Ratcheted Random Search for Self-Programming Boolean Networks

Lee Spector[0000-0001-5299-4797]

Abstract Random Boolean networks exhibit a variety of properties that are characteristic of natural adaptive systems. This suggests that Boolean networks may provide a useful substrate for artificial systems that adapt to solve specified problems. In this chapter I describe an investigation of the capacity of Boolean networks to represent solutions to computational problems and the efficacy of simple stochastic algorithms for finding problem-solving Boolean networks. Because prior work noted that the properties of Boolean networks depend on the set of permitted gate types, I consider here networks constructed only of programmable gates that can be configured to act as any 2-input Boolean gate. Networks of these gates are self-programming because the function of each gate is determined dynamically from its configuration inputs. I consider search algorithms that find self-programming networks through processes of random rewiring, random gate additions, random gate deletions, and a ratchet mechanism that only permits moves in the search space that maintain or increase training case coverage. I present networks found by this algorithm and discuss the implications of these results for future work.

1.1 Introduction

Starting in 1969, Stuart Kauffman and subsequently others have explored the properties of randomly-wired networks of Boolean gates. Much of this work has been motivated by an interest in illustrating features of genetic regulatory networks and other complex biological systems [7, 21, 8, 9, 3]. Among the features of random Boolean networks that have been the focus of this work are the spontaneous emergence of order, attractors with modest cycle lengths in spite of the enormous state spaces in which they occur, dynamics that can be tuned to the “edge of chaos” to facilitate adaptation, and resilience to perturbation.

Lee Spector
Amherst College, Amherst, Massachusetts, USA, e-mail: `lspector@amherst.edu`

One suggestion raised by this research, and explored to some extent in prior work, is that Boolean networks may be useful as targets of adaptive processes that aim to produce executable structures that solve specified problems. For example, Kauffman considered the effects of wire-moving mutations and the prospects for adaptive walks through the space of wirings to find attractors with particular patterns of activation for specified subsets of gates [8]. Lemke et al. presented a more thorough numerical study of adaptation in populations of Boolean networks evolved using genetic algorithms [10], again in the context of finding networks with attractors matching specified patterns. East and Rowe considered the use of Kauffman's Boolean networks in the developmental mechanisms of a system that evolved problem-solving neural networks [2]. Gershenson described methods for guiding the self-organization of random Boolean networks to different regimes of network dynamics [4]. Although this and other related work has revealed and to some extent exploited interesting features of Boolean networks, it does not appear that work on these approaches has yet provided a robust technology for finding Boolean networks that compute specified functions.

Separately, work on evolvable hardware and Cartesian genetic programming has produced methods that can find Boolean circuits that compute specified functions [12, 13, 19, 18, 5, 16]. Most of this work has focused on finding combinatorial circuits, which cannot have cycles and hence cannot exhibit the attractor dynamics that has driven interest in Boolean networks. Prior work on finding sequential circuits, which may have cycles, has focused on the evolution of circuits with highly constrained architectures for sequential processing [1, 15]. Little work appears to have been done so far on the adaptation of unconstrained Boolean networks to compute specified functions that map Boolean inputs to Boolean outputs.

The present study aims to take initial steps in the filling of this gap, by exploring the search for unconstrained Boolean networks that compute functions of interest. I assume a problem environment in which an input consists of some fixed number of Boolean values, and in which an output consists of some possibly different fixed number of Boolean values.

Because the goal at this stage is to uncover basic principles of search for networks that implement target functions, choices of representations and algorithms have been made with primary attention to considerations of simplicity, generality and uniformity. One choice reflecting this priority is that I consider here only networks made of one kind of gate, a "programmable" gate that can act as any 2-input Boolean gate. This is in contrast to prior work that considered heterogeneous networks composed of gates chosen from a specified set such as {`AND, OR, NOT`}. Another choice reflecting my focus on basic principles is that I consider only search methods that sequentially modify a single network until it meets the target specification. Relative to approaches based on more sophisticated methods such as genetic algorithms, this avoids complexities such as those introduced by interactions within populations. It is possible that future work will determine that the most effective search algorithms take advantage of such complexities, but because my current goal is to understand fundamental features of the search space I aim for simplicity, generality, and uniformity when possible here.

In the following sections I describe the programmable gate out of which the networks that I will consider are constructed, preliminary observations about the behavior of self-programming Boolean networks, and the ways in which networks of these gates can be used to compute functions from inputs to outputs. I then describe the search method that I use to find networks that satisfy particular input/output specifications, and I present results of preliminary experiments using this search algorithm. I conclude with some general observations and suggestions for future work.

1.2 Programmable Gates

Prior work on random Boolean networks showed that a variety of network properties depend in critical ways on the subset of possible gate types that one allows to be used in the networks. In the interest of simplicity, generality, and uniformity I have chosen here to use only a single type of gate that can be configured to act as any 2-input Boolean gate.

The function of a 2-input Boolean gate can be specified using a truth table, the right-most column of which indicates the output for each possible combination of input values. For example, the right-most column of the truth table shown in Table 1.1 specifies the behavior of an `AND` gate.

Table 1.1 The truth table an `AND` gate, shown using 0 for `false` and 1 for `true`.

