of input genes for each gene:

```
> net <- generateRandomNKNetwork(n=10, k=c(1,2,3,1,3,2,3,2,1,1))
```

`BoolNet` does not only support this standard case, but allows for different methods of choosing the numbers of input genes (parameter `topology`), the input genes themselves (parameter `linkage`), and the transition functions (parameter `functionGeneration`). In the following, some examples are presented.
The command

```
> net <- generateRandomNKNetwork(n=20, k=20, topology="scale_free")
```

determines the numbers of input genes by drawing values from the scale-free Zeta distribution [1]. According to this distribution, most transition functions will have a small number of input genes, but a few transition functions may depend on a high number of genes. The shape of the Zeta distribution can be customized using an additional parameter `gamma`, which potentially increases the number of input genes when chosen small and vice versa.

```
> net <- generateRandomNKNetwork(n=10, k=3, linkage="lattice")
```

creates a network in which the transition functions of the genes depend on a choice of genes with adjacent indices [2]. This leads to networks with highly interdependent genes.

It is also possible to influence the truth tables of the functions by changing the ratio of 1 and 0 returned by the functions:

```
> net <- generateRandomNKNetwork(n=10, k=3,
+ functionGeneration="biased", zeroBias=0.75)
```

generates a network in which the outcome of a transition function is 0 for around 75% of the inputs.

By default, `generateRandomNKNetwork()` creates functions that cannot be simplified, i.e. that do not contain any genes that are irrelevant for the outcome of the function. If desired, this behaviour can be changed by setting `noIrrelevantGenes` to FALSE.

The presented parameters can be combined, and there are further options and parameters, so that a broad variety of networks with different structural properties can be generated. For a full reference of the possible parameters, please refer to the manual.

2.4 Knock-out and overexpression of genes

`BoolNet` allows for temporarily knocking out and overexpressing genes in a network without touching the transition functions. This means that genes can be set to a fixed value, and in any calculation on the network, this fixed value is taken instead of the value of the corresponding transition function. Knocked-out and overexpressed genes speed up the analysis of the network, as they can be ignored in many calculations. For example, to knock out CycD in the mammalian cell cycle network, we call

```
> data(cellcycle)
> knockedOut <- fixGenes(cellcycle, "cycd", 0)
```

or alternatively use the gene index

```
> knockedOut <- fixGenes(cellcycle, 1, 0)
```

This sets the gene constantly to 0. To over-express the gene (i.e. to fix it to 1), the corresponding call is

```
> overExpressed <- fixGenes(cellcycle, "cycd", 1)
```

The command

```
> originalNet <- fixGenes(knockedOut, "cycd", -1)
```

reactivates the gene (for both knock and overexpression) and resets the network to its original state.

The function also accepts multiple genes in a single call, such as

```
> newNet <- fixGenes(cellcycle, c("cycd","cyce"), c(0,1))
```

which knocks out CycD and overexpresses CycE.

3 Network analysis

3.1 Simulation of state transitions

To simulate a state transition and identify successor states of a given state, BoolNet includes the function stateTransition(). The function supports transitions for all three types of networks.

The following code performs a synchronous state transition for the state in which all genes are set to 1 on the mammalian cell cycle network:

```
> data(cellcycle)
> stateTransition(cellcycle, rep(1,10))
```

```
 [1] 1 0 0 0 0 0 1 1 1 0
```

A random asynchronous transition is performed using

```
> stateTransition(cellcycle, rep(1,10), type="asynchronous")
```

```
 [1] 1 1 1 1 0 1 1 1 1 1
```

In this case, the fifth gene, CycA, was chosen at uniformly at random and updated. We can also specify non-uniform probabilities for the genes, for example

```
> stateTransition(cellcycle, rep(1,10), type="asynchronous",
+ geneProbabilities=c(0.05,0.05,0.2,0.3,0.05,0.05,0.05,0.05,0.1,0.1))
```

```
 [1] 1 1 1 0 1 1 1 1 1 1
```

This obviously increases probabilities for the genes 3 and 4 (E2F and CycE) to be chosen. In this case, CycE was chosen for the update.

Sometimes you do not want a random update at all, but would like to specify which gene should be chosen for the update. This is possible via

```
> stateTransition(cellcycle, rep(1,10), type="asynchronous",
+ chosenGene="cyce")
```

```
 [1] 1 1 1 0 1 1 1 1 1 1
```

In probabilistic Boolean networks, a state transition is performed by choosing one of the alternative functions for each gene and applying this set of functions to the current state. The following performs a state transition with a randomly chosen set of functions on the artificial probabilistic Boolean network taken from [13] with 3 genes, starting from state (0,1,1):

```
> data(examplePBN)
> stateTransition(examplePBN, c(0,1,1), type="probabilistic")
```

```
[1] 1 0 0
```

You may get a different result, as the functions are chosen randomly according to the probabilities stored in the network. If you would like to execute a specific set of transition functions, you can supply this in an additional parameter:

```
> stateTransition(examplePBN, c(0,1,1), type="probabilistic",
+ chosenFunctions=c(2,1,2))

[1] 0 0 0
```

This call uses the second function for gene x1 and x3 and the first function for gene x2.

3.2 Identification of attractors

Attractors are stable cycles of states in a Boolean network. As they comprise the states in which the network resides most of the time, attractors in models of gene-regulatory networks are expected to be linked to phenotypes [7, 10]. Transitions from all states in a Boolean network eventually lead to an attractor, as the number of states in a network is finite. All states that lead to a certain attractor form its *basin of attraction*.

`BoolNet` is able to identify attractors in synchronous and asynchronous Boolean networks. There are three types of attractors in these networks:

Simple attractors occur in synchronous Boolean networks and consist of a set of states whose synchronous transitions form a cycle.

Complex or loose attractors are the counterpart of simple attractors in asynchronous networks. As there is usually more than one possible transition for each state in an asynchronous network, a complex attractor is formed by two or more overlapping loops. Precisely, a complex attractor is a set of states in which all asynchronous state transitions lead to another state in the set, and a state in the set can be reached from all other states in the set.

Steady-state attractors are attractors that consist of only one state. All transitions from this state result in the state itself. These attractors are the same both for synchronous and asynchronous update of a network. Steady states are a special case of both simple attractors and complex attractors.

The `getAttractors()` function incorporates several methods for the identification of attractors. We present these methods using the included mammalian cell cycle network as an example. This network has one steady-state attractor, one simple synchronous attractor consisting of 7 states, and one complex asynchronous attractor with 112 states (see [5]).

