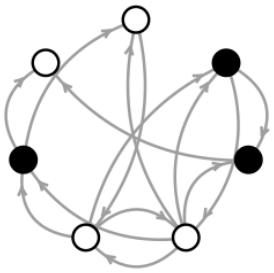
# Random Boolean Networks

## Quick Overview



A *boolean network* is a discrete dynamical system, originally proposed by Stuart Kauffman as a simple model of a genetic regulatory network in a living cell, where genes can switch each other on or off. It consists of a set of *N boolean variables*, each of which is considered a node in a directed graph with links from *K* neighbor nodes, and associated with a *boolean function* of arity *K*. At each time step, the value of each node's boolean variable is obtained by evaluating its function with the current values of its neighbor nodes as arguments. A cellular automaton with two-state cells is a special kind of boolean network where all the nodes use the same function and the links are all arranged in a regular bounded integer lattice structure; the Wolfram elementary cellular automata or the Game of Life are specific examples. In a *random* boolean network (RBN), the functions, links, and initial conditions are all randomly generated: the resulting behavior patterns can be surprisingly regular.

This NetLogo model will help you explore the behavior, orderly or chaotic, of some random boolean networks.

## Learning Targets and Prerequisites

This model is intended to develop in students an understanding of how a complex system with random parameters can exhibit regular behavior, a phenomenon sometimes called "emergence". It also provides an opportunity to explore what is called the "edge of chaos", where a system can transition between exhibiting orderly and disorderly behavior. Kaufmann's 1969 random boolean networks, and many variations on them, have been to subject of much study and debate within the systems biology community and beyond. Interpretation and critique of this model is the subject of several exercises, some of which are rather open-ended. With an accessible interactive implementation of the 'classic' RBN model, experienced students can conduct their own experiments and create their own variations.

Upon gaining familiarity with this model, students may expect to be able to adequately explain, in their own words, all the italicized terms in this document, and to answer most of the questions posed in the exercises. Students are expected to already have some familiarity with binary numbers and boolean logic.

## Essential Concepts

To fully appreciate this model, you will need to be familiar with a small handful of basic terminology from graph theory, probability, classical logic or boolean algebra, and discrete dynamics. Although that might sound like a lot, none of it is difficult. Here are some important vocabulary terms with very brief explanations. For more detailed information, and formal definitions, please consult the NetLogo code, or a good textbook!

A *directed graph* is a collection of *nodes* and *directed links*, which are ordered pairs of nodes: each link is *from* one node and *to* another. (No self-links are allowed in this particular model.)

By *random network*, we mean that each node has uniform probability of a link from another node. If $K = 1$, then this probability is $1/(N - 1)$. If $K = 2$, then the probability is $1/(N - 1)(N - 2)$, because we don't allow multiple links from one node to another. The functions and initial values of the nodes are, similarly, chosen with uniform probability from all possible functions and boolean values.

A *boolean variable* is a variable that can have only two states (or *values*): on/off, yes/no, 1/0, or true/false. A *boolean function* of *arity K*, where *K* is a whole number, is just a way to assign a boolean value to every possible combination of values of *K* boolean variables, taken in order. A few examples should help clarify. A *unary* function (of arity 1) is a way to take one value and consistently give back another. The variable you take (the *argument*) can have one of two values, and you must give back one of two values, so there are four ways to do this:

| | | | |
|---|---|---|---|
| $0 \mapsto 0$ | $0 \mapsto 0$ | $0 \mapsto 1$ | $0 \mapsto 1$ |
| $1 \mapsto 0$ | $1 \mapsto 1$ | $1 \mapsto 0$ | $1 \mapsto 1$ |

Each of these four unary boolean functions has a standard name:

"constant 0" "identity" "negation" "constant 1"

For two variables, there are $2^2 = 4$ possible arguments: 00, 01, 10, 11. A *binary* function (of arity 2) must evaluate each of these to either 0 or 1, so there are $2^4 = 16$ possible binary boolean functions. Some of these have standard names too, like "AND" or "OR". In general, there are $2^{2^K}$ boolean functions of arity *K*.

