SPIDDOR package vignette

Itziar Irurzun-Arana, Iñaki Tróconiz and José David Gómez-Mantilla

September 9, 2018

Contents

1 Installation

Before starting, make sure you have installed an R version >3.1.0.
To install SPIDDOR:

```
> install.packages("devtools")
> library(devtools)
> install_github("SPIDDOR/SPIDDOR")
```

If you use a proxy server you need to install httr package and install SPIDDOR with the following code:

```
> library(httr)
> set_config(use_proxy("10.10.10",8080)) #Set your proxy here
> library(devtools)
> install_github("SPIDDOR/SPIDDOR")
```

Additionally, a C/C++ compiler is needed to perform faster simulations with SPIDDOR. Follow the instructions below depending on your operating system.

1.1 Windows

For Windows users, Rtools is needed to compile the simulation algorithm in C++.
It can be downloaded from: `https://cran.r-project.org/bin/windows/Rtools`

Be aware of the path where Rtools or RBuildTools is installed and then use the *Connect2Rtools* function each time you start a new R session in order to connect the package with Rtools gcc compiler. Example:

```
> library(SPIDDOR)
> Connect2Rtools(path="C:/Rtools")
```

Windows users that do not want to use *Connect2Rtools* each time a R session is opened, they can add the path to Rtools folder and the gcc compiler to their environment variables (in the Control Panel) writting them in the first place (e.g. `C:\\Rtools\\bin;C:\\Rtools\\gcc-4.6.3\\bin;` `rest-of-environment-variables`)

1.2 Mac OS X

If you are using a Mac, you will need to start by making sure you have Xcode + developer tools installed. You can do this in one of two ways. Either:

1- Download and install XCode from the Mac AppStore: `http://itunes.apple.com/us/app/xcode/id497799835?mt=12`
2- Within XCode go to Preferences : Downloads and install the Command Line Tools

However, XCode.app is a huge application that you do not need unless you are developing OS X or iOS applications. Alternatively (for a smaller download size) you can:

1- Register as an Apple Developer here (free): `https://developer.apple.com/programs/register/`
2- Download the Command Line Tools for XCode appropriate for the version of OS X you are running from here: `https://developer.apple.com/downloads/`

Second, you need to tell R which compilers to use. To do that, you need to create a file in the folder where R is installed in your computer named Makevars: `~/.R/Makevars`. Then write the following lines with your favourite text editor:

CC=clang
CXX=clang++

Otherwise, you can do this with the following commands in the Terminal (writing your true directory to the R folder):

```
cd ~/.R
nano Makevars
```

Add the text:

CC=clang
CXX=clang++

And follow the directions at the bottom of the screen to "write out" and close the file. (Control-O Enter and Control-X.)

Now you need to install Rcpp again from source so it is built with the same C++ compiler you are using. In R:

```
> install.packages("Rcpp", type = "source")
```

A good test to check that everything is set up correctly is to use the devtools package. In R:

```
> install.packages("devtools")
> devtools::has_devel()
```

If returns TRUE, you are good to go with SPIDDOR.

1.3 Linux

For Debian/Ubuntu, you can install the core software development utilities required for R package development by executing:

```
sudo apt-get install r-base-dev texlive-full
```

Some packages may require installation of additional R build dependencies. To provide all components needed to build R itself from source you can execute:

```
sudo apt-get build-dep r-base-core
```

2 Introduction

SPIDDOR is an R package which consists on a set of tools to perform Boolean modeling in the context of development therapies for complex diseases. SPIDDOR allows users to simulate synchronous and asynchronous Boolean networks and analyze the results in terms of the average dynamic evolution of the nodes or in terms of attractors.

From a methodological point of view the Boolean analysis performed by SPIDDOR involves certain novelties. Common Boolean modeling approaches only define direct activation-inhibition relationships between the components of the network. In our models, we incorporate the modulation interactions, which are used to modulate the intensity of the activations or inhibitions produced by the regulator nodes. The package also allows users to specify the activity level of their nodes as a percentage ON in order to perform mutational studies or evaluate the inputs of the network with background noises. Additionally, SPIDDOR incorporates new visualization techniques to evaluate the attractors of the system and the effects of perturbations.

3 Assembling networks

3.1 From text files

Networks can be loaded from text files in which the user describe the Boolean functions (*BFs*) of the network extracted from literature data. Nodes can be genes, proteins, metabolites, cellular states, stimulus, etc. *BFs* consist on a set of rules specifying how the nodes' states change as a function of the current or past values of its regulator nodes.