We first demonstrate the use of exhaustive synchronous search. This means that the software starts from all possible states of the network and performs synchronous state transitions until a simple or steady-state attractor is reached.

```
> data(cellcycle)
> attr <- getAttractors(cellcycle)
> attr

Attractor 1 is a simple attractor consisting of 1 state(s)
and has a basin of 512 state(s):
 |--<---------|
 V            |
 0100010100   |
 |            |
 V            |
 |-->---------|


Genes are encoded in the following order: cycd rb e2f cyce
cyca p27 cdc20 cdh1 ubch10 cycb

Attractor 2 is a simple attractor consisting of 7 state(s)
and has a basin of 512 state(s):

 |--<---------|
 V            |
 1001100000   |
 |            |
 1000100011   |
 |            |
 1000101011   |
 |            |
 1000001110   |
 |            |
 1010000110   |
 |            |
 1011000100   |
 |            |
 1011100100   |
 |            |
 V            |
 |-->---------|


Genes are encoded in the following order: cycd rb e2f cyce
cyca p27 cdc20 cdh1 ubch10 cycb
```

Typing `attr` calls a special print method that presents the attractor in a human-readable way. Here, a state in an attractor is represented by a binary vector, where each entry of the vector codes for one gene. An alternative is to print only the names of the active genes (i.e., the genes that are set to 1) instead of the full vector by calling the `print()` method explicitly with a changed parameter:

```
> print(attr, activeOnly=TRUE)

Attractor 1 is a simple attractor consisting of 1 state(s)
and has a basin of 512 state(s).
Active genes in the attractor state(s):
State 1: rb, p27, cdh1
```

```
Attractor 2 is a simple attractor consisting of 7 state(s)
and has a basin of 512 state(s).
Active genes in the attractor state(s):
State 1: cycd, cyce, cyca
State 2: cycd, cyca, ubch10, cycb
State 3: cycd, cyca, cdc20, ubch10, cycb
State 4: cycd, cdc20, cdh1, ubch10
State 5: cycd, e2f, cdh1, ubch10
State 6: cycd, e2f, cyce, cdh1
State 7: cycd, e2f, cyce, cyca, cdh1
```

We can see that the search identified both synchronous attractors. The advantage of the exhaustive search method is that the complete transition table is calculated and stored in the return value. This table stores information that is used by a number of analysis methods described below.

You can extract the transition table in a data frame and print it out using

```
> tt <- getTransitionTable(attr)
> tt
```

```
     State      Next state  Attr. basin  # trans. to attr.
0000000000 =>   0110010111            1                  4
[...]
1111111111 =>   1000001110            2                  1

Genes are encoded in the following order: cycd rb e2f cyce
cyca p27 cdc20 cdh1 ubch10 cycb
```

In the printed table, the first column denotes the initial state, the second column contains the state after the transition, the first column contains the number of the attractor that is finally reached from this state, and the fourth column lists the number of state transitions required to attain this attractor.

A table of the same structure is returned by

```
> getBasinOfAttraction(attr, 1)
```

which extracts all states from the transition table that belong to the basin of attraction of attractor one (i.e., whose attractor number in column 3 is 1).

If you are interested in information on a single state (here: the state with all genes set to 1), you can type

```
> getStateSummary(attr, c(1,1,1,1,1,1,1,1,1,1))
```

```
     State      Next state  Attr. basin  # trans. to attr.
1111111111 =>   1000001110            2                  1

Genes are encoded in the following order: cycd rb e2f cyce
cyca p27 cdc20 cdh1 ubch10 cycb
```

The visualization function getStateGraph() makes use of the transition table as well: It plots a transition graph in which the basins of attraction are drawn in different colors, and the attractors are highlighted. The result of

```
> plotStateGraph(attr)
```

is depicted in Figure˜2. The blue basin belongs to attractor 1, and the green basin belongs to attractor 2.

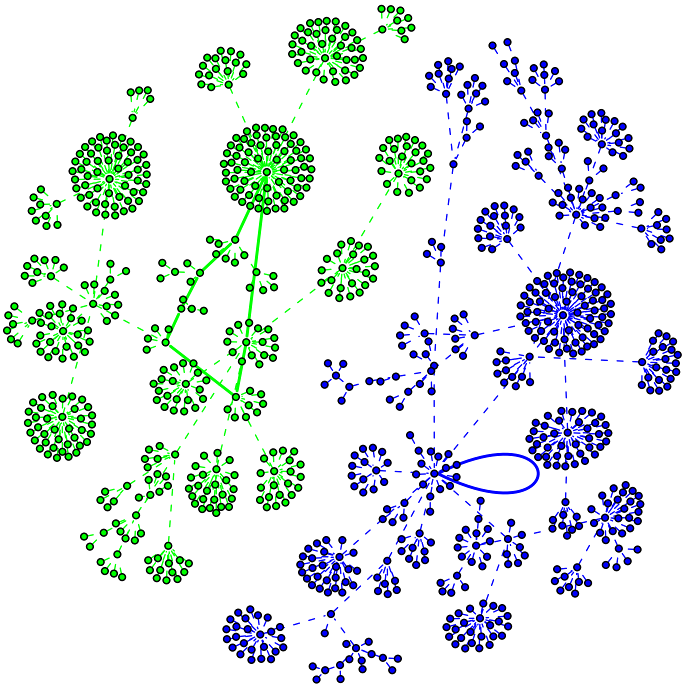


Figure 2: The state graph of the mammalian cell cycle network. Each node represents a state of the network, and each arrow is a state transition. The colors mark different basins of attraction. Attractors are highlighted using bold lines.

Exhaustive search consumes a high amount of time and memory with increasing size of the network, which makes it intractable for large networks (BoolNet currently supports networks with up to 29 genes for exhaustive search due to memory restrictions in *R*). Therefore, BoolNet also allows for heuristic search of attractors, which works for larger networks as well. Heuristic synchronous search starts from a predefined small set of states and identifies the attractors to which state transitions from these states lead. The start states can either be supplied, or they can be calculated randomly.

```
> attr <- getAttractors(cellcycle, method="chosen",
+ startStates=list(rep(0,10),rep(1,10)))
```

starts from the states (0,0,0,0,0,0,0,0,0,0) and (1,1,1,1,1,1,1,1,1,1) and again identifies both synchronous attractors.

```
> attr <- getAttractors(cellcycle, method="random", startStates=100)
```

chooses 100 random start states for the heuristic search and usually identifies both attractors as well.

For these two calls, only the subset of the transition table traversed by the heuristic is returned. This means that there is no guarantee that, e.g. `getBasinOfAttraction()` returns the complete basin of attraction of an attractor in heuristic mode.

Synchronous attractors can be visualized by plotting a table of changes of gene values in the states of the attractor:

```
> plotAttractors(attr,subset=2)
```

plots the state changes of the simple attractor with 7 states, as depicted in Figure˜3. Similarly,

```
> attractorsToLaTeX(attr,subset=2,file="attractors.tex")
```

exports the same state table to a LaTeX document.

