

TWO REVERSE ENGINEERING ALGORITHMS FOR BOOLEAN GENETIC NETWORKS

(last revised 8/21/99)

JOHN E. MYERS

Oceanside, Oregon

jemyers@oregoncoast.com

We introduce two methods for identifying the network architecture from a state transition table (i.e., gene expression matrix). The *lattice algorithm* is guaranteed to find the unique effective architecture corresponding to any complete transition matrix, without any constraint as to connectance or rules. The method can be used to experiment with model networks when nothing is known or assumed about the network other than its dynamical features. *The mismatched pairs algorithm* predicts the effective network architecture given a sample of state transitions and, hence, may be useful in reverse engineering with real biological data sets. Simulations applying the mismatched pairs algorithm to completely random networks with over 1,500 elements and 5 inputs per element are tractable even on a 200 MHz PC. Much larger nets would be feasible with greater computing power and parallel processing. Importantly, the algorithms do not rely on exhaustive (trial and error) search among candidate architectures and, hence, inherently should be more efficient than other published reverse engineering methods for Boolean network models.

1 Introduction

With the development of methods for measuring gene expression time series on a massively parallel scale, interest in “reverse engineering” the underlying genetic network architecture has grown apace.^{6, 7, 8} In the standard Boolean network model, the genes (or, more generally, the biological variables) are represented as binary elements that can be either “on” or “off.” The network *wiring* links genes which have a direct functional relationship. Each gene’s *rule* specifies whether the gene state should be on or off at the next update, depending on the current states of the elements in its wiring pattern. The wiring and rules for all genes comprise the network *architecture*. The state of each element defines the network state at any time. The system is dynamic: gene states update synchronously at discrete time intervals until reaching a repeating pattern or sequence of patterns known as the *attractor*.^{2, 7, 9}

We seek an analytical model of genetic regulation in living cells that can predict the measured time sequence of cell states, including the time sequence that follows experimental intervention. As one key approach, we hope to reverse engineer genetic network architecture from actual experimental time series data. Reverse engineering

essentially amounts to solving the "inverse problem", i.e., finding the wiring and rules that produce the observed dynamical pattern. We seek the most compact (i.e., least-connected) architecture if more than one is possible.

Several reverse engineering algorithms have been developed based on Kauffman's Boolean model.^{1, 2, 3, 4, 5} Three existing alternative algorithms share a common trait: all employ exhaustive trial and error search among the network architecture candidates for each gene until one that "fits the data" is identified.^{1, 3, 5} Since the number of potential architectures is exponential in both the number of elements in the net and the number of effective inputs per element, these methods may be intractable except with respect to small sparsely-connected model networks.

We believe the algorithms presented here are much more efficient than the present alternatives because they directly identify the architecture for each gene, rather than searching exhaustively across a range of network wiring or wiring/rule configurations.

2 The Lattice Algorithm

2.1 Description

Consider a Boolean network comprised of N elements (genes) and having K effective inputs or wires per gene.^a Suppose we have a complete transition table, that is, we know the network state at time $t + 1$ for all 2^N possible states at time t . We want to identify the effective network that exactly yields the dynamics shown in the transition table.

For ease of illustration, we consider a network with three genes labeled A, B, and C. Assuming no a priori constraints, each element can have any of the following eight (2^3) possible sets of effective inputs, shown in a lattice arranged from greatest to least connectance:

ABC
AB AC BC
A B C
NULL ^b

^a An *effective input* determines the element's output for at least one combination of the other $K - 1$ effective inputs.

^b NULL wiring means no effective inputs, i.e., fixed output.

The lattice algorithm unambiguously finds the unique^c effective wiring and rules by examining only N wiring possibilities for each element, regardless of K (even for $K = N$). The basic idea is as follows. Suppose that we want to find whether element C is an effective input to element B . We can do this by testing whether element B 's transitions are deterministic, i.e., single-valued for each input state configuration, assuming that C is NOT one of the effective inputs, that is, by testing AB wiring (see above lattice). This is a simple procedure because a complete transition table includes all pairs of input states that differ only in the state of element C (or any other element). If we find one instance in which B 's output differs between any two input states that are identical except for the value of C , then we know that C is an effective input to B (by definition).^d Else, C is not an effective input and is not part of element B 's effective architecture.