Because the *micro-state* (that is, the variable) of each of the nodes has one of two values, the entire *network state*, which is just the states of all *N* nodes together, has one of $2^N$ values. All of these network states taken together constitute the *state space* of any particular network, where the links and function assignments to nodes are fixed. Each network state has only one "next" state, and so a *path* or *trajectory* through the space, from one state to the next, will eventually reach a *cycle* of repeating states. However, network states may have more (or less!) than one "previous" state, so these cycles are the *attractors* of all their previous states, which are called the *basin* of the attractor. States which have no previous states in the space are sometimes called *garden-of-eden* states. Attractor cycles of length 1 are called *fixed points* in the state space. The number of states on a path between an initial state and a cycle is called the *transient* time.

**Model Mechanism and User Interface**

To visualize a random boolean network, we draw all the nodes arranged in a circle and color them either black (for 1) or white (for 0). The structural parameters of the network are -N-, the number of nodes in the graph; -K-, the number of links *to* each node; and -P-, the proportion (between 0 and 1) of nodes which are initially black. If -P- = 0, all nodes start black, and if -P- = 1, all nodes start white. There is also a switch, label-node-functions? which, when on, labels each node with a green number corresponding to the node's boolean function. These numbers are the decimal representations of the binary numbers formed by putting the function's value at each of

its possible arguments, in order. For example, the "negation" (or "NOT") function above has value 1 at argument 0, and 0 at 1. Lining up these two values, we see 10, which is binary for 2. So, if -K- = 1 and you see a node labelled "2", you know it's performing the NOT function. On the other hand, if -K- = 2 and the node is labelled "12", you can write 12 in four-bit binary as 1010, and interpret this as the function (equivalent to NOT *b*) that evaluates two arguments:

| *a* *b* | |
|---|---|
| 0 0 ↦ 1 | (where *a* and *b* are taken from the |
| 0 1 ↦ 0 | values of the two nodes with links to |
| 1 0 ↦ 1 | this one, and the node corresponding |
| 1 1 ↦ 0 | to *a* comes before that of *b*, counting clockwise from the top of the picture.) |

Clicking setup will make a new network with the specified parameters. The reset button assigns new random values to the nodes, clears the plot, and resets the tick counter. It doesn't change any of the links, so you can use it to see the behavior of the same network with different initial conditions. The go once button increments the network state (and the tick counter) by one step: all nodes first recalculate and then display their new values. The go button does the same but continuously, until clicked again.

The nodes-in-a-circle picture is helpful for seeing the links (as long as there aren't too many) but it doesn't show the history of the network state over time – its path through the state space. For this, we make another, smaller picture (labelled "network states over time") by plotting each black node as a black dot in a row corresponding to the network state at a particular time. It's as if we removed the links from the circle picture and 'unrolled' it clockwise from the top, so that the node at the top of the picture ends up on the far left of the row, and the node just to its left in the circle ends up on the far right of the row. The next state will be plotted as a new row just below the current one, so an attractor cycle is visible as a pattern of vertical lines or repeating dashes. The clear button clears the plot and resets the tick counter, without changing links or state.

The detect-cycles? switch, when on, causes each newly generated network state to be compared against all the previous states. If it matches one of these, then a cycle has been detected. The update process is stopped, and the cycle is highlighted on the network states plot with a red dot to the right of each network state in the cycle. The length of the cycle is printed in the Command Center, along with an identifying code: the sum of all the network states (as *N*-bit binary numbers) in the cycle. This will let you see which different conditions lead to the same attractors.

You can use the Command Center for calculations, too. Of course NetLogo does boolean logic, with true and false, so you can evaluate expressions like not (false or true). You can ask any node (as always, counted clockwise from the top of the picture) its color using, for example, black? node 3. And you can see all of these boolean values in order by typing net-state. That will give you a list of true and false values. To read these as 1's and 0's, you can type to-binary net-state. Or to see them as a decimal number, type decode net-state. (If there are more than 53 nodes in the network then NetLogo, because of its number system, will give you an approximate answer, which may be somewhat misleading.) These commands should be helpful when you want to keep track of initial conditions.

**Exercises**

1. Make a new network with $N = 10$ and $K = 1$, and set the detect-cycles? switch 'on'. Record the cycle and transient lengths for ten different runs of the same network with different initial states, using the reset button. How many different attractors did you find? Do the same with $K = 2$ and $K = 3$. Now try increasing $N$. Do you notice any patterns? Can you summarize your findings?