A	B	OUT
0	0	0
0	1	0
1	0	0
1	1	1

A programmable 2-input gate takes four additional inputs, the values of which specify the right-most column of the truth table for the Boolean function that the gate will implement. For example, if the values of the four additional inputs are `false`, `false`, `false`, and `true`, then the gate will act as an `AND` gate of the original two inputs. As will be useful for execution traces presented below, the right-most column of the truth table can be notated with a single hexadecimal digit by first considering it to be a binary number with the most significant bit on top. Using this convention, the `AND` gate is notated as `1`; the hexadecimal notation for all 2-input Boolean gates is shown in Table 1.2.

Table 1.2 Hexadecimal notation for all 2-input Boolean gates. The first four columns show the right-most column of the gate's truth table, using 0 for `false` and 1 for `true`. The column header "FF" indicates that the value provides the output of the gate when the `A` and `B` inputs are both `true`, while the "FT" header means that the value provides the output when the `A` input is `false` and the `B` input is `true`, and so on. The "Hex" column shows the hexadecimal notation for the gate with that truth table. The "Gate" column provides a common description for that gate, assuming that the first input is `A` and the second input is `B`.

FF	FT	TF	TT	Hex	Gate
0	0	0	0	**0**	FALSE
0	0	0	1	**1**	AND
0	0	1	0	**2**	$A \wedge \neg B$
0	0	1	1	**3**	A
0	1	0	0	**4**	$B \wedge \neg A$
0	1	0	1	**5**	B
0	1	1	0	**6**	XOR
0	1	1	1	**7**	OR
1	0	0	0	**8**	NOR
1	0	0	1	**9**	=
1	0	1	0	**A**	$\neg B$
1	0	1	1	**B**	$B \implies A$
1	1	0	0	**C**	$\neg A$
1	1	0	1	**D**	$A \implies B$
1	1	1	0	**E**	NAND
1	1	1	1	**F**	TRUE

The truth table for the programmable gate is shown in Table 1.3.

The programmable gate is equivalent to the well known 4-to-1 multiplexer [20], which also takes a total of six inputs and is generally discussed as using two of the inputs to specify which of the other four inputs will appear at the output.[1] This turns out to be equivalent to using the latter four inputs to specify the truth table of the Boolean function applied to the original two inputs. Figure 1.1 shows the standard graphical representation for a 4-to-1 multiplexer, while Figure 1.2 shows my preferred graphical representation for the programmable gate. The gates are identical in function, but I prefer the "programmable" terminology and the representation in Figure 1.2 because they remind us that this gate can act as any 2-input Boolean gate, with the choice of gate being specified by additional inputs.

[1] Thanks to Bill Tozier for pointing this out. Ohers have studied the evolution of circuits composed of multiplexers (e.g. [11]), but not, so far as I know, for the production of unconstrained Boolean networks.

Table 1.3 The truth table for the programmable 2-input Boolean gate, which is equivalent to that for a 4-to-1 multiplexer, shown using 0 for `false` and 1 for `true`. The column header “FF” indicates that the value provides the output of the gate when the `A` and `B` inputs are both `true`, while the “FT” header means that the value provides the output when the `A` input is `false` and the `B` input is `true`, and so on.

FF	FT	TF	TT	A	B	OUT
0	0	0	0	0	0	0
0	0	0	0	0	1	0
0	0	0	0	1	0	0
0	0	0	0	1	1	0
0	0	0	1	0	0	0
0	0	0	1	0	1	0
0	0	0	1	1	0	0
0	0	0	1	1	1	1
0	0	1	0	0	0	0
0	0	1	0	0	1	0
0	0	1	0	1	0	1
0	0	1	0	1	1	0
0	0	1	1	0	0	0
0	0	1	1	0	1	0
0	0	1	1	1	0	1
0	0	1	1	1	1	1
0	1	0	0	0	0	0
0	1	0	0	0	1	1
0	1	0	0	1	0	0
0	1	0	0	1	1	0
0	1	0	1	0	0	0
0	1	0	1	0	1	1
0	1	0	1	1	0	0
0	1	0	1	1	1	1
0	1	1	0	0	0	0
0	1	1	0	0	1	1
0	1	1	0	1	0	1
0	1	1	0	1	1	0
0	1	1	1	0	0	0
0	1	1	1	0	1	1
0	1	1	1	1	0	1
0	1	1	1	1	1	1
1	0	0	0	0	0	1
1	0	0	0	0	1	0
1	0	0	0	1	0	0
1	0	0	0	1	1	0
1	0	0	1	0	0	1
1	0	0	1	0	1	0
1	0	0	1	1	0	0
1	0	0	1	1	1	1
1	0	1	0	0	0	1
1	0	1	0	0	1	0
1	0	1	0	1	0	1
1	0	1	0	1	1	0
1	0	1	1	0	0	1
1	0	1	1	0	1	0
1	0	1	1	1	0	1
1	0	1	1	1	1	1
1	1	0	0	0	0	1
1	1	0	0	0	1	1
1	1	0	0	1	0	0
1	1	0	0	1	1	0
1	1	0	1	0	0	1
1	1	0	1	0	1	1
1	1	0	1	1	0	0
1	1	0	1	1	1	1
1	1	1	0	0	0	1
1	1	1	0	0	1	1
1	1	1	0	1	0	1
1	1	1	0	1	1	0
1	1	1	1	0	0	1
1	1	1	1	0	1	1
1	1	1	1	1	0	1
1	1	1	1	1	1	1

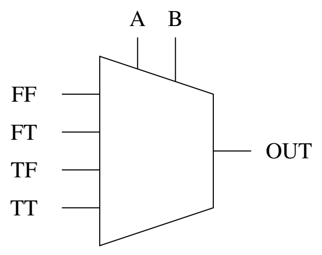

Fig. 1.1 A 4-to-1 multiplexer with inputs FF, FT, TF and TT and switching bits A and B.