Excluding a wiring structure (i.e., set of inputs) at an upper level of the lattice permits eliminating all the component structures at lower levels. For example, suppose we find C to be an effective input by evaluating the transitions for AB ; i.e., at least one set of values for AB leads to different outputs for element B . Then we can also rule out A , B , and $NULL$ as effective wiring structures for element B , since none of them contain element C . In fact, given any complete transition table, one can eliminate all possibilities other than the effective architecture by examining N wiring sets per element, INDEPENDENT of K . The rules are a by-product of the wiring assessment and need not be tested separately. The power of the method is that it eliminates large numbers of wiring candidates by examining only a relative few, specifically, N out of 2^N possibilities.

2.2 Example

To illustrate the lattice algorithm, we consider the simple net used to demonstrate REVEAL. (Reference 3, Figure 1). The transition table is as follows:

^c Ignoring the ordering of inputs in the rule table. Reordering the inputs simply reorganizes the rule table and does not affect the network dynamics.

^d Since all elements other than C have the same value between the two input states, only element C can account for the differing outputs.

time t			time $t + 1$			line #
input			output			
A	B	C	A'	B'	C'	
0	0	0	0	0	0	1
0	0	1	0	1	0	2
0	1	0	1	0	0	3
0	1	1	1	1	1	4
1	0	0	0	1	0	5
1	0	1	0	1	1	6
1	1	0	1	1	1	7
1	1	1	1	1	1	8

Consider the A' column, which shows the time $t + 1$ state of element A for each possible network state at time t . Evaluate inputs in sequence C, B, A (arbitrary order).

1. Consider wiring candidate AB. (Envision by covering column C.) The result of excluding C is to pair input states which are identical when C is ignored, in this case, the states on lines 1 and 2, 3 and 4, 5 and 6, 7 and 8. If we find even one inconsistency in outputs for any pair, we know that C is an effective input to element A. Here, there are no inconsistencies: both instances of AB = 00 have the same output (0), both cases of AB = 01 have the same output (1), etc. Hence, C is not an effective input to A.

2. Consider AC. Covering the B column pairs input states on lines 1 and 3, 2 and 4, 5 and 7, 6 and 8. Here we find an inconsistency right away: on line 1, AC = 00, output = 0; on line 3, AC = 00, output = 1. Hence, B is an effective input to A'. That is, knowledge of element B's state is necessary to determine element A's output whenever both A and C are in state 0. Actually, that's true for all other states of A and C too but, since one inconsistency suffices, here we must examine only one of the four pairs.

3. Consider BC. Cover column A, thereby pairing states on lines 1 and 5, 2 and 6, 3 and 7, 4 and 8. Comparing outputs for each pair, we find no inconsistencies. Hence, A is not a required input for output A'.

Ergo, in the effective architecture for this transition table, B is the only effective input to element A. The rule is immediately evident, that is, $A' = B$. One can find the minimal wiring for elements B and C (i.e., output columns B' and C') in exactly the same way. NOTE: although elements B and C have more than one effective input³, finding their effective architectures using the lattice algorithm requires no more steps.

The lattice algorithm requires no floating-point calculations, such as the entropy and mutual information measures computed in REVEAL.³ Simple bit comparisons suffice.^e

In actual implementation, if removing an input leads to no inconsistencies, the transition table can be collapsed so that the program considers only half as many state

^e However, our program does use a simple integer formula to determine which states are paired.

pairings when testing the next input. Thus, the algorithm moves down as well as across the lattice, but still testing only N wiring arrangements. At the end, the remaining outputs give the rule directly.

2.3 Discussion

As presented, the complexity of the lattice algorithm is exponential in N because the transition table has 2^N entries, all of which are examined for any non-effective input. In practice, if $K \ll N$, the effective architecture can be identified to a high degree of precision by examining a relatively small sample of the state transitions, the sample being an exponential function of K but independent of N . As long as $K > 1$, the algorithm tests far fewer wiring configurations than other algorithms that test all possibilities with up to K inputs.^{1, 3, 5}

The lattice algorithm normally requires a complete transition table in order to access all input state pairs separated by a Hamming distance of one bit. We believe this requirement is an insurmountable obstacle to using the algorithm to reverse engineer real genetic networks. In practice, the entire transition table for biological networks is huge (2^N rows) and only a negligible portion of it can be measured. For example, on a PC with 128 MB of RAM, it is feasible to apply the lattice algorithm only to networks with up to 24 elements.^f The transition table more than doubles in size with each additional element so that even larger computers could handle only quite small networks. Memory, not computation time, is the limiting factor. For example, With our implementation in C, solution time for a 24-element net is under 7 minutes on a 200 MHz PC, essentially independent of K .^g

^f The output side of the transition table for a 24-element net is 49 MB of data (131 MB for a 25-element net!). The input states can be omitted from the transition table by using conventional ordering of the output states.