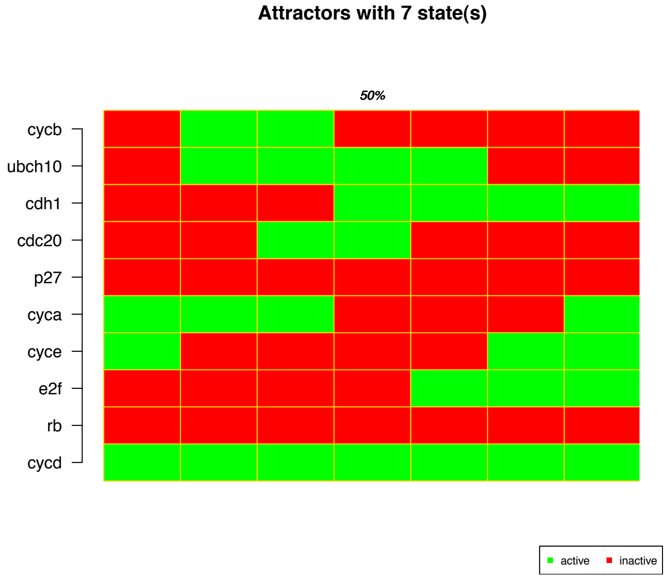


Figure 3: Visualization of the state changes in an attractor. The columns of the table represent consecutive states of the attractor. On top, the percentage of states leading to the attractor is supplied.

To identify asynchronous attractors, another special heuristic algorithm is included. This algorithm again starts from a small subset of states and makes a number of random transitions to reach an attractor with a high probability. After that, a validation step is performed to analyze whether a complex attractor has been identified. The command

```
> attr <- getAttractors(cellcycle, type="asynchronous",
+ method="random", startStates=500)
```

conducts an asynchronous search with 500 random start states on the mammalian cell cycle network. In this case, the algorithm has identified both the steady-state attractor and the complex attractor:

```
> attr
```

```
Attractor 1 is a simple attractor consisting of 1 state(s):

 |--<---------|
 V            |
 0100010100   |
 |            |
 V            |
 |-->---------|


Genes are encoded in the following order: cycd rb e2f cyce
cyca p27 cdc20 cdh1 ubch10 cycb

Attractor 2 is a complex/loose attractor consisting of 112 state(s)
and 338 transition(s):

1011101111 => 1011101110
[...]
1000000000 => 1010000000

Genes are encoded in the following order: cycd rb e2f cyce
cyca p27 cdc20 cdh1 ubch10 cycb
```

For the complex attractor, the involved transitions are printed out. By default, the algorithm tries to avoid self-loops, i.e. transitions that lead to the same state again. This means that self-loop transitions are only allowed if there is no other transition that leads to a different state. If you would like to allow the algorithm to enter self-loops even if transitions to different states are possible, you can call

```
> attr <- getAttractors(cellcycle, type="asynchronous",
+ method="random", startStates=500, avoidSelfLoops=FALSE)
```

In the resulting complex attractor with 112 states, there are 450 transitions instead of 338 transitions, which is due to the additional self-loops.

The asynchronous heuristic search does not return a transition table, such that the above analysis methods cannot be applied here.

As there are multiple possible transitions for each state, complex attractors cannot be visualized as in Figure˜3. For this reason, `plotAttractors()` supports a graph mode that visualizes the transitions among the states in the attractor:

```
> plotAttractors(attr,subset=2,mode="graph",drawLabels=FALSE)
```

plots the 112-state attractor as depicted in Figure˜4. We omit the state labels (i.e. the gene values) due to the high number of states. This plot again requires the `igraph` package.

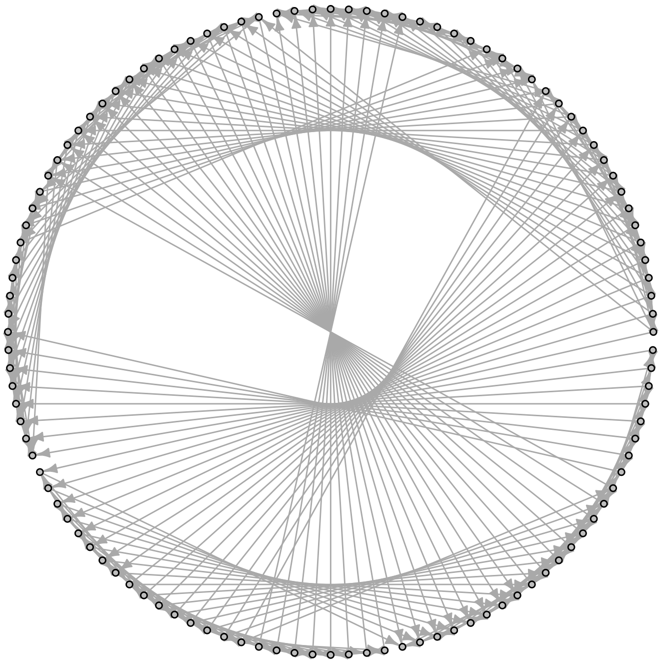


Figure 4: Graph representation of the complex attractor in the mammalian cell cycle network. Each node represents a state of the complex attractor, and each arrow represents a state transition.

3.3 Markov chain simulations

Another way of identifying relevant states in Boolean networks are Markov chain simulations. Instead of identifying cycles explicitly, these simulations calculate the probability that a certain state is reached after a predefined number of iterations. Of course, states in an attractor have a high probability of being reached if the number of iterations is chosen large enough. Markov chain simulations for probabilistic Boolean networks were introduced by Shmulevich et al. [13]. As a special case of probabilistic Boolean networks, these simulations are also suited for synchronous Boolean networks.

The following performs a Markov experiment with the predefined number of 1000 iterations on the example PBN described in [13]:

```
> data(examplePBN)
> sim <- markovSimulation(examplePBN)
> sim

States reached at the end of the simulation:
  x1 x2 x3 Probability
1  0  0  0        0.15
2  1  1  1        0.85

Probabilities of state transitions in the network:
  State     Next state  Probability
    000 =>         000          1.0
    001 =>         110          1.0
    010 =>         110          1.0
    011 =>         000          0.2
    011 =>         100          0.3
    011 =>         001          0.2
    011 =>         101          0.3
    100 =>         010          1.0
    101 =>         110          0.5
    101 =>         111          0.5
    110 =>         100          0.5
    110 =>         101          0.5
    111 =>         111          1.0
```

Only states with a non-zero probability are listed in the two tables. The first table shows the states that are reached after 1000 iterations. The second table is a transition table annotated with transition probabilities. This table can be suppressed by the parameter `returnTable=FALSE`. The results correspond exactly to those in [13].

If the transition table is included in the simulation results, we can plot a graph of the network:

```
> plotPBNTransitions(sim)
```

This graph is displayed in Figure~5. The vertices are the states of the graph. The edges represent transitions and are annotated with the corresponding transition probabilities. For this plot, the `igraph` package must be installed.