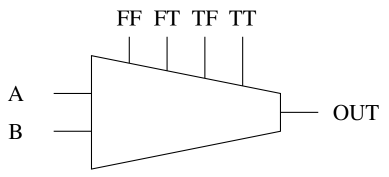

Fig. 1.2 A programmable 2-input Boolean gate with inputs A and B and truth table defined by FF, FT, TF and TT.

1.3 Self-Programming Networks

A self-programming Boolean network of a specified size can be created by creating the specified number of programmable gates and then wiring each input of each gate to the output of a randomly chosen gate. Although various network update schemes might be considered, I consider here only synchronous updates in which all gates simultaneously determine their states for step $s + 1$ by using the values of the state at step s as inputs. Note that this means that the execution of a network is entirely deterministic. While I am looking here at randomly wired networks, and I will later consider random processes for finding networks that behave in certain ways, there is nothing random in the execution of a network once it has been created.

One set of questions that immediately arises concerns the dynamics of random self-programming Boolean networks. What will they generally be like? Although studies have been conducted of the dynamics of networks of various types of Boolean gates, I am not aware of studies of networks of programmable gates in particular.

As a preliminary step in the study of these networks I generated $10,000$ networks of each size from 1 to 100 gates and ran each from a random initial state until a state was repeated, meaning that the network had entered a cycle. I then reported the length of the cycle. Figure 1.3 shows the mean cycle length for each number of gates, while Figure 1.4 shows the maximum observed cycle length for each number of gates.

From results reported by Kauffman one might expect the dynamics of networks of 6-input gates with true/false-balanced outputs to be quite chaotic, since he reports a tendency for chaotic dynamics in networks in which gates have more than two inputs ($K > 2$, in his terminology) and also in which gates have balanced outputs ($P = 0.5$ in his terminology). Nonetheless, the behavior of the specific gate used here appears to be more manageable, with mean cycle lengths growing modestly with the number of gates. Even for networks of 100 gates, for which the size of the state space is $2^{100} > 10^{30}$, one can expect convergence to an attractor cycle—a sequence of network states that henceforth repeats forever—of fewer than six steps. Maximum cycle lengths are subject to outliers in the thousands, but even so their growth is modest in comparison to state space size. For all network sizes the median cycle length was 1.

Although the dynamics seen here are encouraging, they reflect the behavior of randomly initialized networks. When we use networks to compute functions from inputs to outputs initial states will presumably be determined by inputs, and may be constrained. This may change the range of dynamics that we can expect to observe.

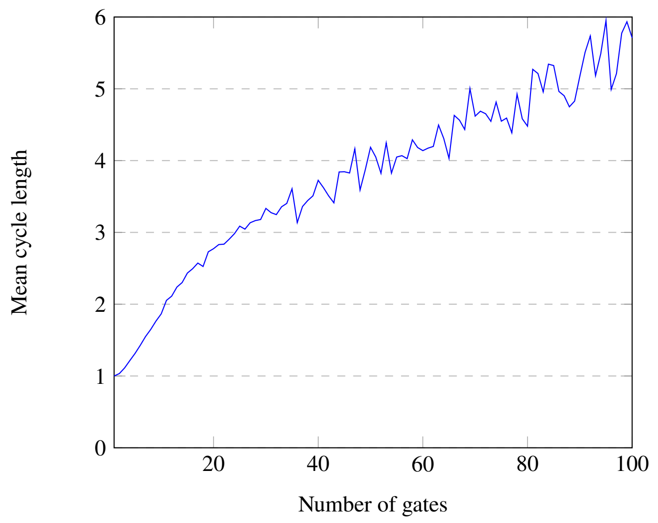

Fig. 1.3 Mean cycle lengths for random networks of programmable gates. For each plotted point 10,000 random networks of the specified size were created, initialized randomly, and run until a state was repeated.

In order to use a self-programming Boolean network as a function that maps inputs to outputs the model is augmented with additional input sources and a convention for extracting outputs. Specifically, when working on a problem involving i Boolean inputs, $i + 2$ sources are added for wires to gate inputs beyond those provided by the outputs of the gates in the network. i of the added sources are set to provide the values of the corresponding inputs, held constant throughout the process of determining the network's outputs for those inputs. The other two added sources provide constant values of `true` and `false`. Note that this means that it is possible to wire a self-programming Boolean network to act as a network of standard 2-input gates, by wiring all of the programming inputs of all of the gates to constants. On the other hand, if the programming inputs of gates are wired to the outputs of other gates then more complex behaviors may result.

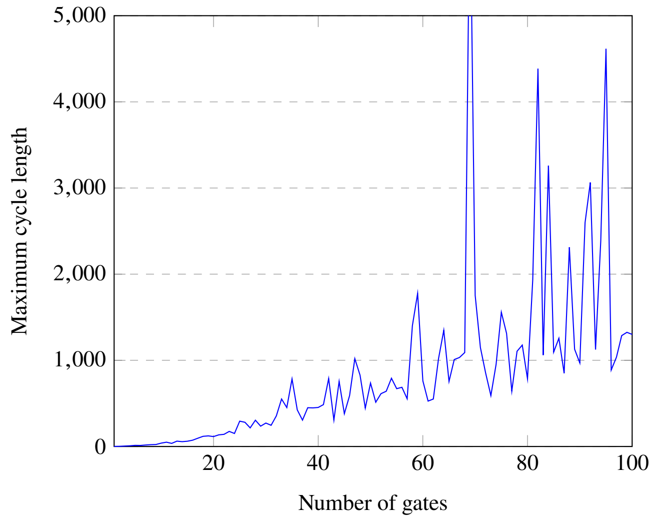

Fig. 1.4 Maximum cycle lengths for random networks of programmable gates. For each plotted point 10,000 random networks of the specified size were created, initialized randomly, and run until a state was repeated.