^g Nets with larger effective- K actually run a bit faster with the lattice algorithm, whereas in bottom-up exhaustive search methods the run times increase rapidly with effective- K .

We believe the lattice algorithm could be used to advantage in a kind of inverted ensemble study. One could generate a sample of transition graphs from assumptions with respect to attractor characteristics, say, and then employ the lattice algorithm to identify the kinds of architectural features that produce that dynamical behavior.^h Having done this on a small network, say one with 20-24 elements, one could apply the identified features to larger networks and run them forward to determine if the same sort of dynamics results. The potential value of this approach is that it may offer a way to identify architectural features corresponding to certain types of observed or expected dynamics, rather than having to, in large measure, guess at these features a priori.

As an aside, the lattice algorithm also can be used to find the effective- K of Boolean rules. On a PC with 128 MB of RAM, for example, this is feasible for rules with up to (at least) 29 nominal inputs.

3 Mismatched Pairs Algorithm

3.1 Description

We developed the mismatched pairs algorithm for reverse engineering from a sample of network input-output (I-O) state pairs, the sample representing only a very small part of the network transition table. The algorithm is quite unlike the lattice algorithm or any other Boolean network reverse engineering method of which we are aware. The mismatched pairs algorithm predicts effective architectures with a high degree of precision (in simulations) without directly testing ANY wiring or rule configurations.

Since the mismatched pairs algorithm predicts the effective architecture for each element (gene) independently, we describe how it works for an arbitrary element. Denote by N the number of elements in the net, by K the number of effective inputs, and by S the number of network I-O state pairs in the sample.

^h There is a one-to-one correspondence between a transition table and a transition graph in which the attractor characteristics and other dynamical features such as run-in lengths are more easily determined.

An element's outputs across the sample transitions are a mixture of 1's and 0's.ⁱ Consider a pair of input states where the element's outputs differ, i.e., the output is 1 for one input state and 0 for the other. We call this a "mismatched pair." The Hamming distance between any pair of input states ranges from 1 to N bits. The differing bit positions between members of each mismatched pair of input states must include at least one of the K effective inputs.^j Thus, the effective inputs tend to appear more often among the differing bits than do the other $N - K$ elements, especially if $N \gg K$.^k The smaller the Hamming distance between the members of a mismatched pair, the greater the relative expected frequencies of the effective inputs versus the non-effective inputs among the differing bit positions. These statistical tendencies are the basis for the mismatched pairs method.

For a given element's outputs, the set of differing bit positions for each mismatched pair of input states can be identified. The bit frequencies can be tabulated across all, or a subset, of mismatched pairs corresponding to the sample transitions. The key task is to find the smallest set of elements, at least one of which is present in the set of differing bit positions for each mismatched pair. This set comprises the predicted effective inputs. We have developed a straightforward procedure to accomplish the task, although there may be a better one. Our procedure is not bullet proof, but it finds the minimal wiring a very high percentage of the time -- depending on sample size, of course -- and seems reasonably efficient. We present test run statistics below.

The mismatched pairs algorithm is quite simple computationally, mainly involving bit-wise operations and counters. The algorithm consists principally of the following steps for each of the N elements:

1. Identify a mismatched pair (two input states with different outputs for the particular element).
2. Identify the mismatched bits between the two input states by performing an "exclusive or" (XOR) operation on them. (We refer to the result of this operation as an "h-vector" - short for "Hamming vector" - because it has a 1 in each position where the two states differ and a 0 where they are the same. Thus, the number of 1s in an h-vector equals the Hamming distance between the two input states.)
3. If the Hamming distance between the input states is less than a ceiling value,^l save the h-vector, else discard it.

ⁱ An element's output could be all 0s or all 1s, in which case the sample provides no information about the dynamics of the element.

^j By definition, only effective inputs can account for different element outputs.

^k The expected relative bit frequencies depend on the actual rule which, of course, is not known a priori. XOR rules are a special problem, but the use of a Hamming distance ceiling reduces it greatly.

^l By raising the Hamming distance ceiling, we reduce the variance but also reduce the expected relative frequency of the effective inputs vs. the non-effective inputs. We currently employ a cutoff value of about $0.45N$, determined by experiment.

4. If more mismatched pairs remain, return to step 1, else continue.
5. Count how often each bit position differs between members of the mismatched pairs (i.e., has a 1 in a saved h-vector).
6. Identify the bit position(s) with the largest count.^m Select this position as a predicted effective input.
7. If the set of predicted effective inputs accounts for all saved h-vectors, STOP, else return to step 5, ignoring the h-vectors with 1 in a position corresponding to one or more previously predicted effective inputs.