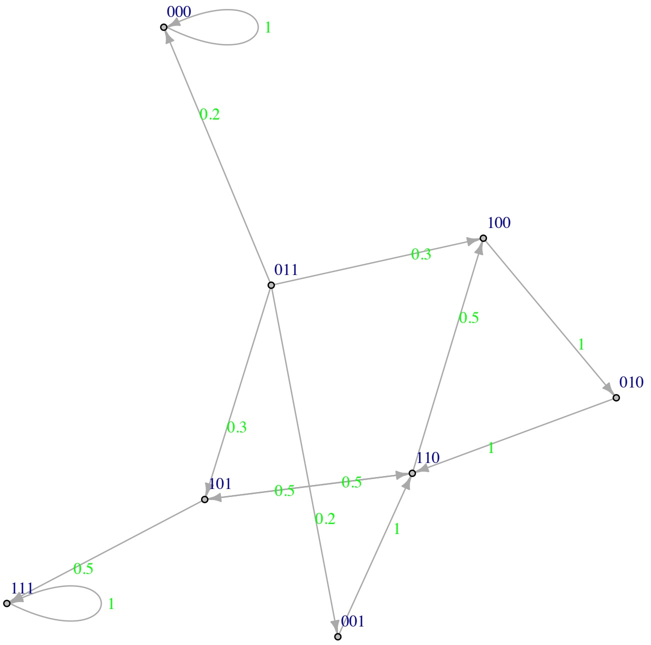


Figure 5: State transition graph of the example probabilistic Boolean network included in `BoolNet`. Each node represents a state of the network, and each arrow is a possible state transition, annotated by the transition probability.

We can also use Markov chain simulations to identify the attractor states in the mammalian cell cycle network:

```
> data(cellcycle)
> sim <- markovSimulation(cellcycle, numIterations=1024,
+ returnTable=FALSE)
> sim
```

```
States reached at the end of the simulation:
  cycd rb e2f cyce cyca p27 cdc20 cdh1 ubch10 cycb Probability
1    1  0   0    1    1   0     0    0      0    0  0.00781250
2    1  0   1    1    0   0     0    1      0    0  0.17187500
3    1  0   1    1    1   0     0    1      0    0  0.02343750
4    0  1   0    0    0   1     0    1      0    0  0.50000000
5    1  0   1    0    0   0     0    1      1    0  0.15625000
6    1  0   0    0    0   0     1    1      1    0  0.10937500
7    1  0   0    0    1   0     0    0      1    1  0.00390625
8    1  0   0    0    1   0     1    0      1    1  0.02734375
```

We set the maximum number of iterations to 1024, which is the number of states in the network. In a deterministic network, this guarantees that all states are found.

The fourth state in the returned table is the steady-state attractor identified previously. It has a probability of 0.5, as the basin of attraction is exactly half of the states. The other 7 states belong to the simple synchronous attractor.

It is also possible to restrict the simulation to a certain set of input states instead of using all possible input states. In the following example, we only consider the state with all genes set to 1, and identify the state belonging to the steady-state attractor again:

```
> sim <- markovSimulation(cellcycle, numIterations=1024,
+ returnTable=FALSE, startStates=list(rep(1,10)))
> sim

States reached at the end of the simulation:
  cycd rb e2f cyce cyca p27 cdc20 cdh1 ubch10 cycb Probability
1    1  0   1    0    0   0     0    1      1    0           1
```

3.4 Perturbation experiments

The generation of perturbed copies of a network is a way to test the robustness of structural properties of the networks to noise and mismeasurements. For example, you could assess the relevance of an attractor by checking whether the same attractor is still found when small random changes are applied to the network. If this is the case, it is less likely that the attractor is an artifact of mismeasurements.

BoolNet includes a set of different perturbation options that can be combined. For example,

```
> data(cellcycle)
> perturbedNet <- perturbNetwork(cellcycle, perturb="functions",
+ method="bitflip")
```

chooses a function of the network at random and flips a single bit in this function. By setting the parameter maxNumBits, you can also flip more than one bit at a time.

Instead of flipping bits,

```
> perturbedNet <- perturbNetwork(cellcycle, perturb="functions",
+ method="shuffle")
```

randomly permutes the output values of the chosen transition functions. This preserves the numbers of 0s and 1s, but may change the Boolean function completely. These kinds of perturbations are supported for synchronous and asynchronous networks as well as for probabilistic networks.

For synchronous networks, a further perturbation mode is available:

```
> perturbedNet <- perturbNetwork(cellcycle, perturb="states",
+ method="bitflip", numStates=10)
```

Here, BoolNet calculates the complete transition table of the network and then flips a single bit in 10 states of the transition table. From this modified table, a network is reconstructed. Changes of this type only affect a few states (which might not be the case when perturbing the functions directly as above), but possibly several of the transition functions. As in the previous examples, it is also possible to modify the number of bits to be flipped or to choose `method="shuffle"`.

A detailed listing of perturbation experiments is shown on page~22. In this experiment, 1000 perturbed copies of the cell cycle network are created, and the occurrences of the original synchronous attractors are counted in the perturbed copies.

The results of such an experiment could look like this:

```
Attractors in original network:
Attractor 1 is a simple attractor consisting of 1 state(s)
and has a basin of 512 state(s):

 [...]

Attractor 2 is a simple attractor consisting of 7 state(s)
and has a basin of 512 state(s):

[...]

Number of occurrences of the original attractors in 1000
perturbed copies of the network:
Attractor 1: 622
Attractor 2: 589
```

We see that the steady-state attractor is slightly more robust to perturbations than the simple attractor with 7 states, as it can be identified in a higher number of perturbed copies.

```
> # Perform a robustness test on a network
> # by counting the numbers of perturbed networks
> # containing the attractors of the original net
>
> library(BoolNet)
> # load mammalian cell cycle network
> data(cellcycle)
> # get attractors in original network
> attrs <- getAttractors(cellcycle, canonical=TRUE)
> # create 1000 perturbed copies of the network and search for attractors
> perturbationResults <- sapply(1:1000,function(i)
+ {
+   # perturb network and identify attractors
+   perturbedNet <- perturbNetwork(cellcycle, perturb="functions", method="bitflip")
+   perturbedAttrs <- getAttractors(perturbedNet, canonical=TRUE)
+
+   # check whether the attractors in the original network exist in the perturbed network
+   attractorIndices <- sapply(attrs$attractors,function(attractor1)
+         {
+           index <- which(sapply(perturbedAttrs$attractors,function(attractor2)
+             {
+               identical(attractor1,attractor2)
+             }))
+           if (length(index) == 0)
+             NA
+           else
+             index
+         })
+   return(attractorIndices)
+ })
> # perturbationResults now contains a matrix
> # with the first 2 columns specifying the indices or the
> # original attractors in the perturbed network
> # (or NA if the attractor was not found) and the next 2
> # columns counting the numbers of states
> # in the basin of attraction (or NA if the attractor was not found)
>
> # measure the total numbers of occurrences of the original attractors in the perturbed copies
> numOccurrences <- apply(perturbationResults[1:length(attrs$attractors),,drop=FALSE], 1,
+                         function(row)sum(!is.na(row)))
> # print original attractors
> cat("Attractors in original network:\n")
> print(attrs)
> # print information
> cat("Number of occurrences of the original attractors",
+         "in 1000 perturbed copies of the network:\n")
> for (i in 1:length(attrs$attractors))
+ {
+   cat("Attractor ",i,": ",numOccurrences[i],"\n",sep="")
+ }
```

