Expected Cycle Length in Random Boolean Networks of Connectivity 2

Patrick W. Yaner*
North Carolina State University

August 24, 1999

Abstract

In this paper we simulate random boolean networks of connectivity 2 and calculate the lengths of state cycles in hopes of gaining some insight into the distribution of cycle lengths for random boolean networks. Past results suggest that for networks with connectivity 2 state cycles are relatively short, but for higher connectivities the cycle length seems to grow exponentially with the size of the network. However, this property has yet to be verified analytically.

1 Introduction

In 1969 Stuart Kauffman [6] found a generalization of the classic McCulloch and Pitts neural network model useful in modeling what he called "genetic regulatory networks." He was modeling genes in organisms as large, randomly connected autonomous networks of binary switching elements, where the state of each element at one moment was a boolean function of the state of some number of other elements at the previous moment. In principle, perhaps a great many things could be modeled in this way; anything that can be thought of as a network of interconnected binary elements might potentially be modeled in the same way.

Boolean networks and their properties are quite simple concepts to describe mathematically, but, like so many problems in discrete mathematics, many of the most interesting questions about them turn out to be difficult or impossible to answer in general. Kauffman studied the state cycles of his genetic networks and suggested some properties of them that were interesting to him. As it has turned out, mathematical confirmation of these properties is far from trivial. In fact, in 30 years his results have yet to be confirmed in a rigorous mathematical fashion, despite all the work that has been done on the problem.

*This research done as part of the REU Program in Mathematics at Oregon State University. Thanks to Paul Cull for his guidance in completion of this project.

In this paper, first we cover some definitions and basic properties of boolean networks. We then discuss past results, including Kauffman's work and related work by other researchers. The subject of NP-Completeness and how it relates to what we're studying here is briefly covered, here. Next we discuss algorithms for randomly generating boolean networks with given connectivities and finding state cycles on these networks. Lastly, we discuss the results from one implementation of these algorithms and ideas.

2 Boolean Networks

2.1 Definitions

A boolean network is simply a network of boolean functions. If $x_i(t)$ is value on the i-th input line of some element of a network at time t then that element can be described by a discrete difference equation:

$$x_i(t+1) = f_i\big(x_1(t), x_2(t), \ldots, x_n(t)\big)$$

In particular, the function $f_i(x_1, \ldots, x_n)$ can be any boolean function.

An *autonomous* boolean network is one in which the inputs to every element come from other elements in the network (even, perhaps, the element itself), and the output of every element is connected to other elements in the network (possibly several at once) if it's connected at all. We can think of the *state* of the network at time t as simply the boolean vector consisting of the outputs from each of the elements at time t. Written in this way, the state S_t of an n-element network at time t is an element of boolean n-space, $\{0,1\}^n$ (which we will denote $\mathcal{B}^n$), and the state at time $t+1$ can be written as a difference equation:

$$S_{t+1} = \begin{pmatrix} x_1(t+1) \\ x_2(t+1) \\ \vdots \\ x_n(t+1) \end{pmatrix} = \begin{pmatrix} f_1(x_1(t), x_2(t), \ldots, x_n(t)) \\ f_2(x_1(t), x_2(t), \ldots, x_n(t)) \\ \vdots \\ f_n(x_1(t), x_2(t), \ldots, x_n(t)) \end{pmatrix} = F(S_t) \qquad (1)$$

Since each f_i can be any function of the form $f_i : \mathcal{B}^n \to \mathcal{B}$ the function F can be any boolean from boolean n-space into boolean n-space. That is, $F : \mathcal{B}^n \to \mathcal{B}^n$.

Also note that if, say, the i-th element only depends on the outputs of elements j and k, you can write it's function f_i as

$$f_i(x_1, x_2, \ldots, x_n) = g_i(x_j, x_k)$$

for some i, j, and k without loss of generality. It works both ways, however. That is, any function of, say, one or two variables can be written as a function of k variables, with $k \geq 2$. Thus, we can think of the *number of inputs* to an element in a network as the number of variables on which it's function actually depends. Note that in the definition of a particular network each variable will be associated with some element of the network.

Definition 2.1. The *connectivity* of a boolean network is the largest number of inputs that any one element receives. That is, if k_i is the number of inputs to element i, then the *connectivity* $k = \max\{k_i : i = 1, \ldots, n\}$ where n is the number of elements in the network.

Thus, we can think of every node in a boolean network as having exactly k inputs, where k is the connectivity of the network. Equation 1 then gives us

$$S_{t+1} = \begin{pmatrix} x_1(t+1) \\ x_2(t+1) \\ \vdots \\ x_n(t+1) \end{pmatrix} = \begin{pmatrix} f_1(x_{11}(t), x_{12}(t), \ldots, x_{1k}(t)) \\ f_2(x_{21}(t), x_{22}(t), \ldots, x_{2k}(t)) \\ \vdots \\ f_n(x_{n1}(t), x_{n2}(t), \ldots, x_{nk}(t)) \end{pmatrix} = F(S_t) \qquad (2)$$

where $x_{ij}(t)$ denotes the j-th input to element i, which can be any of $x_1(t)$ through $x_n(t)$.