3.1.1 Boolean operators

The main operations of Boolean algebra are the conjunction AND (& in the txt file), the disjunction OR (| in the txt file), and the negation NOT (! in the txt file). Some examples:

```
GeneA = TF1 & TF2
GeneB = TF1 | TF3
GeneC = ! TF3
```

Apart from these basic operators, some convenience operations have been defined:

- THR: threshold operator used to check whether a regulator of a Boolean function is activated in the last n iterations that the user selects (n 3 by default). The THR operator requires a duration argument which indicates the number of previous iteration that must be evaluated for a regulator node. This argument can be numerical or a character in which case its name will be saved inside a list where the user can select the value afterwards. Example:

  ```
  CTLA4 = THR_TO_ACT[3] or CTLA4 = THR_TO_ACT[TO_ACTmax]
  ```

- MOD: modulator operator that makes the same threshold function as the THR operator but only affecting to the nodes that have a modulation interaction in the Boolean functions of the network. Example:

  ```
  IL12 =(CD40 & CD40L) | ((IL12 & ICOS) &! (MOD_IL12[4] & MOD_ICOS[4]))
  ```

- ANY: operator used to check whether a regulator of a Boolean function is activated in any of the last n iterations of the simulation that the user selects (n 3 by default). This operator is also used to modulate the Boolean functions of the network and has an argument to indicate the iterations that has to be evaluated (it can also be a number of a character). Example:

  ```
  NK = IL23 &! (NK & ANY_Treg[neg_modulator])
  ```

Here, the Boolean functions of a toy network are shown as they should appear in the text file:

```
APC-Ag = APC-Ag
B71 = APC-Ag
ICOS = APC-Ag
CD40 = APC-Ag
B7H2 = ICOS
CD28 = ! CTLA4
CTLA4= THR_T0_ACT[3]
CD40L = ICOS & B7H2 &! (CD40 & CD40L)
T0_ACT = ((CD28 & B71) | (T0_ACT & B7H2) &! (MOD_T0_ACT & MOD_B7H2))
          &! (CTLA4 & B71)
IL2 = T0_ACT
IL6 = CD28
IL12 = (CD40 & CD40L) | (IL12 & ICOS) &! (MOD_IL12 & MOD_ICOS)
```

If this equations are saved to a "Example_network.txt" file in the working directory, the network can be loaded to R via

```
> BN<-read.Boolean.functions(file="Example_network.txt")
```

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

```
> data(Example_network)
```

3.1.2 Boolean inputs

Input nodes are defined as the nodes that have no incoming relationship with other nodes of the network. Inputs in SPIDDOR are as initial network states. Different combination of inputs, generally, lead to different ouputs of the system.
There are 3 ways of defining an input node in the text files. To illustrate the examples, let's take the input node of the example network previously shown, `APC-Ag`.

1. input node = input node: Recommended way of declaring an input. The user can select in every simulation whether the input node will start as 0 or 1. Example: `APC-Ag = APC-Ag`
2. input node = value: Is the way of declaring a fixed node, which value can only be 0 or 1. The input node will take the same value in every simulation. Example: `APC-Ag = 1`
3. No definition of the input node: If a node name is used in the Boolean equations of other nodes in the network but no declaration exist for this node, SPIDDOR saves this node name as an input node and defines its Boolean function using the input node = input node nomenclature.

3.1.3 Output of the assembled network

Once all the *BFs* are specified in the text file, they can be loaded in R via `read.Boolean.functions`. This function returns a list structure of class `BN` (from Boolean Network) representing the most relevant information of the network. It has the following components:

- **node.names**: A vector of the node names of the network

- **Initial_conditions**: A vector with the name of the nodes that will start in ON state in the first iteration of the simulation algorithm.
- **Modulator**: The duration of the modulation interactions that take place in the network dynamics. It could have specific arguments for each Boolean expression where a modulation occurs, or it can have a general argument `modulation_dur` where the user can specify a general time duration for all the modulation of the network.
- **Arguments**: A list with other arguments needed for a correct simulation of the network. Here, we included the duration of the threshold operators (THR).
- **Polymorphism**: A vector specifying the activity level of each node in the network. Default values are 1 for each node, meaning a 100% activity for all the components of the network. To perform a mutational study and change the activity of a node to 50% the next command can be used:

  ```
  > BN$Polymorphism["node name"]=0.5
  ```

For the example network of Boolean operators section, the BN list has the following structure.

```
> BN <- read.Boolean.functions("Example_network.txt")

>print(BN)
$nodes.names
 [1] "APC_Ag" "B71"    "CD40"   "B7H2"   "ICOS"   "CD28"   "CTLA4"  "CD40L"  "T0_ACT"
    "IL6" "IL12"   "IL2"

$Initial_conditions
[1] "APC_Ag"

$Modulator
modulation_dur
             3
$Arguments
T0_ACTmax_CTLA4
              3
$Polymorphism
APC_Ag    B71   CD40   B7H2   ICOS   CD28  CTLA4  CD40L T0_ACT    IL6   IL12    IL2
     1      1      1      1      1      1      1      1      1      1      1      1
```

As a specific duration argument has not been defined in the modulation interactions of the text file, a general argument `modulation_dur` appears in the `$Modulation` section. We now change the text file to see the difference in the `BN` class when the duration of the modulations are defined numerically in the *BFs* with [] symbols:

Example_network.txt file:

```
APC-Ag = APC-Ag
B71 = APC-Ag
ICOS = APC-Ag
CD40 = APC-Ag
B7H2 = ICOS
CD28 = ! CTLA4
CTLA4= THR_TO_ACT[3]
CD40L = ICOS & B7H2 &! (CD40 & CD40L)
TO_ACT = ((CD28 & B71) | (TO_ACT & B7H2) &!  (MOD_TO_ACT[3] & MOD_B7H2[3]))
&!  (CTLA4 & B71)
IL2 = TO_ACT
IL6 = CD28
IL12 = (CD40 & CD40L) | (IL12 & ICOS) &!  (MOD_IL12[4] & MOD_ICOS[4])
```

```
> BN <- read.Boolean.functions("Example_network.txt")

>print(BN)
$nodes.names
 [1] "APC_Ag" "B71"    "CD40"   "B7H2"   "ICOS"   "CD28"   "CTLA4"  "CD40L"  "TO_ACT"
    "IL6" "IL12"   "IL2"

$Initial_conditions
[1] "APC_Ag"

$Modulator
MOD_TO_ACT   MOD_IL12
         3          4
$Arguments
TO_ACTmax_CTLA4
              3
$Polymorphism
APC_Ag    B71   CD40   B7H2   ICOS   CD28  CTLA4  CD40L TO_ACT    IL6   IL12    IL2
     1      1      1      1      1      1      1      1      1      1      1      1
```

Finally, a character instead of a number can be used to defined the duration argument of the modulations.

Example_network.txt file:

```
APC-Ag = APC-Ag
B71 = APC-Ag
ICOS = APC-Ag
CD40 = APC-Ag
B7H2 = ICOS
CD28 = ! CTLA4
CTLA4= THR_TO_ACT[3]
CD40L = ICOS & B7H2 &! (CD40 & CD40L)
TO_ACT = ((CD28 & B71) | (TO_ACT & B7H2) &!  (MOD_TO_ACT[mod1] &
MOD_B7H2[mod1])) &!  (CTLA4 & B71)
```

```
IL2 = T0_ACT
IL6 = CD28
IL12 = (CD40 & CD40L) | (IL12 & ICOS) &!  (MOD_IL12[mod2] & MOD_ICOS[mod2])
```

```
> BN <- read.Boolean.functions(file="Example_network.txt")
```

```
>print(BN)
$nodes.names
 [1] "APC_Ag" "B71"    "CD40"   "B7H2"   "ICOS"   "CD28"   "CTLA4"  "CD40L"  "T0_ACT"
    "IL6" "IL12"   "IL2"

$Initial_conditions
[1] "APC_Ag"

$Modulator
mod1   mod2
   3      3
$Arguments
T0_ACTmax_CTLA4
              3
$Polymorphism
APC_Ag    B71   CD40   B7H2   ICOS   CD28  CTLA4  CD40L T0_ACT    IL6   IL12    IL2
     1      1      1      1      1      1      1      1      1      1      1      1
```

`mod1` and `mod2` have a default value of 3 when loading the network for the first time. However, these arguments can be modified in R by simply typing:

```
> BN$Modulator["mod2"]<-4
```

Then, all the modulation interactions that have the `mod2` argument will use the value of 4.

3.2 From Cell Collective repository

Networks can be loaded from a previously downloaded file from The Cell Collective repository (`www.thecellcollective.org/`) using `read.cellcollective`.
Here is the example of the Lac operon network written with The Cell Collective semantics [4].

CellCollective_file.txt file:

```
allolactose = ( enviro_lactose )
lac_enzymes = ( lac_mRNA )
lac_operon = ( ( (CAP) AND NOT (CAP_mutation) ) AND NOT (lac_repressor) )
lac_repressor =  NOT ( ( allolactose ) )
CAP = ( cAMP )
lactose_breakdown = ( ( lac_enzymes  ) AND NOT ( lacZ_mutation  ) )
cAMP =  NOT ( ( enviro_glucose ) )
lac_mRNA = ( lac_operon )
```

```
> BN<-read.cellcollective(file="CellCollective_file.txt")
```

```
>print(BN)
$nodes.names
 [1] "allolactose"       "lac_enzymes"       "lac_operon"        "lac_repressor"
 [5] "CAP"               "lactose_breakdown" "cAMP"              "lac_mRNA"
 [9] "enviro_lactose"    "CAP_mutation"      "lacZ_mutation"     "enviro_glucose"

$Initial_conditions
[1] "enviro_lactose" "CAP_mutation"   "lacZ_mutation"  "enviro_glucose"

$Modulator
named numeric(0)

$Arguments
numeric(0)

$Polymorphism
      allolactose       lac_enzymes        lac_operon     lac_repressor               CAP
                1                 1                 1                 1                 1
lactose_breakdown              cAMP          lac_mRNA    enviro_lactose      CAP_mutation
                1                 1                 1                 1                 1
    lacZ_mutation    enviro_glucose
                1                 1
```

4 Simulation algorithm

The `read.Boolean.functions` and `read_cellcollective` functions create a simulation algorithm to compute the dynamic trayectory of the network using the synchronous or asynchronous updating methods. The user can select the programming language (R or C++) where the simulation algorithm will be coded (Defaults is C++). We recommend to choose C++ as it is considerably faster than the code computed in R alone.