3.5 Identifying specific properties of biological networks

The described perturbations could also be used to identify specific properties of real-world networks in comparison to arbitrary (random) networks. For example, one could assume that attractors in biological networks are more robust to perturbations than attractors in random networks with a similar structure, as they should be capable of compensating for small dysfunctions of their components. Similarly to the above code, one could execute a number of random perturbations on the biological network and measure the percentage of original attractors found in the perturbed copies. Afterwards, one could repeat this process on a number of randomly generated networks – i.e., generate perturbed copies from each of the random networks, and measure the percentage of attractors in the copies. If the percentage of the biological network is higher than most of the percentages of the random network, this suggests that the biological network exhibits a higher robustness. This is a kind of computer-intensive test.

BoolNet comprises a generic facility for such computer-intensive tests. This facility already includes two tests for synchronous Boolean networks and can be extended by custom test functions. The outlined example of attractor robustness is one of the integrated functions:

```
> data(cellcycle)
> testNetworkProperties(cellcycle, numRandomNets=100,
+ testFunction="testAttractorRobustness",
+ testFunctionParams = list(copies=100))
```

creates a set of 100 random networks (each with the same number of input genes for the functions as the cell cycle network) and creates 100 perturbed copies for each of these networks and for the cell cycle network. It then measures the percentages of found attractors and plots an histogram of the percentages of the random networks (see Figure~6). The percentage of the cell cycle network is plotted as a red line, and the 95% quantile is plotted as a blue line.

We can see that the average percentage of found attractors is significantly higher in the biological network with a p-value of 0.03.

A second network property can be tested using a built-in function: When looking at the state graph of a biological network (which can be generated using plotStateGraph()), it can often be observed that many state transitions lead to the same successor states, which means that the dynamics of the network quickly concentrate on a few states after a number of state transitions. We call the number of states whose synchronous state transitions lead to a state s the *in-degree* of state s. We expect the biological network to have a few states with a high in-degree and many states with a low in-degree. A characteristic to summarize the in-degrees is the Gini index, which is a measure of inhomogeneity. If all states have an in-degree of 1, the Gini index is 0; if all state transitions lead to only one state, the Gini index is 1.

```
> testNetworkProperties(cellcycle, numRandomNets=100,
+ testFunction="testIndegree")
```

plots an histogram of Gini indices in 100 random networks and draws the Gini index of the cell cycle network as a red line, as depicted in Figure~7.

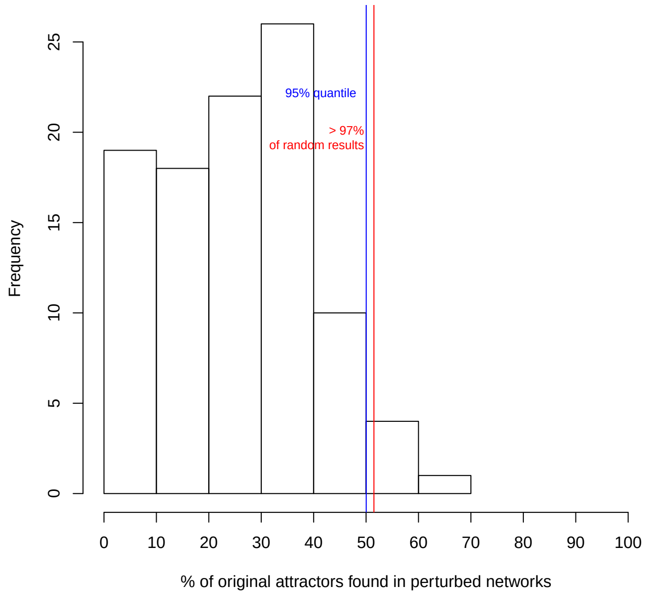


Figure 6: Attractor robustness of randomly generated networks (histogram) in comparison to the mammalian cell cycle network (red line).

The histogram shows that the Gini index of the in-degrees is *always* higher in the biological network. This is probably due to the special structure of functions in biological networks.

Instead of accumulating the in-degrees using the Gini index, it is also possible to compare the distributions of the in-degrees across the networks. For this purpose, the Kullback-Leibler distances of the in-degrees of the supplied network and each of the random networks are calculated and plotted in a histogram. The Kullback-Leibler distance (also called relative entropy) is an asymetric measure of similarity of two distributions [4]. If the distributions are equal, the Kullback-Leibler distance is 0, otherwise it is greater than 0.

```
> testNetworkProperties(cellcycle, numRandomNets=100,
+ testFunction="testIndegree", accumulation="kullback_leibler")
```

results in the plot displayed in Figure˜8.

It is possible to switch between the histogram of an accumulated characteristic (e.g. the Gini index) and the histogram of the Kullback-Leibler distances for all tests.

You can also easily implement your own tests. To do this, the only thing you have to do is implement a custom testing function that replaces `testIndegree()` or `testAttractorRobustness`. Testing functions have the following signature:

```
function(network, accumulate=TRUE, params)
```

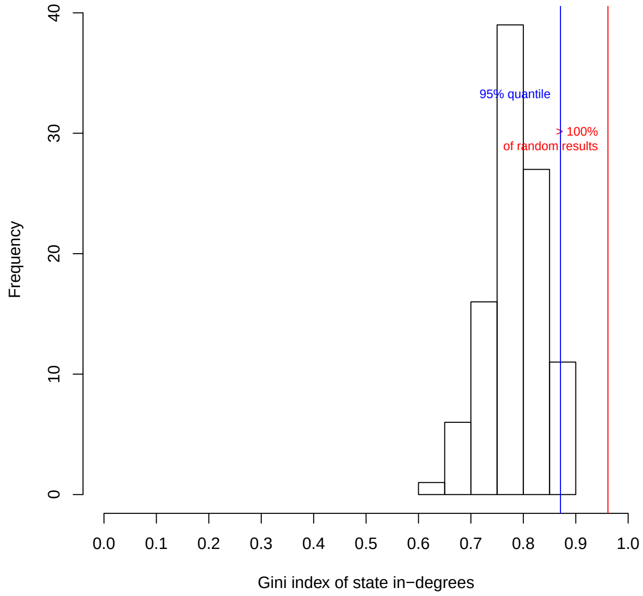


Figure 7: Gini indices of state in-degrees of randomly generated networks (histogram) in comparison to the mammalian cell cycle network (red line)

The first parameter is the network that should be tested. The parameter `accumulate` specifies whether a single characteristic value (e.g., the Gini index of the in-degrees) should be calculated, or whether a distribution of values (e.g., a vector of all in-degrees) should be returned. The third parameter is a list of further arguments needed by your function.