Outputs are taken from pre-specified gates in the network, so a network for a problem involving *u* output bits must include at least *u* gates that are pre-specified to serve as outputs. I refer to gates that are not designated as outputs as “hidden” gates, in analogy to the use of “hidden layers” to describe layers of nodes in neural networks that are neither inputs nor outputs.

To determine the outputs that a network computes from a specific set of inputs one initializes the state of all gates to `false` and then iteratively and synchronously updates the state of all gates. Execution terminates when a state is repeated. If the values of output gates have remained unchanged throughout the attractor cycle into which the network has fallen then those values are returned as the output from the computation. If they have changed over the course of the cycle then the output of the computation is considered to be undefined and no value is returned.

1.4 Ratcheted Random Search

With the aim of advancing our understanding of the space of self-programming Boolean networks, I have chosen to search for networks that solve specified problems using extremely simple search methods.

Training cases are decomposed so that each case specifies the correct value for a single Boolean output. This means that if a problem involves *u* outputs then there will be *u* training cases for each collection of input values, each specifying the correct value of one of the outputs. For example, for the problem of finding a self-programming Boolean network that acts as a 2-bit $\times$ 2-bit multiplier, which must take four Boolean inputs and produce four Boolean outputs, there will be four training cases for each product. So for $2 \times 3 = 6$, which in binary is $10 \times 11 = 0110$, there will be a case specifying that the leftmost output should be 0 (`false`), another specifying that the next output should be 1 (`true`), and so on.

All of the methods discussed here maintain an initially empty collection of solved training cases and an initially complete collection of unsolved cases. At each step the current network (N) is tested on a randomly chosen unsolved case (C). If N produces the correct output for case C then C is moved from the unsolved collection to the solved collection. On the other hand, if it produces the incorrect output then a random change is made to N, producing N'. N' is then tested both on C and on all of the previously solved cases. If it produces the incorrect output for any of the previously solved cases then the change is reverted. That is, only steps in the search space that maintain correctness for all previously-solved cases are allowed. This implements a "ratchet-like" mechanism that only permits changes that are neutral or better with respect to solved cases. If N' does still produce the correct outputs for all previously solved cases, and if it also produces the correct output for C, then the change is retained (N' is used as the next N) and C is moved from the unsolved to the solved collection. If N' produces the correct outputs for all previously solved cases but not for C then the question of whether to retain N' or revert to N is more complicated, as described below.

Although one might think at first that our quest for simplicity would best be served by a search method that is somehow "purely" random, it is not clear what that could or should mean in the context of the fact that there is no limit on network size. Most networks sampled from a purely-random distribution will be bigger than our biggest computers, even though many problems that we care about can probably be solved with rather small networks. Because of this, and because we generally won't know in advance how many gates are required to implement a target function, the search methods considered here all begin with a network that contains no hidden gates and allow changes to the number of gates to be among the random changes made during search.

Three kinds of network changes are permitted in the ratcheted random search algorithms presented here:

- Rewire: Change the source for a single input of a single gate, arbitrarily or locally (see below).
- Add: Add a new gate with all of its inputs set to random sources, and change one input of one pre-existing gate to use the output of the new gate as its source.
- Delete: Delete a randomly chosen gate and randomly choose another single source to replace all references to the deleted gate's output.

If these kinds of changes are equally likely, and if all changes that are neutral with respect to solved cases are retained, then networks tend to grow rapidly and without bound at least for the problems studied here. Experiments have therefore been conducted with three settings that help to control network growth. In one ("add rarely”), the probability for choosing each kind of change is altered, typically by setting the probability of choosing Add to $\frac{1}{100}$ the probability of choosing Rewire or Delete. In the second (“add for improvement”) and third (“add for disruption”), the conditions under which we retain the results of an Add are changed. Specifically, with “add for improvement” the results of an Add are retained only if it correctly solves not only all previously solved cases but also the current case C. In the “add for disruption” setting. although the results of the Add needn’t correctly solve C in order to be retained, the change must at least influence gate states during the processing of C. These variants are of course not the only possible mechanisms for controlling growth, but they are straightforward and not obviously biased. The first, changing the probability for additions relative to other network modifications, seems to control network growth reasonably well while still allowing solutions to be found in many settings. However, the introduction of probability parameters complicates analysis. The second approach, requiring the results of additions to solve C, has produced among the most concise solutions in some preliminary experiments but appeared to slow search and perhaps prevent the algorithm from ever finding solutions in some settings. The third approach, requiring that additions at least disrupt the processing of C, seems at the time of this writing to be the most generally promising of the variants considered here. Although further testing of these and other variants is warranted, we present here only results of the third approach.

Although the approaches described above help to focus search on reasonably small networks, even modestly-sized networks have astronomically large numbers of rewiring options. Consider a problem with i inputs, so that a network with g gates would have $i + g + 2$ possible sources for each of $6g$ destinations. The number of possible wirings for this network is $(i + g + 2)^{6g}$, which is over 10^{73} for $i = 5$ and $g = 10$, and over 10^{180} for $i = 10$ and $g = 20$.