Our actual implementation varies slightly from these steps to achieve greater computational efficiency and to deal with other technical issues and details. For example, in practice, the portion of the sample actually needed to identify the effective inputs varies from element to element. But the number of mismatched pairs increases as the square of the sample size utilized. Hence, computational efficiency requires that we not use any more of the sample than is necessary for each element. We start by considering a minimum number of sample transitions, which depends on the largest K we expect to encounter. If the number of h-vectors used to specify any predicted effective input is judged insufficient for meaningful statistics, we add h-vectors corresponding to more of the sample and start again.

In short, the mismatched pairs algorithm identifies spikes in a series of histograms containing a decreasing number of data points. Each histogram shows how often each element appears among the differing input bits for a set of mismatched pairs. The element with the largest count in each histogram is a predicted effective input. The set of elements so identified is the predicted effective wiring. The rule is found by looking up the outputs corresponding to each input state configuration.

In common with the other reverse engineering methods for Boolean networks, the mismatched pairs algorithm is completely parallelizable. It performs exactly the same procedures N times on the same set of I-O pairs, simply using a different column on the output side in each case.

3.2 Example

We apply the mismatched pairs algorithm to predict the wiring and rule for element B in the example net shown in section 2.2. Assume the sample consists of the transitions on lines 1, 2, 3, 5, and 8. Let I_j be an input state for which $B' = j$, for $j \in \{0, 1\}$, and let H be the h-vector corresponding to a mismatched pair of input states, where $H = I_0 \text{ XOR } I_1$.

^m In the (normally rare) case of a tie, all tying inputs are selected.

Step 1: Get the mismatched pairs and h-vectors. (The first, second, and third bits in each 3-bit string represent the states of elements A, B, and C, respectively.)

I_0	I_1	H
000	001	001
000	100	100
000	111	111
010	001	011
010	100	110
010	111	101

Counting the number of 1's in each column under H we find that A, B, and C, differ between the input states I_0 and I_1 in four, three, and four pairs, respectively. Take A as having the highest count.ⁿ Thus, element A is a predicted effective input. Since the pairs in which A differs do not include all mismatched pairs, A by itself does not account for all the sample transitions of element B.

Step 2: Get the mismatched pairs and h-vectors, ignoring the pairs in which A differs.

I_0	I_1	H
000	001	001
010	001	011

In step 2 we find that element C differs between the input states I_0 and I_1 twice, versus once for element B. Thus C is a predicted input and, since C accounts for the mismatched outputs in both remaining mismatched pairs, the algorithm stops. The predicted effective architecture for element B can be summarized as $B' = A \text{ OR } C$, where the rule is found by simple table lookup in the sample state transition data. This is the correct effective architecture, as shown in reference 3, Figure 1.

3.3 Current Status

We have a working implementation of the mismatched pairs algorithm, programmed in C, and have performed a limited number of test runs. Some results are presented below.

ⁿ Here the counts for A and C are the same, which rarely occurs when larger networks and samples are employed. Rather than taking both A and C as predicted effective inputs, we choose A alone in order to show how the algorithm normally proceeds and stops.

We consider the implementation to be in an early stage of development and probably (hopefully?) far from optimal. Nevertheless, we believe the implementation is sufficiently well developed to illustrate the power of the approach.

There are many details involved in the program, including several tunable parameters. For the most part, we have chosen the parameters by trial and error experimentation, although we have done enough analytical work to convince ourselves that some parameter values can be determined analytically. Whether the potential improvements warrant the effort remains to be discovered.

3.4 Test Methods and Results

To test the algorithm, we wrote a program to generate a sample of I-O state pairs for a randomly chosen network architecture. The program also outputs a file containing the architecture corresponding to the sample. After the mismatched pairs algorithm specifies the predicted wiring and rules, the program compares the predictions to the actual architecture in order to measure performance accuracy.

It is important to note that the results presented here relate to samples consisting of randomly chosen I-O pairs, not to time series data such as the sequence of states on an attractor or on a transient. We have done very limited simulation testing using time series of network states, such as might be produced in the laboratory. This is an important subject for another time. But, clearly, the reverse engineering problem is more difficult using time series data. Element states tend to “freeze” and, therefore, a given sample size provides less information about the network architecture than does a same size random sample of I-O states. We have yet to explore to what extent using several time series starting from different initial states may ameliorate this difficulty.

The following table shows execution times, sample size, and error rates for thirteen test runs using networks with from 64 to 1,536 elements, from 3 to 5 inputs per element, and randomly chosen architectures. The sample sizes shown represent the number of I-O pairs required to achieve 99% correctly identified architectures.^o In each run, the large majority of element architectures were identified using far fewer I-O pairs than the table shows. The tests were performed on a 200 MHz PC.