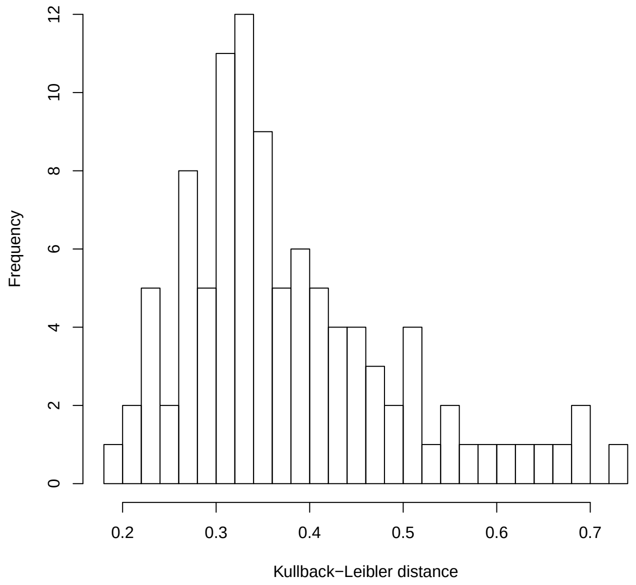


Figure 8: Kullback-Leibler distances of in-degrees of the mammalian cell cycle network and 100 random networks.

If, for example, we would like to compare the sizes of the basins of attractions of synchronous attractors in biological and random networks, we would write a function like this:

```
> testBasinSizes <- function(network, accumulate=TRUE, params)
+ {
+   attr <- getAttractors(network)
+   basinSizes <- sapply(attr$attractors, function(a)
+                       {
+                          a$basinSize
+                       })
+    if (accumulate)
+      return(mean(basinSizes))
+    else
+      return(basinSizes)
+ }
```

This function calculates the mean basin size as a characteristic value if accumulation is required, or returns the sizes of all basins of attraction in a vector otherwise. It does not need any further parameters in `params`.

Now, we can start a test using

```
> testNetworkProperties(cellcycle, numRandomNets=1000,
+ testFunction="testBasinSizes",
+ xlab="Average size of basins of attraction")
```

to produce the plot shown in Figure˜9. Apparently, the average basin sizes do not differ as much as the built-in test characteristics between the random networks and the cell cycle network.

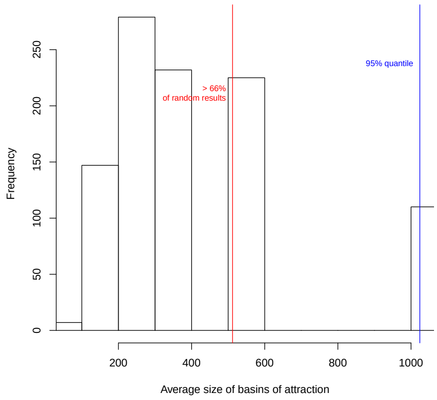


Figure 9: A custom test statistic measuring the basin sizes on randomly generated networks (histogram) and the mammalian cell cycle network (red line).

By writing custom test functions, you can extend the test facility to perform a wide variety computer-intensive test. Of course, it is also possible to plot the Kullback-Leibler distances with such new methods by using `accumulation="kullback_leibler"`.

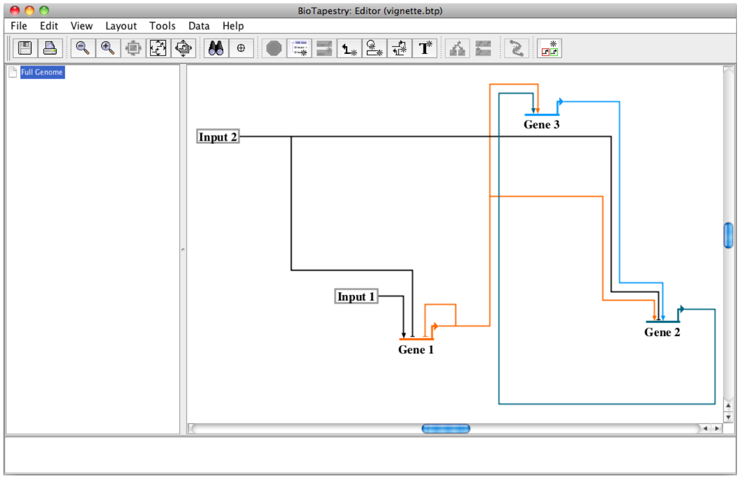
`testNetworkProperties()` accepts most of the parameters of `generateRandomNKNetwork()`. If necessary, you can generate more specialized kinds of random networks which resemble the original network in certain aspects, for example by setting a proportion of 0 and 1 in the function outputs similar to the original network using `functionGeneration="biased"`.

4 Import and export

4.1 Importing networks from BioTapestry

BioTapestry is a widely-used application for visual modeling of gene-regulatory networks [12]. It can be freely accessed at `http://www.biotapestry.org`. Although its primary purpose is visualization, the software supports specifying logical functions for the genes. `BoolNet` can read in the top-level ("Full genome") plot of a BioTapestry file (*.btp) and convert it into a Boolean network.

As an example, we assume the following BioTapestry model with 5 genes (2˜inputs and 3˜dependent genes):



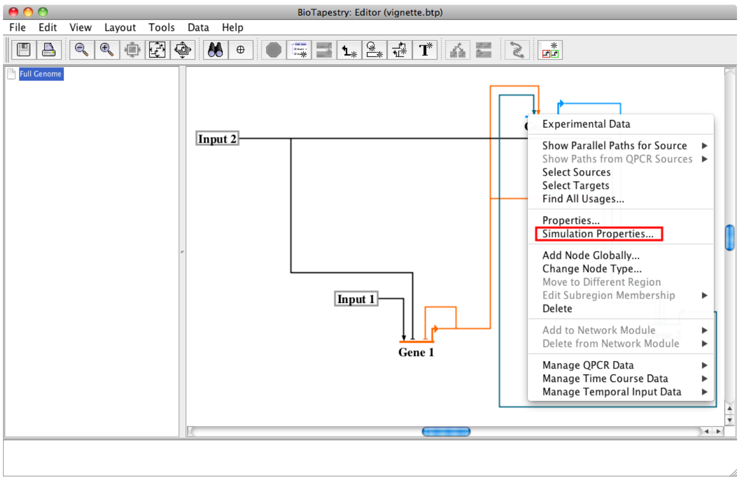
The corresponding BioTapestry file is included in `BoolNet`. You can determine its path using

```
> system.file("doc/example.btp", package="BoolNet")
```