Based on considerations of the vastness of this space, and on observations of modular structure in complex systems, I have also experimented with two variants of Rewire changes. In one (“arbitrary rewiring”), the new source for the chosen input is picked from all possible sources, with each having equal probability. In the second (“local rewiring”), the source is chosen with uniform probability from a collection that contains one arbitrary source but also, if the current source is a gate rather than a network input or constant, all sources for all inputs to that gate. Because it has performed best in our preliminary investigations, I present here only results of runs with local rewiring.

I make no claim that the search methods used here will be the most effective for finding Boolean networks that solve specified problems, only that they are relatively simple and that, if they prove to be capable of finding solutions at all, then their simplicity may help us to understand the nature of the search space. For this reason I have tried to rely mostly on uniform random variation and the “ratchet” mechanism described above, making strategic modifications to this minimalist approach only to mitigate the effects of network growth, specifically by allowing additions only if they disrupt network dynamics and by preferring a kind of local rewiring. It may well be that more effective methods will utilize more sophisticated techniques, possibly based on traditional methods for digital circuit design or on work on evolvable hardware. For the present, however, I am aiming to deepen our understanding of the search space by relying on ratcheted random search as much as possible.

1.5 Results

To assess the scaling behavior of our search algorithms, experiments were conducted on search for even parity functions and multipliers, both of which have previously been studied as targets for the learning or evolution of combinational circuits.

For even parity, I searched 100 times for solutions to each of the even-2-parity, even-3-parity, even-4-parity and even-5-parity problems. All searches found solutions, requiring the numbers of search steps shown in figure 1.5. Figure 1.6 shows the numbers of hidden gates in solution networks in those same runs.

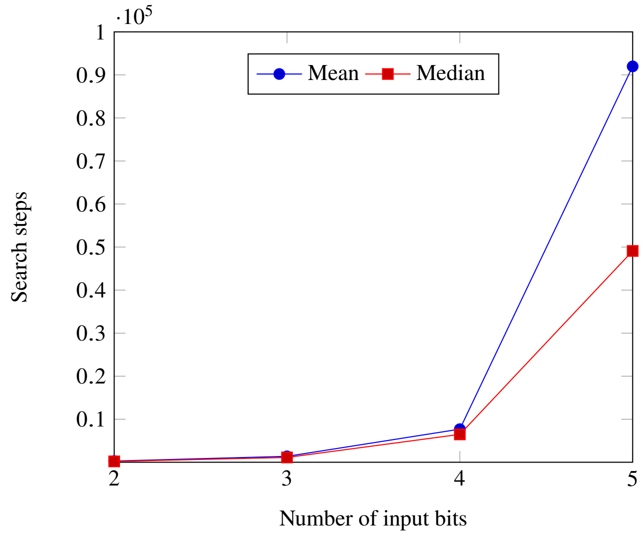

Fig. 1.5 Ratcheted random search steps required to find solutions to parity problems of the specified sizes, aggregated over 100 runs of each problem size.

Note that the number of search steps required to find a solution grows non-linearly with problem size. This is not surprising given the exponential growth in the state space as networks grow, but it does appear to show that more refined search methods will be required to solve larger problems and it provides a benchmark that can be used in experiments with such methods. The growth in the number of hidden gates in solutions, however, appears to be more modest and approximately linear.

The search for multipliers is more challenging than the search for parity functions, in part because each computation produces multiple output bits. The algorithm described here can reliably find 3-bit $\times$ 3-bit multipliers, which have six input bits

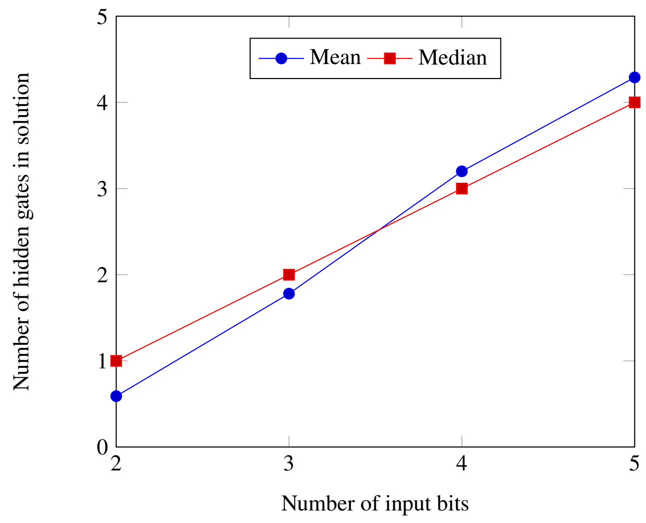

Fig. 1.6 Numbers of hidden gates in solutions to parity problems of the specified sizes, aggregated over the same 100 runs of each problem size used for Figure 1.5.

and six output bits, but these searches take long enough that statistics over large numbers of searches have not been gathered. That said, 3-bit $\times$ 3-bit multipliers found by the algorithm can be interesting in structure and behavior, and one will be presented below.