2.2 Basic Properties

Since we can think of our set $\mathcal{B} = \{0, 1\}$ as equivalent to the simplest finite field $\mathbb{Z}_2$ (the integers modulo 2), let us now recall some information about this field. The two operations are addition modulo 2 and multiplication modulo 2, defined in the usual way, namely

$$\begin{array}{lll} 0+0=0 & & 0\cdot 0=0 \\ 0+1=1=1+0 & \text{and} & 0\cdot 1=0=1\cdot 0 \\ 1+1=0 & & 1\cdot 1=1 \end{array}$$

These operations correspond, respectively, to the boolean operations EXCLUSIVE OR (or XOR), and AND. The following proposition is an interesting property of this field that will be useful later.

Proposition 2.1. *Every function of k variables on $\mathbb{Z}_2$ (i.e. functions of the form $f : \mathbb{Z}_2^k \to \mathbb{Z}_2$) can be written as a polynomial with no variable raised to a power higher than 1.*

Proof. Now the cardinality of $\mathbb{Z}_2$ is $|\mathbb{Z}_2| = 2$ and $|\mathbb{Z}_2^k| = 2^k$, so the total number of possible functions of the form $f : \mathbb{Z}_2^k \to \mathbb{Z}_2$ is 2^{2^k}; there are 2^k elements in the domain of the function, and each one can map to one of 2 elements—0 or 1—and so we get

$$\underbrace{2 \cdot 2 \cdots 2}_{2^k} = 2^{2^k}$$

possible mappings.

Note that $0^2 = 0$ and $1^2 = 1$, so $x^2 = x$ for $x \in \mathbb{Z}_2$. If $x^n = x$ then $x^{n+1} = x^n \cdot x = x \cdot x = x^2 = x$, and so by induction $x^n = x \ \forall n \geq 1$.

Given this property, any polynomial of k variables could be reduced to a multinomial in k variables (an example with three: $\alpha_0 + \alpha_1 x_1 + \alpha_2 x_2 + \alpha_3 x_1 x_2 +$

$\alpha_4 x_3 + \alpha_5 x_1 x_3 + \alpha_6 x_2 x_3 + \alpha_7 x_1 x_2 x_3$ for some coefficients $\alpha_i \in \mathcal{B}$). In general, there will be $\binom{k}{i}$ terms of i variables. Each one gets one coefficient, and so there will be

$$\binom{k}{0} + \binom{k}{1} + \cdots + \binom{k}{k} = 2^k$$

coefficients in the multinomial. Each coefficient can be either 0 or 1, and so there are 2^{2^k} different multinomials on k variables. Since two multinomials of the same form and the same number of variables have different coefficients if and only if their values differ for at least one setting of their variables, and since we're counting the number of choices for coefficients, each function of k variables on $\mathbb{Z}_2$ corresponds to a unique multinomial. □

Because the state of a boolean network of size[1] n at any moment in time is simply a vector in boolean n-space $\mathcal{B}^n$, there are 2^n possible states of the network. This is a finite number—and the network itself is a deterministic system (that is, there is exactly one possible next state for any given state of the network)—and so if the network is started in some state and run for long enough, it will eventually return to a state S_t that it was in previously. It will then repeat the sequence of states following S_t. In general, from any given starting state (which can be any possible state), the network will transition through some number of transient states (possibly none) before entering a cycle.

Given that the only behavior of the network possible is for it to enter a cycle after some number of transient states, we define the following:

Definition 2.2. Given an n-element boolean network F, with $S_{t+1} = F(S_t)$, a *cycle* is an ordered set of states $\mathcal{C} = \{S_1, S_2, \ldots, S_j\}$ such that $S_1 = F(S_j)$, $S_2 = F(S_1)$, etc. The *length* ℓ of a cycle $\mathcal{C}$ is the cardinality of the set. That is, $\ell = |\mathcal{C}|$.

From determinism it should be clear that for a particular network any two given cycles $\mathcal{C}_1$ and $\mathcal{C}_2$ are either disjoint or equal. In addition any given state is either an element of some cycle or a transient state, but not both. Thus, the total number of states involved in cycles in a network may not necessarily be all possible states of that network, *i.e.* in general if $\mathcal{C}_i$ are all distinct (meaning disjoint) cycles in a boolean network of size n then

$$\bigcup_{\text{all } i} \mathcal{C}_i \subseteq \mathcal{B}^n$$

$$\text{but} \qquad \mathcal{B}^n \nsubseteq \bigcup_{\text{all } i} \mathcal{C}_i$$

To illustrate some of these properties, we turn now to a simple example of a boolean network with 3 elements taken from Kauffman [7, pages 189–190].

[1] throughout this paper *size* will refer to the number of elements in a network

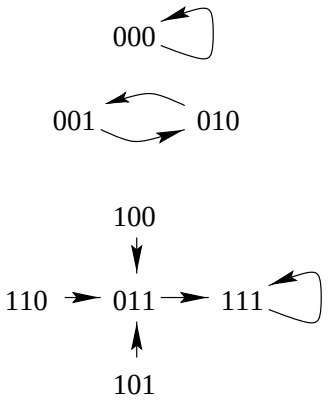

Figure 1: State-transition diagram for an example boolean network

Example 2.1. Consider a boolean network of three elements defined by

$$S_{t+1} = \begin{pmatrix} x_1(t+1) \\ x_2(t+1) \\ x_3(t+1) \end{pmatrix} = \begin{pmatrix} x_2(t)x_3(t) \\ x_1(t) + x_3(t) + x_1(t)x_3(t) \\ x_1(t) + x_2(t) + x_1(t)x_2(t) \end{pmatrix} = F(S_t) \qquad (3)$$

The operations that the three elements compute are the functions corresponding to, respectively, the boolean operations of AND, OR, and OR. There are $2^3 = 8$ possible states of this network. For our purposes here, we will denote a state vector (x_1, x_2, x_3), where $x_i \in \mathcal{B}$, by the binary string $x_1x_2x_3$.