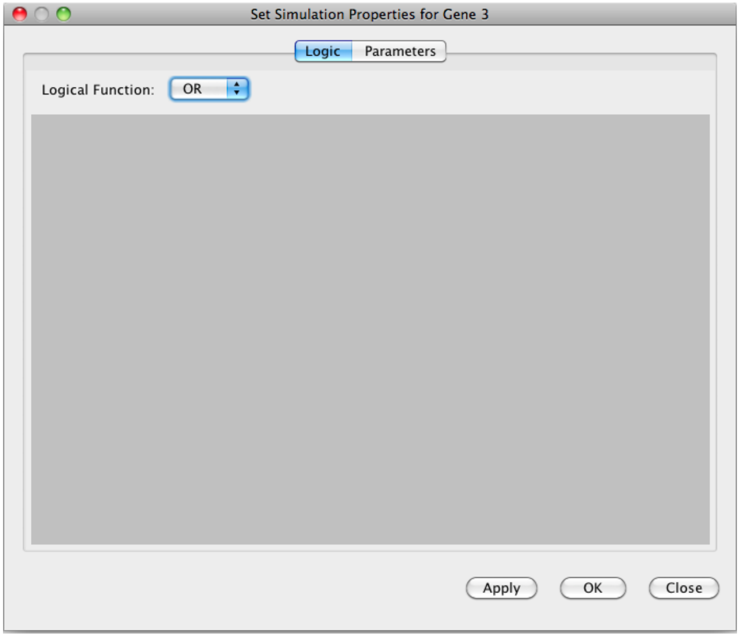
to access it in BioTapestry or `BoolNet`.

For the import, `BoolNet` needs to know the type of influence a gene has on another gene. Therefore, imported networks should only use links that are either enhancers or repressors. Neutral links are ignored in the import.

We now set further simulation parameters for the model. These parameters are imported by `BoolNet` to construct the functions of the Boolean network. First, we want to change the function of Gene˜2 to `OR`. Right-click on Gene˜2 and choose `Simulation Properties...`.



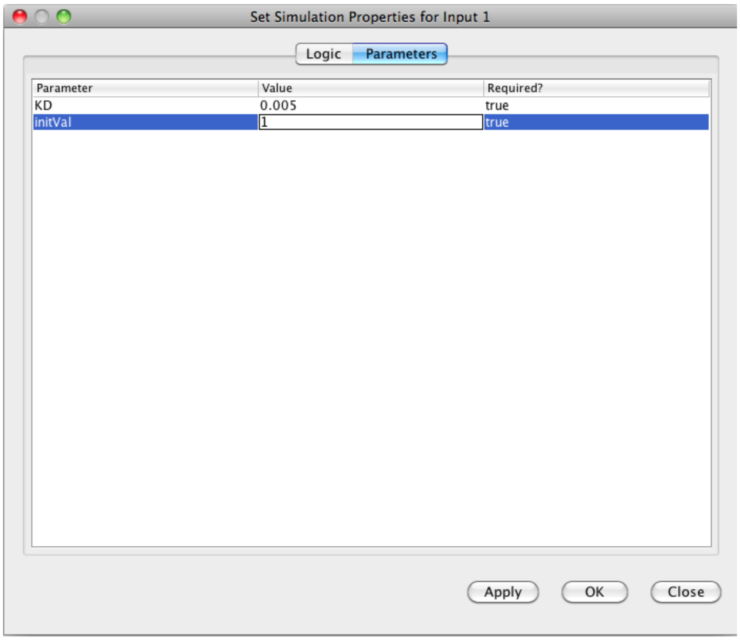
In the properties dialog, choose the `Logic` tab, and select `OR` for the logical function.


Set Simulation Properties for Gene 3

Logical Function: OR

Now set the function of Gene˜1 to `XOR` (exclusive or) in the same way.

You can also specify initial values for constant genes, i.e., genes with no input links. Choose the simulation properties of Input˜1, and change to the `Parameters` tab. Choose `initVal` and set it to 1.



Parameter	Value	Required?
KD	0.005	true
initVal	1	true

Press Return to store the result, and exit the dialog with `OK`. This will create a fixed gene with value 1 (i.e., an over-expressed gene) in the `BoolNet` import. Note that values other than 0 and 1 are ignored by the import, as well as initialization values for non-constant genes.

We assume that you save the network to a file "example.btp" in your working directory. In `R`, type

```
> net <- loadBioTapestry("example.btp")
```

to import the network. Alternatively, replace the file name by the command on page˜28 to use the file in the package if you do not want to create the file yourself.

The imported network looks like this:

```
> net

Boolean network with 5 genes

Involved genes:
Input 1 Input 2 Gene 1 Gene 2 Gene 3

Transition functions:
Input 1 = 1
Input 2 = Input 2
Gene 1 = (!Gene 1 & !Input 1 & Input 2) | (!Gene 1 & Input 1 & !Input 2)
         | (Gene 1 & !Input 1 & !Input 2) | (Gene 1 & Input 1 & Input 2)
```

```
Gene 2 = Gene 1 & Gene 3 & !Input 2
Gene 3 = Gene 1 | Gene 2

Knocked-out and over-expressed genes:
Input 1 = 1
```

We can see that Input˜1 is specified as an over-expressed constant gene. Input˜2 is modeled as depending only on itself, i.e. it keeps its initial value. Gene˜1 is a representation of the XOR function in Disjunctive Normal Form (DNF), using only logical ANDs, logical ORs, and negations. Gene˜2 and Gene˜3 consist of conjunctions and disjunctions of their inputs respectively. In addition to this textual description, we can visually verify the network by plotting its wiring:

```
> plotNetworkWiring(net)
```

The resulting plot is shown in Figure˜10.

