rGriffin

Mariana E Martinez-Sanchez

Stalin Muñoz

Miguel Carrillo

Eugenio Azpeitia

David Rosenblueth

2018-08-04

Introduction

Boolean networks allow us to give a mechanistic explanation to how cell types emerge from regulatory networks. However, inferring the regulatory network and its functions is complex problem, as the available information is often incomplete. `rGriffin` uses available biological information (regulatory interactions, cell types, mutants) codified as a set of restrictions and returns the Boolean Networks that satisfy that restrictions. This Boolean networks can then be used to study the biological system.

The `rGriffin` package is an R connector to `Griffin` (Gene Regulatory Interaction Formulator For Inquiring Networks), a java library for inference and analysis of Boolean Network models. `Griffin` takes as inputs biologically meaningful constraints and turns them into a symbolic representation. Using a SAT engine, `Griffin` explores the Boolean Network search space, finding all satisfying assignments that are compatible with the specified constraints. The`rGriffin` package includes a number of functions to interact with the BoolNet package.

Queries

The first step is to attach `rGriffin`. This will initialize the Java Virtual Machine and start `Griffin` with the default JVM options.

If you want to initialize the JVM with different options like more memory see the functions `init.griffin()`. It is also posible to modify the default parameters changing the file “rGriffin/java/jvm-param.R”.

Create a query

All queries start with a topology that describes the nodes and its interactions. The function `create.gquery.graph()` takes a dataframe with columns for: source node, target node, and type of interaction. It also takes a vector with the node names.

Depending on the sign the interactions can **positive** or **negative**. If in every condition the regulation will have the same sign the interaction is **ambiguous**. However, if you are not sure if the regulation is positive or negative in all contexts you can say that the interaction is **ambiguous**. Depending on the degree of confidence in the existence of the interaction, the interactions can be: **mandatory** if you are sure the interaction will happen or **optional** if you suspect the interaction exists but you are not sure.

The valid types of interctions are:

- false: Contradiction
- MA: Mandatory, ambiguous
- MPU (or +): Mandatory, positive, unambiguous
- MPPA: Mandatory, positive, possibly ambiguous
- MNU (or -): Mandatory, negative, unambiguous
- MNPA: Mandatory, negative, possibly ambiguous
- MUSU: Mandatory, unknown sign, unambiguous
- MUSPA: Mandatory, unknown sign, possibly ambiguous
- NR: No regulation
- OA: Optional, ambiguous
- OPU: Optional, positive, unambiguous
- OPPA: Optional, positive, possibly ambiguous
- ONU: Optional, negative, unambiguous
- ONPA: Optional, negative, possibly ambiguous
- OUSU: Optional, unknown sign, unambiguous
- true: Tautology

For example, suppose a network where:

- **a** activates **b**
- **b** and **c** inhibit each other
- **b** and **c may** have a positive self-regulatory loops.

We can codify this information as:

```
genes = c('a','b','c')
inter = data.frame(source=c('a','b','b','c','c'),
                   target=c('b','b','c','b','c'),
                   type=c('+','OPU','-','-','OPU'),
                     stringsAsFactors = F )
inter
#>   source target type
#> 1      a      b    +
#> 2      b      b  OPU
#> 3      b      c    -
#> 4      c      b    -
#> 5      c      c  OPU
```

We then create the query `q`. This creates an instance of the class query in the JVM.

```
q = create.gquery.graph(inter, genes)
q
#> [1] "Java-Object{mx.unam.iimas.griffin.r.RGriffinQuery@1bba400}"
```