If we start the machine in the state 000, we can see from 3 that $F(000) = (0 \cdot 0)(0 + 0 + 0 \cdot 0)(0 + 0 + 0 \cdot 0) = 000$ (we will write simply $000 \mapsto 000$), and so we have the cycle $\mathcal{C}_0 = \{000\}$, a cycle of length 1. If we start the machine in state 001, we see that $001 \mapsto 010 \mapsto 001 \mapsto \cdots$ and so we have the cycle $\mathcal{C}_1 = \{001, 010\}$, a cycle of length 2. From the state 100 we get $100 \mapsto 011 \mapsto 111 \mapsto 111 \mapsto \cdots$, which gives us the cycle $\mathcal{C}_2 = \{111\}$, a cycle of length 1. The only remaining states are 110 and 101. From these we get $110 \mapsto 011 \mapsto 111 \mapsto 111 \mapsto \cdots$, which is $\mathcal{C}_2$ again, and $101 \mapsto 011 \mapsto 111 \mapsto \cdots$, which is also $\mathcal{C}_2$.

Notice, here, that 100, 110, and 101 each lead in one step to the state 011, which then leads in one step into the cycle $\mathcal{C}_2$. Also, all of the states 100, 110, 101, and 011 are transient states. As described above, if the machine passes through these states it only enters them once, and then passes on into a cycle that does not involve that state. The total number of states involved in cycles, here is 4 (two cycles of length 1 plus one cycle of length 2). There are five states that lead to cycle $\mathcal{C}_2$ (including the one state in the cycle itself), and so for this network the *expected cycle length* $E(\ell) = \widehat{\ell} = \frac{5}{8} \cdot 1 + \frac{2}{8} \cdot 2 + \frac{1}{8} \cdot 1 = \frac{5+4+1}{8} = 1\frac{1}{4}$ (note that it is not $\frac{1+1+2}{3} = 1\frac{1}{3}$). The state diagram associated with this network appears in figure 1.

2.3 Previous Results

Kauffman [6] suggested that for boolean networks constructed at random with a connectivity of 2 (and a given size), the median of the cycle length was approximately $\sqrt{n}$, where n is the number of elements. In addition, he suggested that the distribution of cycle lengths was not symmetric, and specifically that short cycles tended to dominate.

A number of researchers have tried to understand the behavior of random boolean networks and the $k = 2$ case in particular. Cull [2] developed a linearization technique for boolean networks and showed how it could be used to analyze the behavior of random networks, Sherlock [9, 10] attempted to analyze Kauffman's results [6] mathematically, and Fogelman-Soulie, et al. [1, 5] have attempted to understand specific sorts of random networks, to cite a few examples.

Nicole Mayer [8] attempted to verify—among other things—that the average cycle length for $k = 2$ networks is $\hat{c}\sqrt{n}$ for some $\hat{c} > 0$. Computer simulation of random boolean networks of sizes 2 through 40 was used for this, but the results were inconclusive.

Although the most general case—namely, that in which k is equal to n, also called the *totally connected* case—has been described rigorously, in 30 years nobody has rigorously confirmed or denied Kauffman's original conjecture, or, for that matter, described the probabilistic behavior of random boolean networks with connectivities $k < n$ in a rigorous manner, despite all the work that has been done on the problem. For instance, Some researchers have found evidence confirming Kauffman's claim (Kauffman [7] cites one or two examples), and others find evidence against it or are otherwise unable to confirm the results (such as Sherlock [10]).

This paper is not an attempt to actually solve the problem, but just to produce more computer simulations of boolean networks in hope of gaining more insight into the problem.

2.4 Boolean Networks and NP-Completeness

It should be mentioned at this point that many of the questions we would like to answer about boolean networks are NP-Complete. Cull [4] gives some examples of NP-Complete problems in the specific case of neural networks using the classic McCulloch and Pitts model. This neural network model uses neurons that compute linear threshold functions, which are specific types of boolean functions. As mentioned in §1, the model we're using here is a generalization of that model to allow the "neurons" to now compute any boolean function.

A brief word about NP-Completeness would be appropriate before mentioning which problems can be classified as such. The following description is taken from Cull [4]. Those problems which possess an algorithm with run time[2] bounded by a polynomial in the size of the input[3] are, in general, easy problems.

[2] number of steps required to complete the algorithm

[3] the data on which the algorithm is being applied

Let us call such algorithms *fast*. Problems without such an algorithm are hard. In general it may be difficult to show that a problem has no fast algorithm, but if one can show that the problem is the hardest problem within some well-defined class the problem can still be called hard. For example, NP is the class of problems which have fast algorithms if you write your algorithms to allow for nondeterministic computation (you might think of this as supposing, for instance, your computer has a magical "guess" instruction which always chooses the correct alternative from a set of possibilities). The hardest problems in NP are called NP-Complete, and NP-Complete problems are generally considered too hard to be solvable by some by any practical algorithm.