For 2-bit $\times$ 2-bit multipliers, with four input bits and four output bits, I conducted 100 searches with the same algorithm as used above. All searches succeeded, requiring numbers of search steps with a mean of $16,952.51$ and a median of $13,169.5$. This means that for our algorithm finding a 2-bit $\times$ 2-bit multiplier requires roughly twice as many search steps as needed to find an even-4-parity function, but only roughly a quarter as many as needed to find an even-5-parity function. The mean number of hidden gates was 2.34 and the median was 2. These numbers are lower than those for even-4-parity, but the 2-bit $\times$ 2-bit multipliers include 4 output gates, in comparison to the 1 output gate for even-4-parity, so the total number of gates in 2-bit $\times$ 2-bit multipliers is higher.

I will use a 2-bit $\times$ 2-bit multiplier found by our algorithm to demonstrate some of the features of self-programming Boolean networks. A diagram of the multiplier that I will consider is shown in Figure 1.7. Unfortunately, because of the large number of wires and the irregular connection pattern, it is difficult to glean much about how a network functions from diagrams such as this one, even for such a small network. That said, it is clear from Figure 1.7 that this multiplier makes no use of the constant `true` source and that all but one of its gates have self-loops, with the gate itself serving as one of its sources. One can also get a sense of the gate efficiency of the network by looking at it in comparison to diagrams of traditional combinational 2-bit $\times$ 2-bit multipliers that use 8 gates.

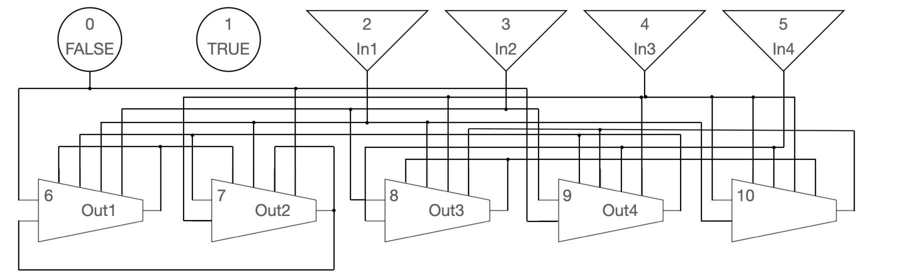

Fig. 1.7 A circuit diagram for a 2-bit × 2-bit multiplier found by the search algorithm presented here.

I have found it more useful to look at networks in tabular form in conjunction with execution traces for particular input patterns. Figure 1.8 shows such a trace for the multiplier in Figure 1.7 when multiplying 3 × 3.

The first line of the trace shows the indices for all of the network sources. The first two sources (with indices 0 and 1) are the constant `false` and `true` sources, respectively. The next four (2, 3, 4, and 5), following an empty column, are inputs; for this 2-bit × 2-bit multiplier the first number to be multiplied will be encoded in the two values appearing at indices 2 and 3, while the second number to be multiplied will be encoded in the values appearing at indices 4 and 5. The remaining indices all refer to programmable gates, with the output gates appearing first and then, after another empty column, any hidden gates. For all programmable gates the following six rows show the sources for the gates' inputs. The labels printed at the left of these rows show which input is which, with `A` and `B` indicating the sources for the inputs to the gate that will be programmed by the values at indices `FF`, `FT`, `TF`, and `TT`.

For example, consider the gate with index 6 in Figure 1.8. The right-most column of the truth table for this gate will be whatever values are present at indices 6, 9, 2, and 3. If we suppose that there are `false` values at indices 6, 9, and 2, and a `true` value at index 3, then this will act as an `AND` gate on the values at indices 0 and 7. Incidentally, in this case that would mean that we already know that the gate's output will be `false`, because the 0 index always refers to the constant `false`, because a single `false` input is enough to guarantee a `false` output from an `AND` gate.

Starting on the eighth row of the trace in Figure 1.8, following the rows for indices and sources of gate inputs, are rows showing the states of the network's sources (represented with `-` for false and `*` for true for the sake of legibility) and the hexadecimal representations of the logic functions that each gate is programmed by that state to perform. The first of these lines shows the network's initial state, in which the constant sources are `-` and `*` (as always), the four input values are all `*` (because both inputs are 3, which in binary is 11), and all other gates are initialized to `false`. The next line shows the programming of the gates as specified by the initial state. Specifically, the five gates are programmed to be hexadecimal `3`, `3`, `6`, `3`, and `E`, which by consulting Table 1.2 we can see are `A`, $B \wedge \neg A$, `XOR`, `A`, and `NAND`. The following row shows the state that results from the processing of these gates, in which we can see that the gates at indices 7 and 9 have flipped to `true`. Skipping to the bottom of the diagram we can see that the last two states are identical, which is why the computation terminated—in this case with a cycle length of 1, although that will not always be the case. We can also see that the values of the output gates in the final state are `true`, `false`, `false`, and `true`, which encode 1001 which is binary for 9, which is the correct answer for 3×3. We can also see that along the way to producing this result the state at some indices changed more than once, and several gates were repeatedly reprogrammed.

```
0  1    2  3  4  5     6  7  8  9     10
                    A  0  9  3  3     4
                    B  7  4  5  0     2
                   FF  6  6  8  9     4
                   FT  9  2  2  10    5
                   TF  2  7  4  5     4
                   TT  3  0  10 4     8
_  *    *  *  *  *     _  _  _  _     _
                       3  4  6  3     E
_  *    *  *  *  *     _  *  _  *     _
                       7  6  6  B     E
_  *    *  *  *  *     *  _  _  *     _
                       F  C  6  B     E
_  *    *  *  *  *     *  _  _  *     _
```

Fig. 1.8 A trace of the multiplier shown in Figure 1.7 multiplying 3×3. See text for description.