The following code performs a dynamic evolution of the network for 20 time steps (19+Initial condition) using the synchronous mode:

```
> BN<-read.Boolean.functions(file="Example_network.txt", language="C")
> Pattern<-dynamic_evolution.f(BN,time.steps=19,asynchronous=FALSE)
> head(Pattern)
```

```
       1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
APC_Ag 1 1 1 1 1 1 1 1 1  1  1  1  1  1  1  1  1  1  1  1
B71    0 1 1 1 1 1 1 1 1  1  1  1  1  1  1  1  1  1  1  1
CD40   0 1 1 1 1 1 1 1 1  1  1  1  1  1  1  1  1  1  1  1
B7H2   0 0 1 1 1 1 1 1 1  1  1  1  1  1  1  1  1  1  1  1
ICOS   0 1 1 1 1 1 1 1 1  1  1  1  1  1  1  1  1  1  1  1
CD28   0 1 1 1 1 0 1 1 1  1  0  1  1  1  1  0  1  1  0  1
```

To perform a dynamic evolution using the asynchronous updating method:

```
> Pattern2<-dynamic_evolution.f(BN,time.steps=19,asynchronous=TRUE)
```

The dynamics of the network might change each time `dynamic_evolution.f` is run in the asynchronous mode due to the randomness involved in this updating scheme. Therefore, a large number of repetitions are needed to calculate the fraction of simulations in which the nodes are ON for each time step. To perform this average function the user can call to:

```
> AVG<-Average_simulations.f(BN,time.steps=49,repetitions=2500)
> head(AVG)

       1      2      3      4      5      6      7      8 ...
APC_Ag 1 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
B71    0 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
CD40   0 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
B7H2   0 0.4792 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
ICOS   0 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
CD28   0 1.0000 1.0000 0.9420 0.6796 0.3904 0.6664 0.9476
CTLA4  0 0.0000 0.0000 0.1660 0.5720 0.5396 0.0800 0.0500
CD40L  0 0.1616 0.5944 0.4056 0.5944 0.4056 0.5944 0.4056
T0_ACT 0 0.3356 1.0000 1.0000 0.7436 0.2528 0.4168 0.8060
IL6    0 0.5052 1.0000 0.9848 0.8540 0.5484 0.4900 0.8052
IL12   0 0.0344 0.3624 0.6836 0.8420 0.9056 0.7868 0.6368
IL2    0 0.0868 0.6620 1.0000 0.8872 0.5172 0.2980 0.5856
```

To plot these results using ggplot2 package we need to transform the `AVG` matrix:

```
> AVG2<-toggplot(AVG)
>  ggplot(data=AVG2,aes(x=time,y=value)) +
+         geom_line( colour="#336600",size = 1.5) + ylab(" % of activation") +
+         xlab("Time steps") + facet_wrap(~variable)
```

4.1 Perturbation of the system

In order to know which node perturbations can result in a significant change of the network dynamics, we can manipulate the system by knocking out or overexpressing the nodes. A knockout implies the deactivation of a component during all the simulation, whereas an overexpression generates a persistent activation of a node. Another possibility is to overexpress a node but only after its first activation or to activate/deactivate a node for some iterations. The majority of the function of SPIDDOR have arguments to introduce these perturbations:

- Knockouts: A character vector with the name of the nodes to knockout (fixed to 0) over all the simulations(empty by default).
- Over_expr: A character vector with the name of the nodes to overexpress (fixed to 1) over all the simulations (empty by default).
- Over_expr_AA: A character vector with the name of the nodes to overexpress (fixed to 1) after their first activation (empty by default).
- KO_times: A numeric vector specifying the iterations where the nodes in Knockouts argument will be fixed to 0. If empty the knockout is applied to the nodes for the entire simulation (empty by default).
- OE_times: A numeric vector specifying the iterations where the nodes in Over_expr argument will be fixed to 1. If empty the overexpression is applied to the nodes for the entire simulation (empty by default).