^o We print out a table of the sample size required across the N elements and have arbitrarily picked out the sample size corresponding to 99% correct.

SAMPLE OF 13 TEST RUNS
 Execution time/Sample size(# of errors)

<i>N</i>	<i>K</i> = 3	<i>K</i> = 4	<i>K</i> = 5
64	1.3 sec/120(0)	4.0 sec/225(0)	9.8 sec/275(0)
256	50 sec/300(1)	-----	3.2 min/450(0)
512	-----	9 min/575(1)	-----
768	-----	22 min/600(6)	46 min/750(0)
1,024	28 min/625(0)	50 min/750(2)	1.8 hr/1000(2)
1,536	2.0 hr/850(2)	-----	4.9 hr/1000(0)

There were 14 errors out of 8,896 total predicted architectures for a 0.16% error rate (nearly half in one run). The data are merely illustrative, of course, since they represent a sample of one in each category. However, each run involves *N* repetitions of the algorithm for randomly selected architectures. We do anticipate a substantial variance in the sample size cutoff for 99% accuracy.

3.5 Discussion

Preliminary assessment indicates that the run-time complexity of the mismatched pairs algorithm is proportional to $K 2^K N^3 (\text{LOG}_2 N)^2$, which is exponential in *K* but a low order polynomial in the more critical parameter *N*. The mismatched pairs algorithm likely requires a somewhat larger sample size than an exhaustive search trial and error method, but we believe the tradeoff in run-time complexity must be considered. For example, applying a bottom-up trial and error approach to a network with *N* = 1,536 and *K* = 5, one expects to examine some 35 trillion wiring candidates per element!

If computationally tractable for a given problem, a trial and error method that tests all wiring possibilities from the bottom up until the correct one is found, is guaranteed to get the right answer.^p The mismatched pairs algorithm is a statistical method; there is some chance of error even when an exhaustive search method yields the right answer. Our experience so far suggests that erroneous predictions are few and they usually have a signature: the predicted effective architecture has a very large number of inputs and exhausts the sample without finding a complete rule. If error rates are low enough, it may be feasible to test the few “obvious” errors exhaustively, even if it is not possible to improve the algorithm to eliminate them.

^p Assuming, of course, that the data contain the complete rule for the correct wiring and that there are no degenerate solutions.

3.6 Next Steps

The mismatched pairs algorithm can no doubt be significantly improved, refined, and tested. It may also be important to investigate whether and to what extent the algorithm works if we relax the Boolean network assumptions: binary variables, synchronous updating, and state-determined transitions. Also, the other issues and problems identified in Zoltan Szallasi's paper presented at PSB99 must be addressed.⁸

The mismatched pairs algorithm could be used to study model networks in various ways. Eventually, we hope that some version of the algorithm, or at least some of the underlying concepts, will be of value in reverse engineering actual biological networks.

References

1. Akutsu, T., Miyang, S., Kuhara, S., "Identification of genetic networks from a small number of gene expression patterns under the Boolean network model", *Pacific Symposium on Biocomputing*, 1999.
2. Kauffman, S.A., *The origins of order, self-organization and selection in evolution*, Oxford University Press, 1993.
3. Liang, S., Fuhrman, S., Somogyi, R., "REVEAL, a general reverse engineering algorithm for inference of genetic network architectures", *Pacific Symposium on Biocomputing* 3:18-29, 1998.
4. Murphy, K., Mian, S., "Modelling gene expression data using dynamic Bayesian networks", *Technical Report*, Computer Science Division, University of California Life Sciences Division, Lawrence Berkeley National Laboratory, 1999.
5. Somogyi, R., Askenazi, M., "Reverse engineering genetic networks: heuristic network extraction from gene activity patterns", <http://www.santafe.edu/~manor/gene.html>.
6. Somogyi, R., Fuhrman, S., Askenazi, M., Wuensche, A., "The gene expression matrix: toward the extraction of genetic network architectures", *Proceedings of the Second World Congress of Nonlinear Analysis*, Elsevier Science, 1996.
7. Somogyi R., Sniegoski C., "Modeling the complexity of genetic networks: understanding multigenic and pleiotropic regulation", *Complexity* 1(6):45-63, 1996.
8. Szallasi, Z., "Genetic Network Analysis in Light of Massively Parallel Biological Data Acquisition", *Pacific Symposium on Biocomputing*, 4:5-16, 1999.
9. Wuensche, A., "Genomic regulation modeled as a network with basins of attraction", *Pacific Symposium on Biocomputing* 3:89-102, 1998.