Figure 1.9 shows a trace of the same multiplier multiplying 1×0. Here we can see that the initial gates immediately reproduce the initial state, causing immediate termination with the correct answer of 0. This illustrates the fact that a network may take different number of steps to produce a result for different inputs.

```
0  1    2  3  4  5     6  7  8  9     10
                    A  0  9  3  3     4
                    B  7  4  5  0     2
                   FF  6  6  8  9     4
                   FT  9  2  2  10    5
                   TF  2  7  4  5     4
                   TT  3  0  10 4     8
_  *    _  *  _  _     _  _  _  _     _
                       1  0  0  0     0
_  *    _  *  _  _     _  _  _  _     _
```

Fig. 1.9 A trace of the multiplier shown in Figure 1.7 multiplying 1×0. See text for description.

Figure 1.10 shows a trace of a network that our algorithm found for a 3-bit $\times$ 3-bit multiplier, shown multiplying 7×5. Again we can see that the network is reasonably space efficient, using 17 programmable gates in comparison to the 40 gates used in a traditional combinational 3-bit $\times$ 3-bit multiplier. That said, networks that use as few as 15 programmable gates have also been found. We can also see again a variety of interesting structures, including self loops, and the reprogramming of gates over the course of the computation.

```
0  1     2  3  4  5  6  7     8  9  10 11 12 13    14 15 16 17 18 19 20 21 22 23 24
                            A 23 18 2  13 6  16    7  5  0  15 3  22 1  4  21 1  19
                            B 5  5  12 23 18 2     5  13 2  8  3  20 14 7  12 15 3
                           FF 23 23 15 14 12 0     7  15 4  6  0  6  17 12 2  0  2
                           FT 23 22 10 20 7  7     21 3  21 21 21 4  1  7  21 14 6
                           TF 5  14 17 14 4  7     2  3  18 5  17 13 6  1  22 3  7
                           TT 2  19 17 14 21 0     16 24 23 16 23 13 0  21 2  6  2
_  *     *  *  *  *  _  *     _  _  _  _  _  _     _  _  _  _  _  _  _  _  _  _  _
                              3  0  0  0  6  6     A  6  8  2  0  4  4  6  9  2  B
_  *     *  *  *  *  _  *     _  _  _  _  _  *     _  *  _  _  _  _  _  _  *  *  _
                              F  C  8  0  6  6     A  E  9  2  1  7  4  6  B  2  B
_  *     *  *  *  *  _  *     *  *  _  _  _  *     _  _  _  *  *  *  _  _  *  _  _
                              3  5  3  0  6  6     A  6  A  2  2  7  C  6  B  2  B
_  *     *  *  *  *  _  *     _  *  *  _  *  *     _  _  _  _  _  *  _  _  *  *  *
                              F  D  4  0  E  6     A  7  9  2  1  7  4  E  B  2  B
_  *     *  *  *  *  _  *     *  *  _  _  *  *     _  *  _  _  *  *  _  _  _  *  *
                              F  9  8  0  E  6     A  F  B  2  1  7  4  E  9  2  B
_  *     *  *  *  *  _  *     *  *  _  _  *  *     _  *  _  _  *  _  _  _  _  _  *
                              3  0  8  0  E  6     A  F  A  2  0  7  4  E  9  2  B
_  *     *  *  *  *  _  *     _  _  _  _  *  *     _  *  _  _  _  _  _  _  _  _  _
                              3  0  8  0  E  6     A  E  8  2  0  7  4  E  9  2  B
_  *     *  *  *  *  _  *     _  _  _  _  *  *     _  _  _  *  _  _  _  _  _  _  _
                              3  0  3  0  E  6     A  6  8  2  2  7  C  E  9  2  B
_  *     *  *  *  *  _  *     _  _  *  _  *  *     _  _  _  _  _  _  _  _  _  *  _
                              F  8  4  0  E  6     A  6  9  2  1  7  4  E  9  2  B
_  *     *  *  *  *  _  *     *  _  _  _  *  *     _  _  _  _  *  _  _  _  _  *  _
                              F  8  0  0  E  6     A  6  B  2  1  7  4  E  9  2  B
_  *     *  *  *  *  _  *     *  _  _  _  *  *     _  _  _  _  *  _  _  _  _  *  _
```

Fig. 1.10 A trace of the 3-bit × 3-bit multiplier found by our algorithm, shown multiplying 7 × 5 to produce 35.

1.6 Conclusions and Next Steps

The material presented here provides conventions and baselines for studying the use of self-programming Boolean networks as a substrate for computational systems that adapt to solve specified problems. That said, it can be seen to raise more questions than it answers, and to provide several avenues for further research.

What has been demonstrated so far is that simple random search methods can indeed find self-programming Boolean networks that solve specified problems. The problems studied here, while modest in difficulty, do nonetheless involve astronomically large search spaces. The methods employed combine random walks with a ratchet-like mechanism that prevents the loss of function during search. The problem-solving networks that have been found are often parsimonious in their use of gates and employ feedback and context-dependent execution dynamics in novel and interesting ways.