For example, to knock out ICOS when computing the dynamic evolution of the example network, we call:

```
> data(Example_network)
> BN <- read.Boolean.functions(Lines=BN$BooleanFunctions)
```

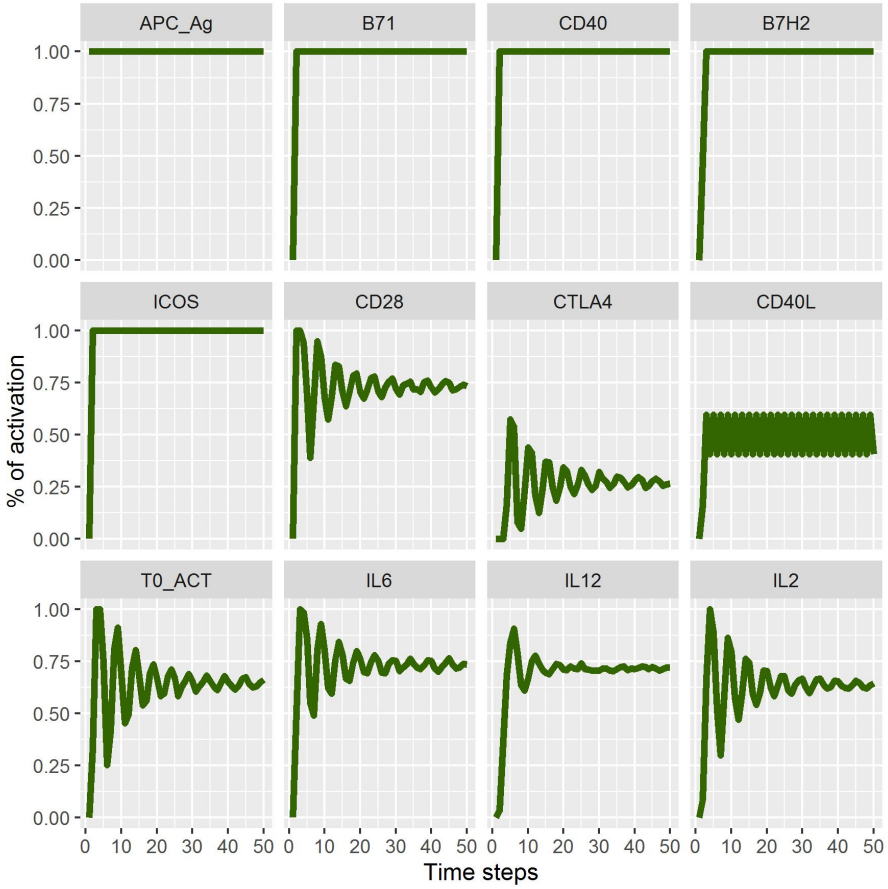


Figure 1: Average evolution of the network after 50 time steps in asynchronous mode.

```
#Dynamic evolution of the network with a knockout in ICOS:

> P_KOICOS<-dynamic_evolution.f(BN,time.steps=19,Knockouts="ICOS",asynchronous = T)
> P_KOICOS["ICOS",]

 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

But if we only want to knock out ICOS from time step 5 to 10, we call:

```
> P_KOICOS2<-dynamic_evolution.f(BN,time.steps=19,Knockouts="ICOS",
+                                KO_times=seq(5,10),asynchronous = T)
> P_KOICOS2["ICOS",]

 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
 0  1  1  1  0  0  0  0  0  0  1  1  1  1  1  1  1  1  1  1
```

To knockout ICOS and B71, one from time step 5 to 10 and the other from time step 10 to 15:

```
> P_KOICOS_B71<-dynamic_evolution.f(BN,time.steps=19,Knockouts=c("ICOS","B71"),
+                                   KO_times=list(seq(5,10),seq(10,15)),
+                                   asynchronous = T)
> P_KOICOS_B71[c("ICOS","B71"),]

     1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
ICOS 0 1 1 1 0 0 0 0 0  0  1  1  1  1  1  1  1  1  1  1
B71  0 1 1 1 1 1 1 1 1  0  0  0  0  0  0  1  1  1  1  1
```

To overexpress node CD40L after its first activation when compunting the average trajectory of the network, we use:

```
> AVG_OE_CD40L<-Average_simulations.f(BN,time.steps=19,Over_expr_AA = "CD40L",
+                                     asynchronous = TRUE,repetitions=2500)
> AVG_OE_CD40L["CD40L",]

    1     2     3     4     5     6     7     8     9    10    11    12 ...
0.000 0.159 0.755 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000
```

Additionally, users can change the activity level of the nodes in order to introduce "polymorphisms-like" mutations that, instead of completely deactivating a component of the network, they decrease the activity of a node to a lower extent (75%, 50%, 25%...). This can be done easily by setting the ranges of the activity of the nodes in the `Polymorphism` argument from the `BN` class from 0 to 1 (0% activity, 100%activity). In the next example, we decrease the activity level of B71 node to 50%.