If you want to see the query use the `print` method.

```
print(q)
#> Gene Regulation Specification
#>  Set of genes: [a, b, c]
#>  geneOrder: null
#>  Known regulations: None
```

```
#>  Hypothetical regulations: None
#>  Gene Interactions: [GeneInteraction [source=a, target=b, type=MPU], GeneInteraction
[source=b, target=b, type=OPU], GeneInteraction [source=b, target=c, type=MNU],
GeneInteraction [source=c, target=b, type=MNU], GeneInteraction [source=c, target=c,
type=OPU]]
#>  Stable states: None
#>  Prohibited states: null
#>  Known Regulators: None
#>  Hypothetical Regulators: None
#>  Extended regulators: None
#>  Known Regulations: None
#>  Hypothetical Regulations: None
#>  Cyclic attractors: None
#>  Input-output pairs: []
#>  Subspaces: None
#>  Mutant experiments: None
#>  State transition rules: None
#>  exactRegulations: false
#>  additionalSteadyStates: false
#>  divideQueryByTopology: false
#>  topologyIteratorType: SEQUENTIAL
#>  limitTopologyRange: false
#>  limitToNumberOfNetworks: -1
#>  topologyRange: None
#>  topologicalDistanceRadius: 1
#>  blockSteadyAPosteriory: false
#>  additonalCycles: false
#>  ambiguityNotAllowed: true
#>  numberOfCycles:0
```

Add restrictions to the query

It is possible to add more restrictions to the query. It is important to remember that if there are more restrictions we expect less networks.

For example, suppose that we have some information about the expected cell types. We can add this restrictions as attractors.

For example, suppose that we know that the attractors are:

- Only express **b** => c(0,1,0)
- Only express **c** => c(0,0,1)

We can also add partial attractors where we lack information

- Do not express **a** or **c** but we have NO information on **b** => c(0,'*',0)

We codify this information as:

```
attr = data.frame(a=c(0,'*',0),
                  b=c(0,1,0),
                  c=c(0,0,1),
                  stringsAsFactors = F )
```

We can add this information to the query with the `add.gquery.attractors()` function:

```
q = add.gquery.attractors(q, attr)
```

You can add aditional restrictions like:

- `add.gquery.cycle()` Add target cycle
- `add.gquery.mutant()` Add mutant with attractors
- `add.gquery.prohibited.attractors()` Add prohibited attractors
- `add.gquery.transition()` Add a transition between two succesive states
- `add.gquery.trapspace()` Add a trapspace

Run the query

Once you have created the query with `run.gquery()`. This function will return all the networks that satisfy the restrictions

```
nets = run.gquery(q)
print(nets)
#> [1] "targets,factors\na,false\nb,((((!a&b)&!c)|((a&!b)&!c))|((a&b)&!c))\nc,(!b&c)\n"
#> [2] "targets,factors\na,false\nb,(((((!a&b)&!c)|((a&!b)&!c))|((a&b)&!c))|((a&b)&c))\nc,
(!b&c)\n"
#> [3] "targets,factors\na,false\nb,((((((!a&b)&!c)|((!a&b)&c))|((a&!b)&!c))|((a&b)&!c))|
((a&b)&c))\nc,(!b&c)\n"
#> [4] "targets,factors\na,false\nb,((((!a&b)&!c)|((a&b)&!c))|((a&b)&c))\nc,(!b&c)\n"
#> [5] "targets,factors\na,false\nb,((((((!a&b)&!c)|((a&!b)&!c))|((a&!b)&c))|((a&b)&!c))|
((a&b)&c))\nc,(!b&c)\n"
```

The function `run.gquery()` includes multiple options that can be seen in the documentation. Some of the most important are:

- `allow.hypothesis` activate or deactivate hypothetical regulations
- `allow.additional.states` allows networks with additional fixed-point attractors to those specified in the query
- `allow.additional.cycles` allows networks with additional cyclic attractors to those specified in the query
- `return.network.limit` limit the maximum number of networks that the query will return

Connect to other packages

BoolNet

The R package includes various functions to import and export data to `BoolNet`.