It should be pointed out that although there are precise ways to think about what a "practical algorithm" is, it has never been proven that any of the problems in NP (including those that are NP-Complete) have no practical algorithm associated with them. In other words, it has never been proven that NP-Complete problems do *not* have fast algorithms in the conventional sense (*i.e.* no "guess" instruction).

Cull [4] mentions the following problems (among others) about neural networks which are NP-Complete:

- Given an autonomous neural network and a specified state S, is there a state such that if the network is started in this state then it will enter the specified state S?
- Given an autonomous network, are there transient states?

These problems both seem easy, and for particular networks they may be easy, but in general they are not. If these questions are difficult to answer for neural networks, then surely generalizing the model to boolean networks does not make things any easier. Indeed, these seem like questions we might want to answer if we were to calculate something like a probability distribution for cycle length in random boolean networks.

3 Computer Simulation

In order to generate a boolean network at random (with a given connectivity), we would need to generate the function F from equation 2 in §2.1. That function is defined by the component functions f_i and the n sequences of k variables on which each f_i depends. Thus, we have two major steps in generating a boolean network at random with a given size and connectivity.

From Proposition 2.1, any boolean function of k variables can be represented as a multinomial in the variables. Since that multinomial will be uniquely determined by it's coefficients (provided they, two, are boolean values), to generate a function of k variables at random, we need only generate the 2^k coefficients at random.

Now, to determine which k variables (*i.e.* elements) each function depends on, we need only generate n finite sequences of k integers from the set $E_n =$

$\{1, 2, \ldots, n\}$. Although in general we might like to generate each sequence by choosing at random from the set without replacement, to do so would add complexity—and thus run time—to the algorithm. Note that if we are dealing with, say, a 100 element network, the probability that we will choose the same number twice (assuming independence, of course, which is a reasonable assumption in this case) is $\frac{1}{100} \cdot \frac{1}{100} = \frac{1}{10000} = 0.0001$, and so we should be able to ignore the issue without noticeably altering the results.

Hence, to generate a boolean function of size n and connectivity k, we do the following:

1. Generate n sequences of 2^k boolean values to serve as the coefficients for the functions f_i defining our network. Let α_j^i be the j-th coefficient of the i-th function.

2. Generate n sequences of k integers from the set E_n (mentioned above). Call η_j^i the j-th number in the i-th sequence.

Now, say $k = 2$. Given a state S_t (where $S_t(i)$, now, denotes the state of the i-th element at time t), to evaluate the state of element i at time $t+1$ we would compute f_i as follows:

$$S_{t+1}(i) = \alpha_1^i \ + \ \alpha_2^i S_t(\eta_1^i) \ + \ \alpha_3^i S_t(\eta_2^i) \ + \ \alpha_4^i S_t(\eta_1^i) S_t(\eta_2^i) \tag{4}$$

Where multiplication and addition are computed mod 2. Note that this is equivalent to using AND and XOR (exclusive or) for multiplication and addition, respectively. The advantage of this is that on most computers, AND and XOR are functions that can be evaluated in a single instruction, which improves the speed of our computations (as opposed to using mod 2 arithmetic). If we compute $S_{t+1}(i)$ as above for $i = 1, \ldots, n$, then we will have computed the entire next state of our network.

Since we are interested in cycle lengths, we need to somehow find cycles on our networks. Note that since there are 2^n states in any n-element net, finding every cycle would get out of hand very quickly for even reasonable values of n (for example $2^{100} \approx 10^{30}$). Thus, we want to take a random sample of the cycles in a network. We could start from a random state and find the cycle this state leads to. By repeating this some number of times for a given (randomly constructed) network we should have something like a random sample.

So, given a starting state A, how do we find a cycle and calculate it's length? First we need to find a state actually in the cycle, and then from that state we can iterate until we arrive back at that state again. If $\mathcal{C} = \{s_1, s_2, s_3\}$ is a cycle of length three, then starting from s_1 we will get $s_1 \mapsto s_2 \mapsto s_3 \mapsto s_1$. The cycle had a length of 3 and it took 3 iterations to return to state s_1. In general, from any state in any cycle of length ℓ it takes exactly ℓ iterations of the function F to return to that state. So how do we find a state actually in the cycle? Starting from a state A and letting $B = F(A)$, we repeat the following until $B = A$:

1. Let $B = F(F(B))$

2. Let $A = F(A)$

For example, let us say the state $s_\alpha \mapsto s_\beta \mapsto s_1$ from our cycle $\mathcal{C}$ above in some network. Then if we start from $A = s_\alpha$ and $B = s_\beta$ we will get, after the first iteration of our loop, $B = s_2$ ($s_\beta \mapsto s_1 \mapsto s_2$) and $A = s_\beta$ ($s_\alpha \mapsto s_\beta$). Next we have $B = s_1$ ($s_2 \mapsto s_3 \mapsto s_1$) and $A = s_1$ ($s_\beta \mapsto s_1$) and so $B = A$ and we stop. We have now found a state in the cycle. Note after each repetition B is one iteration further from A than it was before, and so eventually B will equal A if there is a cycle (and there always is, from §2.2, recall). This is not a proof that the algorithm will always work (and always terminate), but with a bit of thought hopefully it will be clear that this is the case.