2. We might wonder how much the behavior of the network will change if its initial state changes by just a small amount. This is called 'damage spreading' or 'divergence', and it can take several forms: for example, we can flip a few bits of the initial state, rewire the inputs of just a few nodes, or change their functions. With $N = 50$, your favorite value of $K$, and $P = 0.1$, measure the cycle and transient lengths for ten runs. (Setting P = 0.1 means that only *about* 10% of nodes will be black when you press reset. If you want *exactly* five black nodes, you can set $P$ to 0 and type ask n-of 5 nodes [flip] into the Command Center.) How much did the behavior change? Now, with the same network, but $P = 0$, press reset and type export-world "init.csv" and then ask n-of 5 nodes [pick-random-sources]. Press go, and then type import-world "init.csv" to recover your initial conditions. Record the attractor data for ten runs like this from the all-white initial condition. How much variation did you find? Finally, leaving $P = 0$, type ask n-of 5 nodes [pick-random-function] and record data from ten runs, using the import-world command to reset between runs. Which of these three different kinds of 'damage' has the greatest effect on the behavior of the network?

3. One major challenge to the classic RBN model has to do with the update scheme: in 1997, Harvey and Bossomaier argued that the assumption that all nodes change their values at the same time was unjustified and unrealistic in the context of genetic regulatory networks. They introduced a revised model, the ARBN, with an *asynchronous* update scheme, where nodes update their states one at a time, in random order. As it happens, this asynchronous update pattern is built into the NetLogo ask command, and this NetLogo model uses a commonplace work-around to provide synchronized behavior. Look at the model's Code tab, and figure out how to make the update asynchronous (it's a very simple change). Then try the first exercise over again. What happened? Do any of your original findings still hold? There's also a hybrid model called DARBN (*Deterministic* Asynchronous RBN) which picks a random order for the nodes, but then always updates them in that exact order. For a little bit more of a programming challenge, try modifying the NetLogo model to use this DARBN update rule. What difference does it make?

4. Another straightforward extension of the model is to restrict the set of functions that the nodes can have. Part of the reason for doing that is that the constant functions tend to 'freeze' parts of the network after a few steps. (Can you explain why?) Modify the NetLogo code to be able to specify the code numbers of the possible functions. How differently does a $K = 1$ RBN with only the identity and negation functions (codes 1 and 2) behave? How about a $K = 2$ RBN with only the AND and OR functions (codes 8 and 14)?

5. Adapt the NetLogo model to use the Wattz – Strogatz 'small-world' or Barabási – Albert 'scale-free' network topologies, which are the subjects of their own Complexity Explorer lesson modules. In which ways does this affect the behavior of the networks?

## Advanced Topics

There are several different named classes of random boolean network, some of which are discussed above. The model is very general, however. Boolean networks (called 'switching circuits') are essentially the basis for conventional computer architecture, although these networks are very far from random. Of course, real organisms in their environments are highly non-random too! For this reason, RBNs under evolutionary selection pressures have been studied. To get an idea of how this might work, see the Genetic Algorithms lesson modules.

As an abstract model of the genetic regulatory networks which help govern the growth and maintenance of real organisms, RBNs have both strong and weak points relative to other modeling methods like systems of ordinary differential equations. They are not intended to be accurate models of the regulatory mechanisms of any particular organism. However, concepts developed in their study have proven helpful in the analysis of real genetic regulatory networks.

Much of Kauffman's original RBN research was focused around findings that for $K = 2$ networks, the number and lengths of attractor cycles was on average proportional to the square root of $N$, just as the number of cell types was believed to be proportional to the square root of the number of genes. Later work showed that both model and data were based on incomplete evidence: there are many fewer genes for complex organisms, and many more attractors for RBNs with high $N$.

## Bibliography


Drossel, B. (2007) Random Boolean Networks. arXiv:0706.3351v2.

Gershenson, C. (2004) Introduction to Random Boolean Networks. arXiv:nlin/0408006v3

Kauffman S, Peterson C, Samuelsson B, Troein C. (2003) Random Boolean network models and the yeast transcriptional network. Proc Natl Acad Sci U S A. 100(25): pp. 14796-9. http://www.ncbi.nlm.nih.gov/pubmed/14657375

Kauffman, S. (1969). Metabolic stability and epigenesis in randomly constructed genetic nets. Journal of Theoretical Biology, 22:437-467.