It is possible to obtain the topology of a `BooleanNetwork` object using `get.net.topology()`. This function determines the sign of each regulation as positive '+', negative '-' or ambiguous 'MA'. All regulations between nodes are considered mandatory. The function can also detect Non-functional regulations 'NR'.

```
library(BoolNet)
data("cellcycle")
topology <- get.net.topology(cellcycle)
topology
#>    Source Target Interaction
#> 1    CycD   CycD           +
#> 2    CycD     Rb           -
#> 3    CycE     Rb           -
#> 4    CycA     Rb           -
#> 5     p27     Rb           +
#> 6    CycB     Rb           -
#> 7      Rb    E2F           -
#> 8    CycA    E2F           -
#> 9     p27    E2F           +
#> 10   CycB    E2F           -
#> 11     Rb   CycE           -
#> 12    E2F   CycE           +
#> 13     Rb   CycA           -
#> 14    E2F   CycA           +
#> 15   CycA   CycA           +
#> 16  Cdc20   CycA           -
#> 17   Cdh1   CycA           -
#> 18 UbcH10   CycA           -
#> 19   CycD    p27           -
#> 20   CycE    p27           -
#> 21   CycA    p27           -
#> 22    p27    p27           +
#> 23   CycB    p27           -
#> 24   CycB  Cdc20           +
#> 25   CycA   Cdh1           -
#> 26    p27   Cdh1           +
#> 27  Cdc20   Cdh1           +
#> 28   CycB   Cdh1           -
#> 29   CycA UbcH10           +
#> 30  Cdc20 UbcH10           +
#> 31   Cdh1 UbcH10           -
#> 32 UbcH10 UbcH10           +
#> 33   CycB UbcH10           +
#> 34  Cdc20   CycB           -
#> 35   Cdh1   CycB           -
```

It is also possible to convert an `AttractorInfo` object into a data.frame using `attractor2dataframe`.

```
cc.attr <- getAttractors(cellcycle)
cycle <- attractor2dataframe(cc.attr)
cycle <- cycle[cellcycle$genes] #remove info columns
cycle
#>   CycD Rb E2F CycE CycA p27 Cdc20 Cdh1 UbcH10 CycB
#> 1    0  1   0    0    0   1     0    1      0    0
#> 2    1  0   0    1    1   0     0    0      0    0
```

```
#> 3    1  0   0    0    1   0     0    0      1    1
#> 4    1  0   0    0    1   0     1    0      1    1
#> 5    1  0   0    0    0   0     1    1      1    0
#> 6    1  0   1    0    0   0     0    1      1    0
#> 7    1  0   1    1    0   0     0    1      0    0
#> 8    1  0   1    1    1   0     0    1      0    0
```

Using this functions, it is possible to create a query that corresponds to a Boolean network. As we expect a cyclic attractor we will use `add.gquery.cycle()`.

```
q <- create.gquery.graph(topology, cellcycle$genes)
q <- add.gquery.cycle(q, cycle)
```

It is also possible to export the networks generated by `run.gquery()` directly to `BoolNet` with the option `return = "BoolNet"`. This option generates an iterator object, that returns the `BooleanNetwork` objects one by one using the function `iterators::nextElem()`.

For example, if we use the three node example network:

```
nets = run.gquery(q,return = "BoolNet")
#> Warning in boolnet.model.iterator(controller): iterators is not attached
iterators::nextElem(nets)
#> Boolean network with 3 genes
#>
#> Involved genes:
#> a b c
#>
#> Transition functions:
#> a = 0
#> b = ((((!a&b)&!c)|((a&!b)&!c))|((a&b)&!c))
#> c = (!b&c)
#>
#> Knocked-out and over-expressed genes:
#> a = 0
```

If there are no more available networks the `nextElem()` method will rise an error: `Error in obj$nextElem() : StopIteration`.

Other

It is possible to plot the network topology dataframe with the R package `igraph`. This dataframe can also be used to import and export the network topology to other resources like the python library `networkx` or to the software `Cytoscape`.

It is possible to export the network functions as an SBML file using the `BoolNet` function `toSBML()`.

References

Muñoz, S., Carrillo, M., Azpeitia, E., & Rosenblueth, D. A. (2018). Griffin: A Tool for Symbolic Inference of Synchronous Boolean Molecular Networks. Frontiers in genetics, 9, 39.