The source code to a program implementing these algorithms is given in the appendix. Various versions of that program were used to gather the results that appear in the next section.

4 Results

As mentioned above, Mayer [8] computed average cycle lengths for simulated random boolean networks with $k = 2$ and for n running from 2 to 40. That data was inconclusive, and so an attempt was made, here, to extend that data to larger values of n with the hope that an obvious pattern might emerge.

Figure 2 shows a graph of average cycle length as a function of network size on a logarithmic scale. A graph of $\sqrt{n}$ is shown for comparison. The curve in this graph clearly does not even remotely resemble any smooth function such as $\sqrt{n}$. Comparison of this graph with results from other runs of the program revealed at least two things: first, the spikes and dips are not necessarily occurring in the same places in every run and, second, the extreme dips and spikes get larger as n gets larger. In short, although it seems clear that cycle length is increasing with network size, the plot appears random and not much more information can be read into the data than that. The only conclusions that suggest themselves, then, from this data is that we really cannot draw any conclusions from this data. The important insight behind Kauffman's original suggestion that the median cycle length for $k = 2$ nets is $\sqrt{n}$ is that the cycle length is far shorter than for higher connectivities, where it seems to be growing exponentially as n increases (see Cull [4]), but this data cannot even confirm that.

Perhaps more could be learned by looking at the distribution itself. So the next thing we might look at are histograms of the cycle length for certain values of n. Figure 3 shows four histograms of the cycle length (from 2000 total cycles per network size) for networks of sizes 50, 100, 150, and 200. It seems from that graph that the majority of the cycles consist of 10 states or less for n up as high as 200. From this we can see that at $n = 50$ almost no cycles appear longer than about $\ell = 30$, but as n ranges up to 200 more long cycles start to show up. However, the median cycle length for $n = 200$ appears to be less than 10 from the graph. Variance was not calculated in these runs, but it appears that although small cycles dominate, there are an increasing number of long cycles

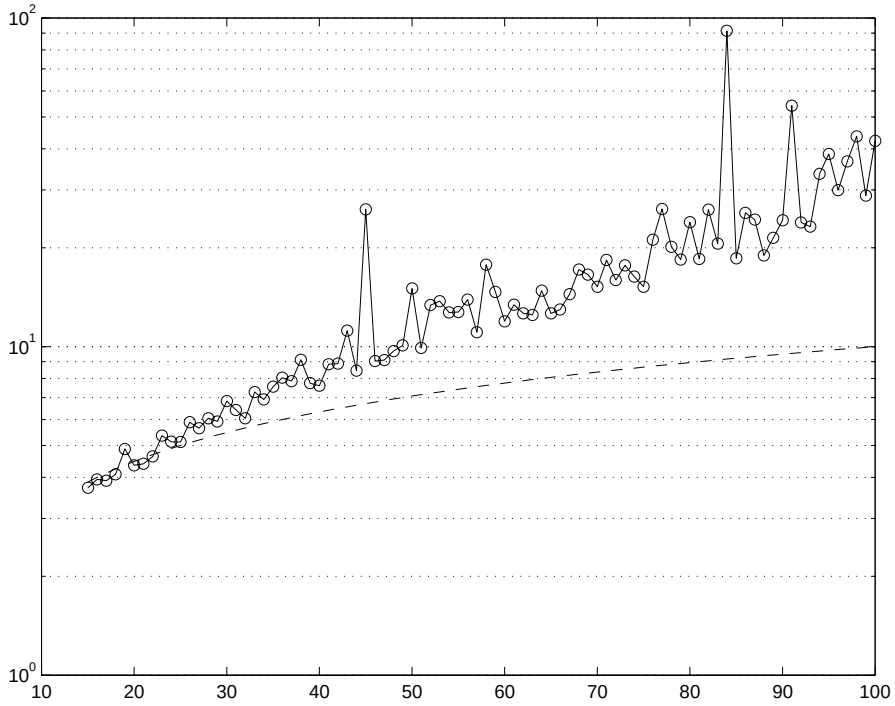

Figure 2: Plot of the cycle length on a logarithmic scale as a function of the size of the network. The dotted line is a graph of $f(n) = \sqrt{n}$ for comparison