```
> BN$Polymorphism["B71"]=0.5
> print(BN$Polymorphism)

APC_Ag    B71   CD40   B7H2   ICOS   CD28  CTLA4  CD40L TO_ACT    IL6   IL12    IL2
   1.0    0.5    1.0    1.0    1.0    1.0    1.0    1.0    1.0    1.0    1.0    1.0

> AVG_B7150<-Average_simulations.f(BN,time.steps=19,Over_expr_AA = "CD40L",
+                                  asynchronous = TRUE,repetitions=5000)
> print(AVG_B7150["B71",])

    1     2     3     4     5     6     7     8     9    10    11    12 ...
0.000 0.501 0.493 0.502 0.509 0.494 0.489 0.495 0.499 0.505 0.506 0.501
```

5 Attractors

Boolean models eventually evolve into a limited set of stable states known as attractors [6]. SPIDDOR is able to identify attractors using synchronous and asynchronous updating methods. An attractor can be a fixed-point if it consists on only one state, a simple cycle if it is composed by more than one state that oscillate in a cycle or a complex attractor if the set of states oscillate irregularly (due to the randomness involved in asynchronous networks). In our models the states of the nodes in the current iteration not only depend on the states of the nodes in the previous step, but also on prior steps due to the temporal predicates implemented with SPIDDOR. This produces an additional attractor type using the synchronous attractor search that we called "complex cycle", which consists on regular cycles with duplicated states.
The `Get_Attractor.f` function gets the attractor using the synchronous or asynchronous updating methods starting from an initial condition. To demonstrate this we will use the cell cycle Boolean network from [2].

```
> data(cellcycle)
> print(BN_cellcycle)

$nodes.names
 [1] "CycD"   "Rb"     "E2F"    "CycE"   "CycA"   "p27"    "Cdc20"  "Cdh1"   "UbcH10"
[10] "CycB"

$Initial_conditions
[1] "CycD"

$Modulator
named numeric(0)

$Arguments
numeric(0)

$Polymorphism
  CycD     Rb    E2F   CycE   CycA    p27  Cdc20   Cdh1 UbcH10   CycB
     1      1      1      1      1      1      1      1      1      1

$BooleanFunctions
 [1] "CycD= CycD"
 [2] "Rb= (! CycA & ! CycB & ! CycD & ! CycE) | (p27 & ! CycB & ! CycD)"
 [3] "E2F= (! Rb & ! CycA & ! CycB) | (p27 & ! Rb & ! CycB)"
 [4] "CycE= (E2F & ! Rb)"
 [5] "CycA= (E2F & ! Rb & ! Cdc20 & ! (Cdh1 & UbcH10)) | (CycA & ! Rb & ! Cdc20
      & ! (Cdh1 & UbcH10))"
 [6] "p27= (! CycD & ! CycE & ! CycA & ! CycB) | (p27 & ! (CycE & CycA)
      & ! CycB &! CycD)"
 [7] "Cdc20= CycB"
 [8] "Cdh1=(! CycA & ! CycB) | (Cdc20) | (p27 & ! CycB)"
 [9] "UbcH10= ! Cdh1 | (Cdh1 & UbcH10 & (Cdc20 | CycA | CycB))"
[10] "CycB= ! Cdc20 & ! Cdh1"

#Create the simulation algorithm for the cell cycle network

> BN_cellcycle <- read.Boolean.functions(Lines=BN_cellcycle$BooleanFunctions)

#Get the attractor using the synchronous search (asynchronous=FALSE):
```

```
> Attractor_syn<-Get_Attractor.f(BN_cellcycle,asynchronous=FALSE,Percent.ON=FALSE)
> print(Attractor_syn)

  CycD Rb E2F CycE CycA p27 Cdc20 Cdh1 UbcH10 CycB
1    1  0   1    0    0   0     0    1      1    0
2    1  0   1    1    0   0     0    1      0    0
3    1  0   1    1    1   0     0    1      0    0
4    1  0   0    1    1   0     0    0      0    0
5    1  0   0    0    1   0     0    0      1    1
6    1  0   0    0    1   0     1    0      1    1
7    1  0   0    0    0   0     1    1      1    0
```

If Percent.ON=TRUE is selected, the function returns the attractor represented with the probability of activation of the nodes. If Percent.ON=FALSE is chosen, it returns all the states that form the attractor in a data.frame. For this example, we get a cycle composed of 7 states. If we want to see this in %ON without recalling to the function with Percent.ON=TRUE, we can type:

```
> print(apply(Attractor_syn,2,sum)/dim(Attractor_syn)[1])

  CycD     Rb    E2F   CycE   CycA    p27  Cdc20   Cdh1 UbcH10   CycB
 1.000  0.000  0.429  0.429  0.571  0.000  0.286  0.571  0.571  0.286
```

If the asynchronous attractor search is used, the function looks for the attractor by re-computing the simulation algorithm several times with a large number of iterations (more than 1000). Therefore, an additional argument `repetitions` is needed. Even when few repetitions are selected, the algorithm estimates a good approximation of the final attractor, that is why, we recommend first to try with a number smaller that 20 for this argument.

```
> Attractor_asyn<-Get_Attractor.f(BN_cellcycle,repetitions=12,asynchronous=TRUE)
> print(Attractor_asyn)

  CycD     Rb    E2F   CycE   CycA    p27  Cdc20   Cdh1 UbcH10   CycB
 1.000  0.000  0.404  0.404  0.542  0.000  0.322  0.576  0.580  0.307
```

If you compare the synchronous and asynchronous attractors, you can appreciate that they are not completely the same. This is because, with the synchronous attractor search we get a simple cycle, whereas with the asynchrnous search the attractor is a complex cycle.

In some cases there is not enough information to specify the initial condition of a system and sampling of a multitude of initial conditions is necessary. For those interested in this feature, SPIDDOR includes the `Get_all_attractors.f` function. Now, we will select the cardiac gene regulatory network from [5].