Many open questions were raised in the development of this work, and more were raised in discussions at the 2024 Genetic Programming Theory and Practice workshop.[2] Some of these concern the execution mechanics of individual networks; for example, one can ask whether it would be helpful to initialize networks differently, to terminate execution under different conditions, or to extract outputs even when the output gates are not all stable over the attractor cycle. Others concern the distribution of random moves that are considered during search, asking for example whether search will be more efficient if different probabilities of change are applied to data inputs (`A` and `B`) than to programming inputs (`FF`, `FT`, `TF`, and `TT`). A variety of other analyses of the experimental data may also be revealing, ranging from analysis of variance of the measures that have already been presented to the calculation of measures of chaotic dynamics. Additional control experiments might also be conducted, for example with the ratchet mechanism disabled or with programming inputs wired only to constants. The effects of other constraints on wiring patterns, such as restriction to a 3D lattice, might also be considered. Comparisons and/or synergies with other work on search for executable networks might be explored, for example with recurrent Cartesian genetic programming [17] or Markov Brains [6]. Finally, it would be interesting to examine this work in the context of studies of other computational substrates that involve complex underlying dynamics, for example with physical reservoir computing [14].

Acknowledgements Many thanks to Bill Tozier, Mahran Yousef, members of the PUSH lab at Amherst College and the University of Massachusetts, Amherst. Thanks also to participants in the 2024 Genetic Programming Theory and Practice workshop, especially including Wolfgang Banzhaf, Alex Lalejini, Penousal Machado, Charles Ofria, Bill Tozier, and Leonardo Vanneschi. This work was performed in part using high performance computing equipment obtained under National Science Foundation Grant No. 2117377. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Competing Interests The author has no conflicts of interest to declare that are relevant to the content of this chapter.

References

1. Ali B, Almaini AEA, Kalganova T (2004) Evolutionary Algorithms and Theirs Use in the Design of Sequential Logic Circuits. Genetic Programming and Evolvable Machines 5:11–29.
2. East R, Rowe J (1997) Abstract Genetic Representation of Dynamical Neural Networks Using Kauffman Networks. Artificial Life 3(2): 67–80.
3. Gershenson C (2004) Introduction to Random Boolean Networks. Available via `https://api.semanticscholar.org/CorpusID:5278615`
4. Gershenson C (2012) Guiding the self-organization of random Boolean networks. Theory Biosci. 131(3):181–191.
5. Haddow PC, Tyrrell AM (2018) Evolvable Hardware Challenges: Past, Present and the Path to a Promising Future. In: Stepney S, Adamatzky A (eds) Inspired by Nature. Emergence, Complexity and Computation, vol 28. Springer, Cham.

[2] Thanks especially to Wolfgang Banzhaf, Alex Lalejini, Penousal Machado, Charles Ofria, Bill Tozier, and Leonardo Vanneschi for suggestions related to the questions raised in this paragraph.

6. Hintze A, Edlund JA, Olson RS, Knoester DB, Schossau J, Albantakis L, Tehrani-Saleh A, Kvam P, Sheneman L, Goldsby H, Bohm C, Adami C (2017) Markov Brains: A Technical Introduction. Available via https://doi.org/10.48550/arXiv.1709.05601
7. Kauffman S (1969) Homeostasis and Differentiation in Random Genetic Control Networks. Nature 224:177—178.
8. Kauffman S (1993) The Origins of Order: Self-Organization and Selection in Evolution. Oxford University Press, New York.
9. Kauffman S (1995) At home in the universe. Oxford University Press, New York.
10. Lemke N, Mombach JCM, Bodmann BEJ (2021) A numerical investigation of adaptation in populations of random boolean networks. Physica A: Statistical Mechanics and its Applications 301:589—600.
11. Mazare A, Ionescu L, Serban G, Barbu V. (2011) Evolvable hardware with Boolean functions network implementation. In 2011 International Conference on Applied Electronics Applied Electronics (AE), IEEE.
12. Miller J (ed) (2011) Cartesian genetic programming. Springer Berlin, Heidelberg.
13. Miller, JF (2020) Cartesian genetic programming: its status and future. Genet Program Evolvable Mach 21, 129–168.
14. Nakajima K (2020) Physical reservoir computing — an introductory perspective. Japanese Journal of Applied Physics, Volume 59, Number 6.
15. Tetteh M, Dias DM, Ryan C. (2022) Grammatical Evolution of Complex Digital Circuits in SystemVerilog. SN Comput Sci.(3):188.
16. Shanthi AP, Parthasarathi R (2009) Practical and scalable evolution of digital circuits. Applied Soft Computing 9:618–624.
17. Turner AJ, Miller JF (2014) Recurrent Cartesian Genetic Programming. In: Bartz-Beielstein T, Branke J, Filipič B, Smith J. (eds) Parallel Problem Solving from Nature – PPSN XIII. PPSN 2014. Lecture Notes in Computer Science, vol 8672. Springer, Cham.
18. Vasicek Z (2018) Bridging the Gap Between Evolvable Hardware and Industry Using Cartesian Genetic Programming. In: Stepney S, Adamatzky A (eds) Inspired by Nature. Emergence, Complexity and Computation, vol 28. Springer, Cham.
19. Walker JA, Miller JF (2005) Improving the Evolvability of Digital Multipliers Using Embedded Cartesian Genetic Programming and Product Reduction. In: Moreno JM, Madrenas J, Cosp J (eds) Evolvable Systems: From Biology to Hardware. ICES 2005. Lecture Notes in Computer Science, vol 3637. Springer, Berlin, Heidelberg.
20. Wikipedia (2024) Multiplexer. Available via https://en.wikipedia.org/wiki/Multiplexer#Digital_multiplexers
21. Wuensche A (1994) The Ghost in the Machine: Basins of Attraction of Random Boolean Networks. In Artificial Life III Proceedings, Santa Fe Institute Studies in the Sciences of Complexity.