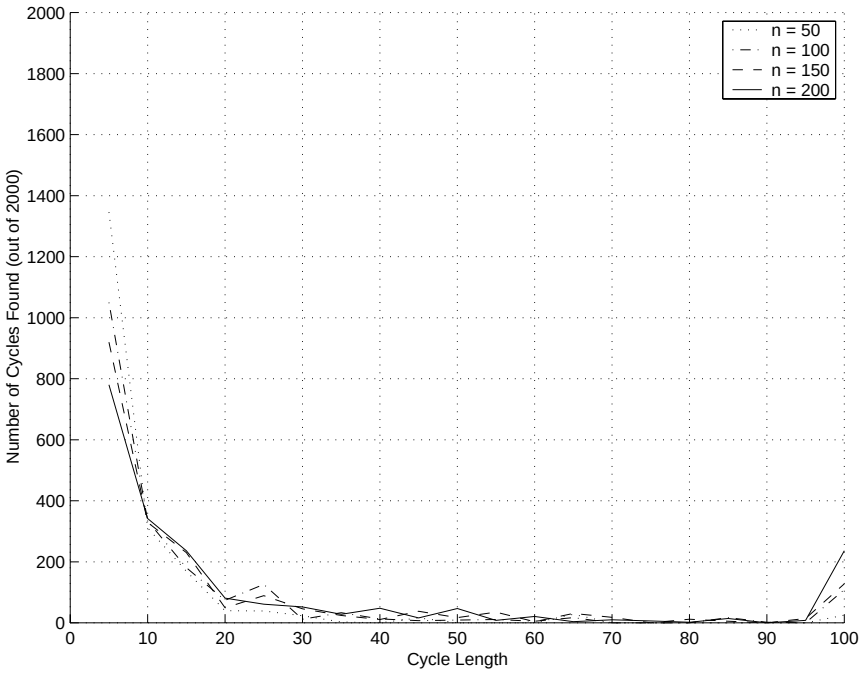

Figure 3: Distribution of cycle lengths for networks of sizes 50, 100, 150, and 200 (histograms of cycle lengths—2000 total for each size network)

as n increases, and perhaps that is why calculations of the average vary as much as they do.

5 Conclusion

The data from these simulations is still incomplete; it is difficult to draw conclusions from the information presented herein. It does seem to suggest that most, if not all, of the cycles are of length less than n, or at least $2n$. In a 200 element network there are $2^{200} \approx 1.6 \cdot 10^{60}$ possible states, but on average these networks seem to be limiting themselves to cycling amongst 100 states or less. This is an incredibly small fraction of the number of states available. This would be a significant result if it were true, but all we have here are results from simulations, and statistics such as the standard deviation and confidence intervals were not calculated from the data presented.

Perhaps simulating networks of sizes up to, say $n = 10,000$ would produce some information from which some conclusions could be drawn. The dominance of very small cycles (less than $\sqrt{n}$) is fascinating, and so it would be interesting to see if this trend continues for more reasonable values of n (note that the situations to which this model applies may involve thousands of elements or more in a network).

And, of course, a more detailed analysis of what numbers we do have is still in order.

References

[1] H. Atlan, F. Fogelman-Soulie, J. Salomon, and G. Weisbuch. Random boolean networks. *Cybernetics and Systems*, 12:103–121, 1981.

[2] P. Cull. A matrix algebra for neural nets. In G. Klir, editor, *Applied General Systems Research*, pages 563–573. Plenum, New York, 1978.

[3] P. Cull. Dynamics of random neural nets. In N. Boccara, E. Goles, S. Martinez, and P. Picco, editors, *Cellular Automata and Cooperative Systems*, pages 111–120. Kluwer, Netherlands, 1993.

[4] P. Cull. NET PROPHET: McCulloch and developments from his neural net model. Technical Report 96-20-01, Department of Computer Science, Oregon State University, February 1996.

[5] F. Fogelman-Soulie, E. Goles-Chacc, and G. Weisbuch. Specific roles of the different boolean mappings in random networks. *Bulletin of Mathematical Biology*, 44:715–730, 1982.

[6] S. A. Kauffman. Metabolic stability and epigenesis in randomly constructed genetic nets. *Journal of Theoretical Biology*, 22:437–467, 1969.

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

[8] N. Mayer. Random neural nets with small connectivities. Oregon State University REU in Mathematics program research, August 1993.

[9] R. A. Sherlock. Analysis of the behavior of kauffman binary networks—i. state space description and the distribution of limit cycle lengths. *Bulletin of Mathematical Biology*, 41:687–705, 1979.

[10] R. A. Sherlock. Analysis of the behavior of kauffman binary networks—ii. the state cycle fraction for networks of different connectivities. *Bulletin of Mathematical Biology*, 41:707–724, 1979.