```
> data(Cardiac_network)
> print(BN_Cardiacnetwork$nodes.names)

 [1] "Isl1"             "canWnt"           "exogen_BMP2_II"   "Bmp2"
 [5] "exogen_canWnt_II" "Tbx1"             "Nkx2_5"           "Dkk1"
 [9] "Mesp1"            "Fgf8"             "Tbx5"             "Foxc1_2"
[13] "GATAs"            "exogen_CanWnt_I"  "exogen_BMP2_I"

> BN_Cardiacnetwork<-read.Boolean.functions(Lines=BN_Cardiacnetwork$BooleanFunctions)
```

This network has 15 nodes, so the possible initial conditions to test are 32768 (2^{15}). As the number of initial conditions to test grow exponentially with the number of nodes, `Get_all_attractors.f`

function searches all the attractors for networks with less than 20 nodes. For larger networks, we allow the specification of a subset of nodes (<20) in `BN$Initial_conditions` in which all the combination of nodes will be tested, or the specification of a number of starting states to test (<100,000). Furthermore, this algorithm is parallelized using the snowfall library [7], so the user must specified the number of cpus to use. In the next example, we use 4 cpus and the synchronous search to get all the attractors of the cardiac network:

```
> All_attr_syn<-Get_all_attractors.f(cpus=4,BN_Cardiacnetwork,asynchronous=FALSE)
> print(All_attr_syn)

  Isl1 canWnt exogen_BMP2_II Bmp2 exogen_canWnt_II Tbx1 Nkx2_5 Dkk1 Mesp1 Fgf8 Tbx5
1    0      0              1    1                0    0      0    0     0    0    0
2    0      0              1    1                0    0      1    0     0    0    1
3    1      1              1    0                1    1      1    0     0    1    0
  Foxc1_2 GATAs exogen_CanWnt_I exogen_BMP2_I  Recurrence
1       0     0               0             1 0.007263184
2       0     1               0             1 0.492736816
3       1     1               1             1 0.500000000
```

If we want to get the attractors by only checking 1000 random initial states, we call:

```
> All_attr_syn<-Get_all_attractors.f(cpus=4,BN_Cardiacnetwork,asynchronous=FALSE,
+                                    startStates=1000)
> print(All_attr_syn)

  Isl1 canWnt exogen_BMP2_II Bmp2 exogen_canWnt_II Tbx1 Nkx2_5 Dkk1 Mesp1 Fgf8 Tbx5
1    0      0              1    1                0    0      0    0     0    0    0
2    0      0              1    1                0    0      1    0     0    0    1
  Foxc1_2 GATAs exogen_CanWnt_I exogen_BMP2_I Recurrence
1       0     0               0             1       0.06
2       0     1               0             1       0.94
```

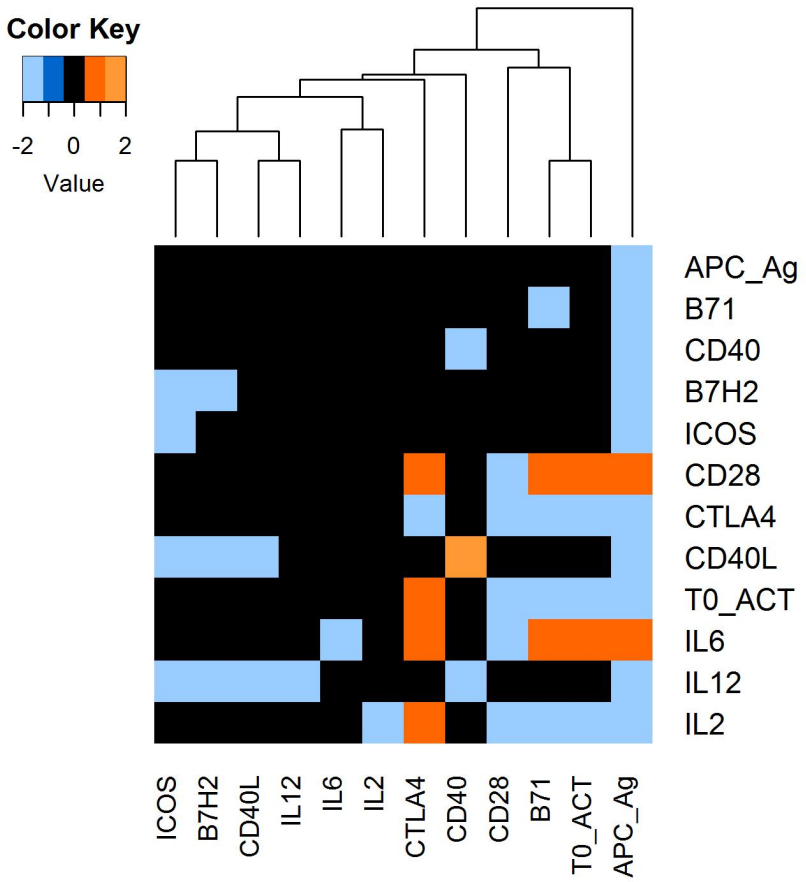
By only checking 1000 initial states we do not get the third fixed-point of the network.

6 Perturbation analysis

`KO_matrix.f` function evaluates the effect of single node knockouts on the network by computing the Perturbation index ($Prob_{Perturbed}/Prob_{Normal}$) of the nodes. The result of this function is a square matrix indicating how the knockout of the "column node" affects each "row node".
`OE_matrix.f` function is equal to the previous function but it evaluates the effects of overexpressions on the nodes of the network.

`Create_heatmap` function is used to visualize the individual values contained in the matrix returned by these two functions as colors with a heatmap of the column nodes. The perturbations that lead to a higher activation of the nodes compared to an unperturbed situation are represented in orange while a lower activation of the nodes is indicated in blue. This color palette can be changed by the user.