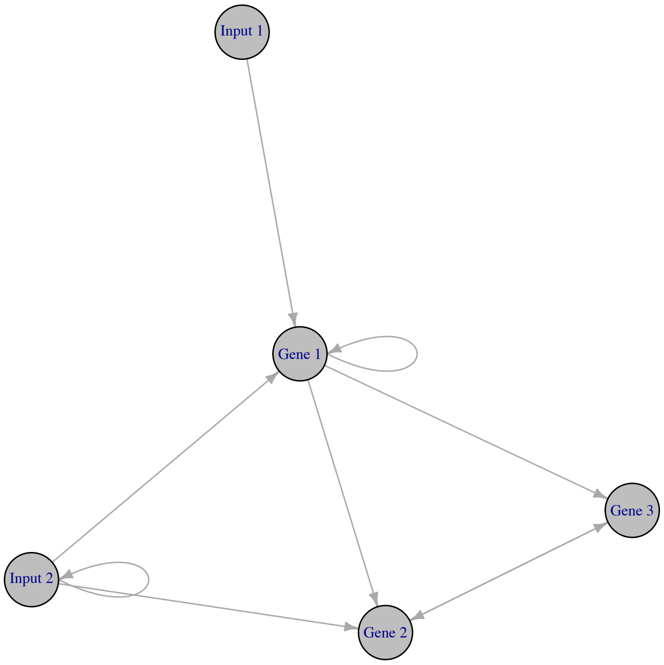


Figure 10: The wiring graph of the imported network specified in BioTapestry.

You can now use the imported network just like any other network in `BoolNet`.

4.2 Exporting networks to Pajek

For further analysis, networks can be exported to Pajek, a Windows application that provides visualization and analysis methods for graph structures [3]. For more information on Pajek, please refer to `http://pajek.imfm.si/doku.php`.

The export function writes the state transition graph to a Pajek file (*.net). This requires a synchronous exhaustive attractor search in `BoolNet` to build the full transition table.

To export the mammalian cell cycle network to Pajek, call

```
> data(cellcycle)
> attr <- getAttractors(cellcycle)
> toPajek(attr, file="cellcycle.net")
```

This will export the graph of the state transitions, which is usually sufficient for plotting. If you want to include the state information (i.e., the gene assignment vectors), call

```
> toPajek(attr, file="cellcycle.net", includeLabels=TRUE)
```

Now, start Pajek, load the network with `File | Network | Read`, and check out the tools provided by this application. For example, visualizations can be accessed using the menu item `Draw | Draw`.

Figure˜11 shows a plot of the cell cycle network with the Kamada-Kawai layout (Menu entry `Layout | Energy | Kamada-Kawai | Separate Components`).

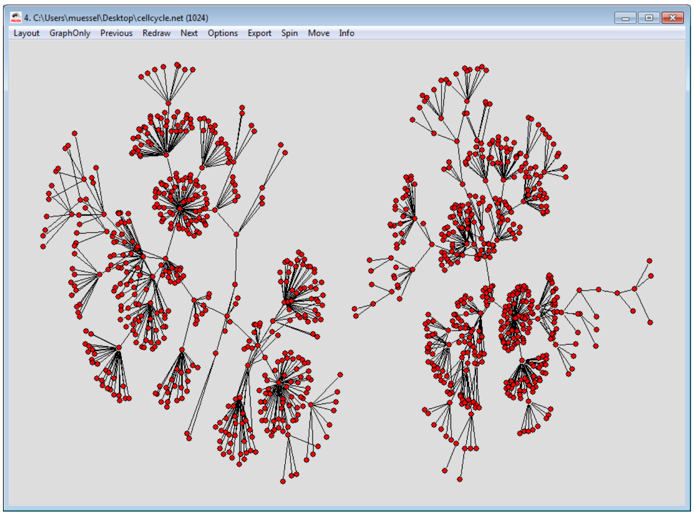

4. C:\Users\muessel\Desktop\cellcycle.net (1024)
Layout GraphOnly Previous Redraw Next Options Export Spin Move Info

Figure 11: A visualization of the mammalian cell cycle network in Pajek.

References

[1] M.~Aldana. Boolean dynamics of networks with scale-free topology. *Physica D*, 185(1):45–66, 2003.

[2] M.~Aldana, S.~Coppersmith, and L.~P. Kadanoff. Boolean dynamics with random coupling. In E.~Kaplan, J.~E. Marsden, and K.~R. Sreenivasan, editors, *Perspectives and Problems in Nonlinear Science*. Springer, 2003.

[3] V.~Batagelij and A.~Mrvar. Pajek – program for large network analysis. *Connections*, 21(2):47–57, 1998.

[4] T.~M. Cover and J.~A. Thomas. *Information Theory*. Wiley, New York, 1991.

[5] A.~Fauré, A.~Naldi, C.~Chaouiya, and D.~Thieffry. Dynamical analysis of a generic Boolean model for the control of the mammalian cell cycle. *Bioinformatics*, 22(14):e124–e131, 2006.

[6] S.~A. Kauffman. Metabolic Stability and Epigensis in Randomly Constructed Genetic Nets. *Journal of Theoretical Biology*, 22(3):437–467, 1969.

[7] S.~A. Kauffman. *The Origins of Order: Self-Organization and Selection in Evolution*. Oxford University Press, 1993.

[8] S.~Kim, J.~Kim, and K.-H. Cho. Inferring gene regulatory networks from temporal expression profiles under time-delay and noise. *Computational Biology and Chemistry*, 31:239–245, 2007.

[9] H.~Lähdesmäki, I.~Shmulevich, and O.~Yli-Harja. On Learning Gene Regulatory Networks Under the Boolean Network Model. *Machine Learning*, 52(1-2):147–167, 2003.

[10] F.~Li, T.~Long, Q.~Ouyang, and C.~Tang. The yeast cell-cycle network is robustly designed. *PNAS*, 101:4781–4786, 2004.

[11] S.~Liang, S.~Fuhrman, and R.~Somogyi. REVEAL, a general reverse engineering algorithm for inference of genetic network architectures. *Pacific Symposium on Biocomputing*, 3:18–29, 1998.

[12] W.~J.~R. Longabaugh, E.~H. Davidson, and H.~Bolouri. Computational representation of developmental genetic regulatory networks. *Developmental Biology*, 283(1):1–16, 2005.

[13] I.~Shmulevich, E.~R. Dougherty, S.~Kim, and W.~Zhang. Probabilistic Boolean networks: a rule-based uncertainty model for gene-regulatory networks. *Bioinformatics*, 18(2):261–274, 2002.

[14] P.~T. Spellman, G.~Sherlock, M.~Q. Zhang, V.~R. Iyer, K.~Anders, M.~B. Eisen, P.~O. Brown, D.~Botstein, and B.~Futcher. Comprehensive Identification of Cell Cycle-regulated Genes of the Yeast Saccharomyces cerevisiae by Microarray Hybridization. *Molecular Biology of the Cell*, 9(12):3273–3297, 1998.

5 Appendix

5.1 Network file format

This section provides a full language description for the network file format of BoolNet. The language is described in Extended Backus-Naur Form (EBNF).

```
Rule = GeneName Separator BooleanExpression [Separator Probability];
BooleanExpression = GeneName
                    | "!" BooleanExpression
                    | "(" BooleanExpression ")"
                    | BooleanExpression " & " BooleanExpression
                    | BooleanExpression " | " BooleanExpression;
GeneName = ? A gene name from the list of involved genes ?;
Separator = ",";
Probability = ? A floating-point number ?;
```