A Program Source

This is the source code of one version of the program used to produce the results in this paper. It is written in the 'C' programming language and was compiled using the GNU C Compiler, `gcc`. The functions `rrand()` and `rseed()` were written so that the usual ANSI-standard system calls `rand()` and `srand()` could be replaced by better a better random number generator that might happen to exist on a particular system (this was, in fact, done with other versions of the program used to get the results that appear above).

```
/*************************************************************
 *************************************************************
 *************************************************************
 *                                                           *
 * main.c - main program.                                    *
 *                                                           *
 * Generates boolean networks of sizes from 15 to the        *
 * given first argument and of the connectivity given as     *
 * the second argument, generating 200 of each size and      *
 * finding 10 cycles in each one and outputting a matrix     *
 * of histograms of the cycle lengths in a format            *
 * understandable by Matlab.                                 *
 *                                                           *
 * Patrick Yaner - Mon Aug 2 1999                            *
 *                                                           *
 *************************************************************
 *************************************************************
 *************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include "boolnet.h"
```

```
#include "rrand.h"

void histogram(int buckets[])
{

    int i;
    int len = findcycle();

    for (i = 1; i < 20; ++i) {
        if ( len < (5 * i) ) {
            buckets[i-1] += 1;
            break;
        }
    }

    if ( 20 == i ) buckets[19] += 1; /* it's >= 95 */

}

void getcycles(int n, int k, int buckets[])
{

    int i, j;

    for (i = 0; i < 200; ++i) {
        newbnet(n,k);
        for (j = 0; j < 10; ++j)
            histogram(buckets);
    }

}

void calcavgs(int k, int maxN)
{

    int n;
    int i;
    int buckets[20] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                       0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

    printf("X = [");
    for (n = 15; n < maxN; ++n) printf("%i, ",n);
    printf("%i];\nY = [",maxN);
    for (n = 5; n < 100; n = n + 5) printf("%i, ",n);
    printf("%i];\nH = [",100);
```

```
    for (n = 15; n < maxN; ++n) {
        getcycles(n,k,buckets);
        for (i = 0; i < 19; ++i) {
            printf("%i, ",buckets[i]);
            buckets[i] = 0;
        }
        printf("%i;\n     ",buckets[i]);
        buckets[i] = 0;
    }

    getcycles(n,k,buckets);
    for (i = 0; i < 19; ++i)
        printf("%i, ",buckets[i]);
    printf("%i];\n",buckets[i]);

}

int main(int argc, char **argv)
{

    int k, maxN;

    if ( 3 == argc ) {
        maxN = atoi(argv[1]);
        k = atoi(argv[2]);
    } else if ( 3 == argc ) {
        maxN = atoi(argv[1]);
        k = 2;
    } else if ( 1 == argc ) {
        maxN = 50;
        k = 2;
    } else {
        fprintf(stderr,"too many arguments\n");
        return 1;
    }

    rseed();

    calcavgs(k,maxN);

    return 0;

}


/************************************************************
```

```
 ************************************************************
 ************************************************************
 *                                                          *
 * boolnet.h - header for use of a random boolean network   *
 *                                                          *
 * Patrick Yaner - Mon Aug 2 1999                           *
 *                                                          *
 ************************************************************
 ************************************************************
 ************************************************************/

#ifndef _BOOLNET_H
#define _BOOLNET_H

#ifndef NULL
#define NULL      0   /* just in case */
#endif /* !NULL */

#define MAX_NODES  512        /* size limit for our networks */
#define MAX_STATES 1073741824 /* upper limit on cycle length */

#define C0(i)  (coeff[i] & 0x01)       /* these are the */
#define C1(i) ((coeff[i] & 0x02) >> 1) /* coefficients of the */
#define C2(i) ((coeff[i] & 0x04) >> 2) /* multinomials that */
#define C3(i) ((coeff[i] & 0x08) >> 3) /* compute the next */
#define C4(i) ((coeff[i] & 0x10) >> 4) /* state of each node */
#define C5(i) ((coeff[i] & 0x20) >> 5)
#define C6(i) ((coeff[i] & 0x40) >> 6)
#define C7(i) ((coeff[i] & 0x80) >> 7)

#define I1(i) inputs[i][0]             /* these are the first, */
#define I2(i) inputs[i][1]             /* second, and third */
#define I3(i) inputs[i][2]             /* inputs of node 'i' */

#define SQUARE(X) (X*X)

void newbnet(int mu, int kappa);
/* generate a new boolean network in the given struct with
 * mu elements and a connectivity of kappa, where kappa is
 * either 2 or 3
 */

int findcycle();
/* returns length of the cycle starting from a random state */

#endif /* !_BOOLNET_H */
```

```
/************************************************************
 ************************************************************
 ************************************************************
 *                                                          *
 * boolnet.c - implementation of a randomly generated       *
 *             boolean network                              *
 *                                                          *
 * Patrick Yaner - Mon Aug 2 1999                           *
 *                                                          *
 ************************************************************
 ************************************************************
 ************************************************************/


#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/types.h>
#include <sys/time.h>
#include "boolnet.h"
#include "rrand.h"

/***** some static functions used only in this module *****/

static void randcoeff();
  /* randomly generate the coefficients */

static void inputvectors();
  /* randomly connect the net */

static unsigned char * F(unsigned char s[]);
  /* computes the next state from s (in place; returns s) */

static void randvector(unsigned char s[]);
  /* generates a random state in s */

static void setvector(unsigned char x[], unsigned char y[]);
  /* set y = x */

static int  samevector(unsigned char A[], unsigned char B[]);
  /* returns true (1) if A = B and false (0) otherwise */

/***** The actual boolean network *****/

/* These variables are global only to this module */
```

```
static int n; /* size of the net */
static int k; /* connectivity of the net */