```
> data(Example_network)
> BN <- read.Boolean.functions(Lines=BN$BooleanFunctions)
> KO.m<-KO_matrix.f(BN,time_steps=999,repetitions=24,asynchronous=TRUE)
> Create_heatmap(KO.m)
```


Color Key
-2
0
2
Value
APC_Ag
B71
CD40
B7H2
ICOS
CD28
CTLA4
CD40L
T0_ACT
IL6
IL12
IL2
ICOS
B7H2
CD40L
IL12
IL6
IL2
CTLA4
CD40
CD28
B71
T0_ACT
APC_Ag

7 Model interoperability

The `export2SBMLqual` function is used to convert the BFs of the user's text file to the SBML qual format[1]. The output file is saved in .sbml extension, and can be imported to other Boolean analysis tools like GINsim[3] or BoolNet[8] (both platforms are part of the CoLoMoTo Consortium [9]).

```
> data(Cardiac_network)
> export2SBMLqual(Lines=BN_Cardiacnetwork$BooleanFunctions,file="cardiac_network.sbml")
```

References

[1] Claudine Chaouiya, Duncan Bérenguier, Sarah M Keating, Aurélien Naldi, Martijn P van Iersel, Nicolas Rodriguez, Andreas Dräger, Finja Büchel, Thomas Cokelaer, Bryan Kowal, Benjamin Wicks, Emanuel Gonçalves, Julien Dorier, Michel Page, Pedro T Monteiro, Axel von Kamp, Ioannis Xenarios, Hidde de Jong, Michael Hucka, Steffen Klamt, Denis Thieffry, Nicolas Le Novère, Julio Saez-Rodriguez, and Tomáš Helikar. SBML qualitative models: a model representation format and infrastructure to foster interactions between qualitative modelling formalisms and tools. *BMC Syst. Biol.*, 7:135, 10 December 2013.

[2] Adrien Fauré, Aurélien Naldi, Claudine Chaouiya, and Denis Thieffry. Dynamical analysis of a generic boolean model for the control of the mammalian cell cycle. *Bioinformatics*, 22(14):e124–31, 15 July 2006.

[3] A Gonzalez Gonzalez, A Naldi, L Sánchez, D Thieffry, and C Chaouiya. GINsim: a software suite for the qualitative modelling, simulation and analysis of regulatory networks. *Biosystems.*, 84(2):91–100, May 2006.

[4] Tomáš Helikar, Christine E Cutucache, Lauren M Dahlquist, Tyler A Herek, Joshua J Larson, and Jim A Rogers. Integrating interactive computational modeling in biology curricula. *PLoS Comput. Biol.*, 11(3):e1004131, March 2015.

[5] Franziska Herrmann, Alexander Groß, Dao Zhou, Hans A Kestler, and Michael Kühl. A boolean model of the cardiac gene regulatory network determining first and second heart field identity. *PLoS One*, 7(10):e46798, 2 October 2012.

[6] Martin Hopfensitz, Christoph Müssel, Markus Maucher, and Hans A Kestler. Attractors in boolean networks: a tutorial. *Comput. Stat.*, 28(1):19–36, 8 April 2012.

[7] Jochen Knaus. snowfall: Easier cluster computing (based on snow), 2013.

[8] Christoph Müssel, Martin Hopfensitz, and Hans A Kestler. BoolNet–an R package for generation, reconstruction and analysis of boolean networks. *Bioinformatics*, 26(10):1378–1380, 15 May 2010.

[9] Aurélien Naldi, Pedro T Monteiro, Christoph Müssel, Consortium for Logical Models and Tools, Hans A Kestler, Denis Thieffry, Ioannis Xenarios, Julio Saez-Rodriguez, Tomas Helikar, and Claudine Chaouiya. Cooperative development of logical modelling standards and tools with CoLoMoTo. *Bioinformatics*, 31(7):1154–1159, 1 April 2015.