static int inputs[MAX_NODES][3];       /* connection graph */
static unsigned char coeff[MAX_NODES]; /* coefficient vectors */

void newbnet(int mu, int kappa)
{

    n = mu;
    k = kappa;

    randcoeff();
    inputvectors();

}

int findcycle()
{

    /* This uses the little trick of starting with two
     * states A, B equal to each other and, in a loop, iterating
     * B twice (i.e. B = F(F(B)) ) and A once
     * (A = F(A) ), checking for equality of the two states
     * after each iteration to find a loop. Once a cycle is
     * found the state is iterated until it is reached again,
     * and the number of required iterations is returned as
     * the length of the cycle (since it is)
     */

    unsigned char A[MAX_NODES];
    unsigned char B[MAX_NODES];
    int len = 0;
    int i;

    randvector(A);  /* randomly generate a starting state in A */
    setvector(B,A); /* let B = A */
    F(B);           /* and iterate it once. (now B=F(A)) */

    /* First we need to find a cyclic state.
     * When this loop is done A will equal B
     * and we'll be in a cyclic state
     */
    for (i = 0; (i < MAX_STATES) && !samevector(A,B); ++i) {
        F(F(B));
```

```
        F(A);
    }

    /* Now we have A = B again, so we've found the cycle
     * next we determine the length by iterating around
     * the cycle once and counting how many iterations
     * it takes.
     */
    F(B);
    for (len = 1; !samevector(A,B); ++len)
        F(B);

    return len;

}

static void randcoeff()
{

    /* here we'll generate 8 constants regardless of the
     * connectivity on the assumption that if k=2 (and thus
     * only 2^2 = 4 constants are needed) the first four
     * bits will be just as random as the last four. Since
     * k can be no more than 3 (requiring 2^3 = 8 constants)
     * no more than 8 constants are required. ( => 8b = 1B
     * is required for storage, where 'b' means 'bits' and 'B'
     * means 'bytes').
     */

    int i;

    for (i = 0; i < n; ++i)
        coeff[i] = (unsigned char) rrand(255);

}

static void inputvectors()
{

    /* here we generate a n x k array of integers at
     * random (called inputs) where inputs(i,j) (in C
     * written as inputs[i][j]) is the node from which
     * the j-th input to the i-th node is coming.
     */

    int i, j;
```

```
    for (i = 0; i < n; ++i)
        for (j = 0; j < k; ++j)
            inputs[i][j] = rrand(n-1);

}

static unsigned char * F(unsigned char s[])
{

    /* This actually evaluates the function F (where
     * S_t+1 = F(S_t) defines the network). It first
     * calculates the new state in a temporary variable
     * called 'rVal' and then it copies this value back
     * into the argument. Arrays are always pass-by-
     * reference in C since an array is really just
     * a pointer to the first element of the array.
     */

    int  i;
    unsigned char rVal[MAX_NODES];

    for (i = 0; i < n; ++i) {
        switch (k) {
        case 2:  rVal[i]  = C0(i);
                 rVal[i] ^= C1(i) & s[I1(i)];
                 rVal[i] ^= C2(i) & s[I2(i)];
                 rVal[i] ^= C3(i) & s[I1(i)] & s[I2(i)];
                 break;

        case 3:  rVal[i]  = C0(i);
                 rVal[i] ^= C1(i) & s[I1(i)];
                 rVal[i] ^= C2(i) & s[I2(i)];
                 rVal[i] ^= C3(i) & s[I1(i)] & s[I2(i)];
                 rVal[i] ^= C4(i) & s[I3(i)];
                 rVal[i] ^= C5(i) & s[I1(i)] & s[I3(i)];
                 rVal[i] ^= C6(i) & s[I2(i)] & s[I3(i)];
                 rVal[i] ^= C7(i) & s[I1(i)] & s[I2(i)] &
                                    s[I3(i)];
                 break;

        default: fprintf(stderr,"AAAARGH!!!!\n"); exit(1);
        }
    }

    for (i = 0; i < n; ++i)
```

```
        s[i] = rVal[i];

    return s;

}

static void randvector(unsigned char s[])
{

    /* generates a random boolean vector in the given
     * argument
     */

    int i;

    for (i = 0; i < n; ++i)
        s[i] = rrand(1);

}

static void setvector(unsigned char x[], unsigned char y[])
{

    /* given two vectors x and y, here we set y = x */

    int i;

    for (i = 0; i < n; ++i)
        x[i] = y[i];

}

static int samevector(unsigned char x[], unsigned char y[])
{

    /* this function returns true (1) if x = y and false (0)
     * otherwise. The loop below short-circuits the moment
     * a difference is encountered (we get a very small
     * increase in run time on average from doing this)
     */

    int retval = 1;
    int i;

    /* x[0] = y[0] and x[1] = y[1] and ... */
    for (i = 0; (i < n) && (retval); ++i)
```

```
        retval = (x[i] == y[i])? 1 : 0;

    return retval;

}

/************************************************************
 ************************************************************
 ************************************************************
 *                                                          *
 * rrand.h - wrapper for rand() and srand() system calls    *
 *                                                          *
 * srand() seeds the random number generator with the       *
 * current clock value (from time()), and rrand(n) returns  *
 * a random long integer between 0 and n inclusive.         *
 *                                                          *
 * Patrick Yaner - Tue Jul 20 1999                          *
 *                                                          *
 ************************************************************
 ************************************************************
 ************************************************************/

#ifndef _RRAND_H
#define _RRAND_H

#ifndef NULL
#define NULL 0
#endif /* !NULL */

extern void rseed();
extern int rrand(int range);

#endif /* !_RRAND_H */

/************************************************************
 ************************************************************
 ************************************************************
 *                                                          *
 * rrand.c - wrapper functions for the srand() and rand()   *
 *           system calls.                                  *
 *                                                          *
 * Patrick Yaner - Tue Jul 20 1999                          *
 *                                                          *
 ************************************************************
 ************************************************************
 ************************************************************/
```

```
#include <stdlib.h>
#include <limits.h>
#include <math.h>
#include <sys/types.h>
#include <sys/time.h>
#include "rrand.h"

void rseed()
{

    unsigned int seed;

    seed = (unsigned int) time(NULL);
    srand(seed);

}

int rrand(int range)
{

    int rval;
    double temp;

    temp = rand() / (double) INT_MAX;
    temp *= range;
    rval = (int) rint(temp);

    return rval;

}
```